ТАШКЕНТСКИЙ УНИВЕРСИТЕТ ИНФОРМАЦИОННЫХ
ТЕХНОЛОГИЙ ИМЕНИ МУХАММАДА АЛ-ХОРЕЗМИ

ФАКУЛЬТЕТ КОМПЬЮТЕРНОГО ИНЖИНИРИНГА

Кафедра "Основы информатики"

По предмету "Программирование II"
"Последовательные контейнеры
Типы последовательных контейнеров"

Методическое указание

ТАШКЕНТ 2023

Данное пособие предназначено для студентов очной формы обучения, обучающихся по специальности бакалавра, и содержит теоретические необходимые методические указания по проведению практических работ по предмету «Программирование II» и варианты практических работ. Это методическое указание может быть использовано всеми студентами, изучающими «Программирование II», а также студентами, магистрами и преподавателями Visual C++.

Разработчики:

Хайдорова М. - доцент кафедры «Основы информатики» ТУИТ

Сандов С. - ассистент кафедры «Основы информатики» ТУИТ

Аликулов А. - ассистент кафедры «Основы информатики» ТУИТ

Методическое указание рассмотрено и утверждено на общем собрании кафедры «Основы информатики» КИФ.

(Протокол № _____ «___» _____ 2023 г.)

Методическое указание обсуждено и рекомендовано к публикации научно-методическом совете факультета КИ.

(Протокол № _____ «___» _____ 2023 г.

Введение

В реальной жизни мы постоянно используем контейнеры. Гречка с куриной грудкой в контейнере для еды, страницы в книге с обложкой и переплетом, вещи в тумбочке рюкзаке и т.д. Без этих контейнеров было бы крайне неудобно работать с объектами, находящимися внутри. Представьте, что вы пытаетесь читать книгу без переплета и обложки, или пытаетесь есть гречку с грудкой, не используя контейнер для еды, миску, тарелку и т.д. Непорядок! Ценность контейнеров заключается в том, что они помогают должным образом организовать и хранить объекты. Если до сих пор мы пользовались «чистыми» массивами в языке C++, и мы многое потеряли. Под «чистыми» массивами подразумеваем обычное использование массивов в C++, без специальных функций и методов. Прочитав эту методическое пособие, вы узнаете как можно работать с массивами на более высоком уровне, вы сможете обрабатывать массивы (объявление, инициализация, поиск, сортировка и многие другие операции) буквально несколькими строчками. Контейнеры работают в динамическом и статическом порядке. Итак, что же такое «Вектор» в языке C++ простыми словами вектор можно описать как абстрактную модель, которая имитирует динамический массив. Пока не стоит углубляться в это определение, прочитав данное пособие вам все станет понятно. Библиотека контейнеров это универсальный набор шаблонов классов и алгоритмов, которые позволяют программистам легко реализовывать общие структуры данных, такие как очереди, списки и стеки. Итак, вектор C ++ это класс шаблона в стандартной библиотеке шаблонов (STL), который функционирует как более совершенный массив. В отличие от массивов, векторы могут автоматически изменять размер при вставке или удалении элементов, поэтому их удобно использовать при работе с постоянно изменяющимися данными. Использование векторов в вашей программе на C ++ позволяет вам хранить данные с большей гибкостью и эффективностью.

Последовательные контейнеры

Типы последовательных контейнеров

Последовательными контейнерами (англ. Sequence containers) в языке программирования C++ считаются несколько предопределённых шаблонных типов данных стандартной библиотеки STL, которые обеспечивают упорядоченный способ хранения своих элементов. Каждый из элементов такого контейнера имеет определённую позицию, которая зависит от времени и места помещения его в контейнер, но не зависит от значения элемента. Как правило, к последовательным контейнерам относятся *списки, вектора, очереди двустороннего доступа, массивы (начиная со стандарта C++11)* и ряд других. Обычно их реализуют на практике в виде связанного списка или массиваю

Последовательные контейнеры (или «контейнеры последовательности») — это контейнерные классы, элементы которых находятся в последовательности. Их определяющей характеристикой является то, что вы можете добавить свой элемент в любое место контейнера. Наиболее распространенным примером последовательного контейнера является массив: при добавлении 4 элементов в массив, эти элементы будут находиться (в массиве) в точно таком же порядке, в котором вы их добавили.

Начиная с C++11, STL содержит 6 контейнеров последовательности:

```
std::vector;
std::deque;
std::array;
std::list;
std::forward_list;
```

Контейнер array: До сих пор мы подробно говорили фиксированных и динамических массивах. Хотя они очень полезны и активно используются в языке C++, у них также есть свои недостатки: фиксированные массивы распадаются в указатели, теряя информацию о своей длине; в динамических массивах проблемы могут возникнуть с освобождением памяти и с попытками изменить их длину после выделения. Поэтому в Стандартную библиотеку C++ добавили функционал, который упрощает процесс управления массивами: std::array и std::vector. Сначала рассмотрим std::array, а после— std::vector.

Array Class – Описывает объект, управляющий последовательностью из элементов N типа Ty. Последовательность хранится как массив Ty в объекте array<Ty, N>

Синтаксис: template <class Ty, std::size_t N>

class array;

Параметры : Ty

Тип элемента: N

Число элементов.

```
#include <array>
std::array<int, 4> myarray; // объявляем массив типа int длиной 4
```

Подобно обычным фиксированным массивам, длина **std::array** должна быть установлена во время компиляции. std::array можно инициализировать с использованием списка инициализаторов или uniform-инициализации:

```
std::array<int, 4> myarray = { 8, 6, 4, 1 }; // список инициализаторов
std::array<int, 4> myarray2{ 8, 6, 4, 1 }; // uniform-инициализация
```

В отличие от стандартных фиксированных массивов, в std::array вы не можете пропустить (не указывать) длину массива:

```
std::array<int, > myarray = { 8, 6, 4, 1 }; // нельзя, должна быть указана длина массива
```

Также можно присваивать значения массиву с помощью списка инициализаторов:

```
std::array<int, 4> myarray;
myarray = { 0, 1, 2, 3 }; // ок
myarray = { 8, 6 }; // ок, элементам 2 и 3 присвоен нуль!
myarray = { 0, 1, 3, 5, 7, 9 }; // нельзя, слишком много элементов в списке инициализаторов!
```

Доступ к значениям массива через оператор индекса осуществляется как обычно:

```
std::cout << myarray[1];
myarray[2] = 7
```

Так же, как и в стандартных фиксированных массивах, оператор индекса не выполняет никаких проверок на диапазон. Если указан недопустимый индекс, то произойдут плохие вещи.

```
std::array поддерживает вторую форму доступа к элементам массива — функция at(), которая осуществляет проверку диапазона:
std::array<int, 4> myarray { 8, 6, 4, 1 };
myarray.at(1) = 7; // элемент массива под номером 1 - корректный, присваиваем ему значение 7
myarray.at(8) = 15; // элемент массива под номером 8 - некорректный, получим ошибку.
```

В примере, приведенном выше, вызов myarray.at(1) проверяет, есть ли элемент массива под номером 1, и, поскольку он есть, возвращается ссылка на

этот элемент. Затем мы присваиваем ему значение 7. Однако, вызов myarray.at(8) не срабатывает, так как элемента под номером 8 в массиве нет. Вместо возвращения ссылки, функция at() выдает ошибку, которая завершает работу программы (на самом деле выбрасывается исключение типа std::out_of_range. Об исключениях мы поговорим на соответствующих уроках). Поскольку проверка диапазона выполняется, то функция at() работает медленнее (но безопаснее), чем оператор [].

std::array автоматически делает все очистки после себя, когда выходит из области видимости, поэтому нет необходимости прописывать это вручную.

Размер и сортировка

С помощью функции size() можно узнать длину массива:

Пример

```
#include <iostream>
#include <array>
int main()
{
    std::array<double, 4> myarray{ 8.0, 6.4, 4.3, 1.9 };
    std::cout << "length: " << myarray.size();
    return 0;
}
```

Результат: length: 4

Заключение

std::array — это отличная замена стандартных фиксированных массивов. Массивы, созданные с помощью std::array, более эффективны, так как используют меньше памяти. Единственными недостатками **std::array** по сравнению со стандартными фиксированными массивами являются немного неудобный синтаксис и то, что нужно явно указывать длину массива (компилятор не будет вычислять её за нас). Но это сравнительно незначительные нюансы. Рекомендуется использовать std::array вместо

стандартных фиксированных массивов в любых нетривиальных задачах. Переменные типа **std::vector** могут сами управлять выделенной себе памятью (что помогает предотвратить утечку памяти), отслеживают свою длину и легко её изменяют, то рекомендуется использовать **std::vector** вместо стандартных динамических массивов.

Элементы

Определение типа	**Описание**
const_iterator	Тип постоянного итератора для управляемой последовательности.
const_pointer	Тип постоянного указателя на элемент.
const_reference	Тип постоянной ссылки на элемент.
const_reverse_iterator	Тип постоянного обратного итератора для управляемой последовательности.
difference_type	Тип расстояния со знаком между двумя элементами.
iterator	Тип итератора для управляемой последовательности
pointer	Тип указателя на элемент.
reference	Тип ссылки на элемент.
reverse_iterator	Тип обратного итератора для управляемой последовательности.
size_type	Тип без знакового расстояния между двумя элементами.
value_type	Тип элемента.
Функция-член	**Описание**
array	Создает объект массива.
assign	Устаревшие. Используйте fill .) Заменяет все элементы.

Определение типа	Описание
at	Обращается к элементу в указанной позиции.
back	Обращается к последнему элементу.
begin	Задает начало управляемой последовательности.
cbegin	Возвращает постоянный итератор произвольного доступа, указывающий на первый элемент в массиве.
cend	Возвращает постоянный итератор произвольного доступа, указывающий на предпоследнюю позицию массива.
crbegin	Возвращает константный итератор, который указывает на первый элемент в обращенном массиве.
crend	Возвращает постоянный итератор, который указывает на последний элемент в обратном массиве.
data	Получает адрес первого элемента.
empty	Проверяет наличие элементов.
end	Задает конец управляемой последовательности.
fill	Заменяет все элементы указанным значением.
front	Обращается к первому элементу.
max_size	Подсчитывает количество элементов.
rbegin	Задает начало обратной управляемой последовательности.
rend	Задает конец обратной управляемой последовательности.
size	Подсчитывает количество элементов.
swap	Меняет местами содержимое двух контейнеров.

array::crbegin. Возвращает константный итератор, который указывает на первый элемент в обращенном массиве.C++Копировать const_reverse_iterator crbegin() const;

Возвращаемое значение

Константный обратный итератор произвольного доступа, указывающий на первый элемент в обращенном массиве или на элемент, который был последним в массиве до его обращения.

Комментарии

Если возвращается значение crbegin, то объект массива невозможно изменить.

Пример

```
#include <array>
#include <iostream>
int main( )
{  using namespace std;
  array<int, 2> v1 = {1, 2};
  array<int, 2>::iterator v1_Iter;
  array<int, 2>::const_reverse_iterator v1_rIter;
  v1_Iter = v1.begin( );
  cout << "The first element of array is "
       << *v1_Iter << "." << endl;
  v1_rIter = v1.crbegin( );
  cout << "The first element of the reversed array is "
       << *v1_rIter << "." << endl;
}
The first element of array is 1.
The first element of the reversed array is 2.
```

array::crend. Возвращает константный итератор, который обращается к месту, следующему за последним элементом в обращенном массиве.

const_reverse_iterator crend( ) const noexcept;

Возвращаемое значение

Константный обратный итератор произвольного доступа, адресующий расположение после последнего элемента в обращенном массиве (расположение перед первым элементом в необращенном массиве).

Комментарии

crend используется с обратным массивом точно так же, как array::cend используется с массивом. Если возвращается значение crend (соответственно уменьшение), объект массива нельзя изменить. crend используется, чтобы проверить, достиг ли обратный итератор конца массива. Значение, возвращаемое crend, не должно быть подвергнуто удалению ссылки.

Пример

```
#include <array>
#include <iostream>
int main( )
{   using namespace std;
   array<int, 2> v1 = {1, 2};
   array<int, 2>::const_reverse_iterator v1_rIter;
   for ( v1_rIter = v1.rbegin( ) ; v1_rIter != v1.rend( ) ; v1_rIter++ )
      cout << *v1_rIter << endl;
}
```

Результат

```
2
1
```

array::data

Получает адрес первого элемента.

Ty *data( );

const Ty *data( ) const;

Комментарии

Функции-члены возвращают адрес первого элемента в управляемой последовательности.

Пример

```
#include <array>
#include <iostream>
typedef std::array<int, 4> Myarray;
int main()
{   Myarray c0 = { 0, 1, 2, 3 };
  // display contents " 0 1 2 3"
  for (const auto& it : c0)
  {
    std::cout << " " << it;
  }
  std::cout << std::endl;
  // display first element " 0"
  Myarray::pointer ptr = c0.data();
  std::cout << " " << *ptr;
  std::cout << std::endl;
  return (0);
}
```

Результат

```
0 1 2 3
0
```

array::difference_type

typedef std::ptrdiff_t difference_type;

Комментарии

Тип целого числа со знаком описывает объект, который может представлять разницу между адресами любых двух элементов в управляемой последовательности. Это синоним для типа std::ptrdiff_t.

Пример

```
#include <array>
#include <iostream>
typedef std::array<int, 4> Myarray;
int main()
{
   Myarray c0 = { 0, 1, 2, 3 };
   // display contents " 0 1 2 3"
   for (const auto& it : c0)
   {
      std::cout << " " << it;
   }
   std::cout << std::endl;
   // display distance first-last " -4"
   Myarray::difference_type diff = c0.begin() - c0.end();
   std::cout << " " << diff;
   std::cout << std::endl;
   return (0);
}
```

Результат

```
0 1 2 3
-4
```

array::empty

Проверяет отсутствие элементов.

constexpr bool empty() const;

Комментарии

Функция-член возвращает значение true, только если N == 0.

Пример

```
#include <array>
#include <iostream>
typedef std::array<int, 4> Myarray;
int main()
{
   Myarray c0 = { 0, 1, 2, 3 };
   // display contents " 0 1 2 3"
   for (const auto& it : c0)
   {
      std::cout << " " << it;
   }
   std::cout << std::endl;
   // display whether c0 is empty " false"
   std::cout << std::boolalpha << " " << c0.empty();
   std::cout << std::endl;
   std::array<int, 0> c1;
   // display whether c1 is empty " true"
   std::cout << std::boolalpha << " " << c1.empty();
   std::cout << std::endl;
   return (0);
}
```

Результат

```
0 1 2 3
false
true
```

array::end

Задает конец управляемой последовательности.

reference end( );

const_reference end( ) const;

Комментарии

Первые две функции-члена возвращают итератор произвольного доступа, указывающий на место сразу за концом последовательности.

Пример

```
#include <array>
#include <iostream>
typedef std::array<int, 4> Myarray;
int main( )
{   Myarray c0 = { 0, 1, 2, 3 };
   // display contents " 0 1 2 3"
   for (const auto& it : c0)
   {
      std::cout << " " << it;
   }
   std::cout << std::endl;
   // display last element " 3"
   Myarray::iterator it2 = c0.end( );
   std::cout << " " << *--it2;
   std::cout << std::endl;
   return (0);
}
```

OutputКопировать

```
0 1 2 3
3
```

array::fill

Удаляет массив и копирует указанные элементы в пустой массив.

void fill(const Type& val);

Параметры

val

Значение элемента, вставляемого в массив.

Комментарии

fill заменяет каждый элемент массива на указанное значение.

Пример

```
#include <array>
#include <iostream>
int main()
{
   using namespace std;
   array<int, 2> v1 = { 1, 2 };
   cout << "v1 = ";
   for (const auto& it : v1)
   {
      std::cout << " " << it;
   }
   cout << endl;
   v1.fill(3);
   cout << "v1 = ";
   for (const auto& it : v1)
   {
      std::cout << " " << it;
   }
   cout << endl;
}
```

array::front

Обращается к первому элементу.

reference front( );

constexpr const_reference front( ) const;

Комментарии

Функции-члены возвращают ссылку на первый элемент управляемой последовательности, который должен быть не пустым.

Пример

```
#include <array>
#include <iostream>
typedef std::array<int, 4> Myarray;
int main( )
{
   Myarray c0 = { 0, 1, 2, 3 };
   // display contents " 0 1 2 3"
   for (const auto& it : c0)
   {
      std::cout << " " << it;
   }
   std::cout << std::endl;
   // display first element " 0"
   std::cout << " " << c0.front( );
   std::cout << std::endl;
   return (0);
}
```

Результат

```
0 1 2 3
0
```

array::iterator

Тип итератора для управляемой последовательности.

typedef implementation-defined iterator;

Комментарии

Этот тип описывает объект, который можно использовать в качестве итератора с произвольным доступом для управляемой последовательности.

Пример

```
#include <array>
#include <iostream>
typedef std::array<int, 4> MyArray;
int main()
{
  MyArray c0 = { 0, 1, 2, 3 };
  // display contents " 0 1 2 3"
  std::cout << "it1:";
  for (MyArray::iterator it1 = c0.begin();
    it1 != c0.end();
    ++it1) {
    std::cout << " " << *it1;
  }
  std::cout << std::endl;
  // display first element " 0"
  MyArray::iterator it2 = c0.begin();
  std::cout << "it2:";
  std::cout << " " << *it2;
  std::cout << std::endl;
  return (0);
}
```

Результат

```
it1: 0 1 2 3
it2: 0
```

array::max_size

Подсчитывает количество элементов.

constexpr size_type max_size() const;

Комментарии

Функция-член возвращает значение N.

Пример

```
#include <array>
#include <iostream>
typedef std::array<int, 4> Myarray;
int main()
{
   Myarray c0 = { 0, 1, 2, 3 };
   // display contents " 0 1 2 3"
   for (const auto& it : c0)
   {
      std::cout << " " << it;
   }
   std::cout << std::endl;
   // display (maximum) size " 4"
   std::cout << " " << c0.max_size();
   std::cout << std::endl;
   return (0);
}
```

Результат

```
0 1 2 3
4
```

array::operator[]

Обращается к элементу в указанной позиции.

reference operator[](size_type off)

constexpr const_reference operator[](size_type off) const;

Параметры

off

Позиция элемента, к которому осуществляется доступ.

Комментарии

Функции-члены возвращают ссылку на элемент управляемой последовательности в позиции *off*. Если эта позиция недопустима, поведение станет неопределенным.

Также существует функция, не относящаяся к члену get, для получения ссылки на элемент array.

Пример

```
#include <array>
#include <iostream>
typedef std::array<int, 4> Myarray;
int main()
{
  Myarray c0 = { 0, 1, 2, 3 };
  // display contents " 0 1 2 3"
  for (const auto& it : c0)
  {
    std::cout << " " << it;
  }
  std::cout << std::endl;
  // display odd elements " 1 3"
  std::cout << " " << c0[1];
```

```
    std::cout << " " << c0[3];
    std::cout << std::endl;
    return (0);
};
```

Результат

```
0 1 2 3
1 3
```

array::operator=

Заменяет управляемую последовательность.

array<Value> operator=(array<Value> right);

Параметры

right

Контейнер для копирования.

Комментарии

Оператор-член присваивает каждому элементу *right* соответствующий элемент управляемой последовательности, а затем возвращает ***this**. Он позволяет заменить управляемую последовательность копией управляемой последовательности в *right*.

Пример

```
#include <array>
#include <iostream>
typedef std::array<int, 4> Myarray;
int main()
{
    Myarray c0 = { 0, 1, 2, 3 };
    // display contents " 0 1 2 3"
    for (const auto& it : c0)
    {
```

```
    std::cout << " " << it;
  }
  std::cout << std::endl;
  Myarray c1;
  c1 = c0;
  // display copied contents " 0 1 2 3"
    // display contents " 0 1 2 3"
  for (auto it : c1)
  {
    std::cout << " " << it;
  }
  std::cout << std::endl;
  return (0);
}
```

Результат

```
0 1 2 3
0 1 2 3
```

array::pointer

Тип указателя на элемент.

typedef Ty *pointer;

Комментарии

Этот тип описывает объект, который можно использовать в качестве указателя на элементы последовательности.

Пример

```
#include <array>
#include <iostream>
typedef std::array<int, 4> Myarray;
int main()
```

```
{
  Myarray c0 = { 0, 1, 2, 3 };
  // display contents " 0 1 2 3"
  for (const auto& it : c0)
  {
    std::cout << " " << it;
  }
  std::cout << std::endl;
  // display first element " 0"
  Myarray::pointer ptr = &*c0.begin();
  std::cout << " " << *ptr;
  std::cout << std::endl;
  return (0);
}
```

Результат

```
0 1 2 3
0
```

array::rbegin

Задает начало обратной управляемой последовательности.

reverse_iterator rbegin()noexcept;

const_reverse_iterator rbegin() const noexcept;

Комментарии

Первые две функции-члена возвращают обратный итератор, указывающий на место сразу за концом управляемой последовательности. Таким образом, задается начало обратной последовательности.

Пример

```
#include <array>
#include <iostream>
```

```
typedef std::array<int, 4> Myarray;
int main()
{   Myarray c0 = { 0, 1, 2, 3 };
   // display contents " 0 1 2 3"
   for (const auto& it : c0)
   {
      std::cout << " " << it;
   }
   std::cout << std::endl;
   // display last element " 3"
   Myarray::const_reverse_iterator it2 = c0.rbegin();
   std::cout << " " << *it2;
   std::cout << std::endl;
   return (0);}
```

Результат

```
0 1 2 3
3
```

array::reference

Тип ссылки на элемент.

typedef Ty& reference;

Комментарии

Тип описывает объект, который можно использовать в качестве ссылки на элемент управляемой последовательности.

Пример

```
#include <array>
#include <iostream>
typedef std::array<int, 4> Myarray;
```

```
int main()
{
   Myarray c0 = { 0, 1, 2, 3 };
   // display contents " 0 1 2 3"
   for (const auto& it : c0)
   {
      std::cout << " " << it;
   }
   std::cout << std::endl;
   // display first element " 0"
   Myarray::reference ref = *c0.begin();
   std::cout << " " << ref;
   std::cout << std::endl;
   return (0);
}
```

Результат

```
0 1 2 3
0
```

array::rend

Задает конец обратной управляемой последовательности.

reverse_iterator rend()noexcept;

const_reverse_iterator rend() const noexcept;

Комментарии

Функции-члены возвращают обратный итератор, указывающий на первый элемент последовательности (или непосредственно за окончание пустой последовательности)). Таким образом, он задает конец обратной последовательности.

Пример

```
#include <array>
#include <iostream>
typedef std::array<int, 4> Myarray;
int main()
{
  Myarray c0 = { 0, 1, 2, 3 };
  // display contents " 0 1 2 3"
  for (const auto& it : c0)
  {
    std::cout << " " << it;
  }
  std::cout << std::endl;
  // display first element " 0"
  Myarray::const_reverse_iterator it2 = c0.rend();
  std::cout << " " << *--it2;
  std::cout << std::endl;
  return (0);
}
```

Результат

```
0 1 2 3
0
```

array::reverse_iterator

Тип обратного итератора для управляемой последовательности.

typedef std::reverse_iterator<iterator> reverse_iterator;

Комментарии

Этот тип описывает объект, который можно использовать в качестве обратного итератора для управляемой последовательности.

Пример

```
#include <array>
#include <iostream>
typedef std::array<int, 4> Myarray;
int main()
{
   Myarray c0 = { 0, 1, 2, 3 };
   // display contents " 0 1 2 3"
   for (const auto& it : c0)
   {
      std::cout << " " << it;
   }
   std::cout << std::endl;
   // display last element " 3"
   Myarray::reverse_iterator it2 = c0.rbegin();
   std::cout << " " << *it2;
   std::cout << std::endl;
   return (0);
}
```

Результат

```
0 1 2 3
3
```

array::size

Подсчитывает количество элементов.

constexpr size_type size() const;

Комментарии

Функция-член возвращает значение N.

Пример

```
#include <array>
#include <iostream>
typedef std::array<int, 4> Myarray;
int main()
{
   Myarray c0 = { 0, 1, 2, 3 };
   // display contents " 0 1 2 3"
   for (const auto& it : c0)
   {
      std::cout << " " << it;
   }
   std::cout << std::endl;
   // display size " 4"
   std::cout << " " << c0.size();
   std::cout << std::endl;
   return (0);
}
```

Результат

```
0 1 2 3
4
```

array::size_type

Тип беззнакового расстояния между двумя элементами.

typedef std::size_t size_type;

Комментарии

Целочисленный тип без знака описывает объект, который может представлять длину любой управляемой последовательности. Это синоним для типа std::size_t.

Пример

```
#include <array>
#include <iostream>
typedef std::array<int, 4> Myarray;
int main()
{
   Myarray c0 = { 0, 1, 2, 3 };
   // display contents " 0 1 2 3"
   for (const auto& it : c0)
   {
      std::cout << " " << it;
   }
   std::cout << std::endl;
   // display distance last-first " 4"
   Myarray::size_type diff = c0.end() - c0.begin();
   std::cout << " " << diff;
   std::cout << std::endl;
   return (0);}
```

Результат

```
0 1 2 3
4
```

array::swap

Выполняет обмен содержимым между этим и другим массивом

void swap(array& right);

Параметры

right

Массив для обмена содержимым.

Комментарии

Функция – член меняет местами управляемые последовательности между ***this** и *вправо*. Он выполняет присваивания элементов и вызовы конструктора у количестве, пропорциональном пропорциональном N.

Также существует функция, не относящаяся к члену swap , для переключения двух **array** экземпляров.

Пример

```
#include <array>
#include <iostream>
typedef std::array<int, 4> Myarray;
int main()
{
  Myarray c0 = { 0, 1, 2, 3 };
  // display contents " 0 1 2 3"
  for (const auto& it : c0)
  {
    std::cout << " " << it;
  }
  std::cout << std::endl;
  Myarray c1 = { 4, 5, 6, 7 };
  c0.swap(c1);
  // display swapped contents " 4 5 6 7"
  for (const auto& it : c0)
  {
    std::cout << " " << it;
  }
  std::cout << std::endl;
  swap(c0, c1);
  // display swapped contents " 0 1 2 3"
```

```
    for (const auto& it : c0)
    {
        std::cout << " " << it;
    }
    std::cout << std::endl;
    return (0);
}
```

Результат

```
0 1 2 3
4 5 6 7
0 1 2 3
```

array::value_type

Тип элемента.

typedef Ty value_type;

Комментарии

Этот тип является синонимом для параметра шаблона Ty.

Пример

```
#include <array>
#include <iostream>
typedef std::array<int, 4> Myarray;
int main()
{   Myarray c0 = { 0, 1, 2, 3 };
    // display contents " 0 1 2 3"
    for (const auto& it : c0)
    {
        std::cout << " " << it;
    }
    std::cout << std::endl;
```

```
    // display contents " 0 1 2 3"
    for (const auto& it : c0)
    {
        Myarray::value_type val = it;
        std::cout << " " << val;
    }
    std::cout << std::endl;
    return (0);
}
```

Результат

```
0 1 2 3
0 1 2 3
```

Класс vector (или просто «вектор») — это динамический массив, способный увеличиваться по мере необходимости для содержания всех своих элементов. Класс vector обеспечивает произвольный доступ к своим элементам через оператор индексации [], а также поддерживает добавление и удаление элементов. В следующей программе мы добавляем 5 целых чисел в вектор и с помощью перегруженного оператора индексации [ ] получаем к ним доступ для их последующего вывода.

```
#include <iostream>
#include <vector>
int main()
{
    std::vector<int> vect;
    for (int count=0; count < 5; ++count)
        vect.push_back(10 - count); // добавляем числа в конец массива
    for (int index=0; index < vect.size(); ++index)
```

```
    std::cout << vect[index] << ' ';
  std::cout << '\n';
}
```

Результат выполнения программы:

```
10 9 8 7 6
```

Что vector в C ++?

Контейнер vector ведет себя как массив, но может автоматически увеличиваться по мере необходимости. Он поддерживает прямой доступ и связанное хранение и имеет очень гибкую длину. По этим и многим другим причинам контейнер vector является наиболее предпочтительным последовательным контейнером для большинства областей применения. Векторные контейнеры могут выделять дополнительное пространство для хранения, чтобы приспособиться к потенциальному росту, поэтому контейнер может иметь большую емкость, чем то, что на самом деле хранится внутри него. Вы можете использовать библиотеки для применения различных стратегий роста, чтобы найти баланс между перераспределением и использованием памяти, но перераспределение должно происходить только с логарифмически увеличивающимися интервалами размера. Это позволяет использовать амортизированную постоянную временную сложность при вставке отдельных элементов в конец вектора.

Поскольку векторы C ++ выполняют автоматическое перераспределение, они могут эффективно управлять хранилищем и динамически расти. Хотя они потребляют больше памяти, чем массивы, их эффективность и гибкость делают использование векторов целесообразным.

Давайте обсудим еще несколько преимуществ C ++ vector:

- Максимального размера нет, что полезно, если вы заранее не знаете, насколько велики будут ваши данные.

- Их возможности изменения размера упрощают работу с динамическими элементами.
- Поскольку вектор С ++ — это шаблонный класс, вам не нужно вводить один и тот же код для обработки разных данных.
- Когда используется общий объем памяти, происходит автоматическое перераспределение.
- Вы можете легко копировать и назначать другие векторы.

И так, ВЕКТОР — это структура данных, которая уже является моделью динамического массива.

Давайте вспомним о том, что для создания динамического массива (вручную) нам нужно пользоваться конструктором new и вдобавок указателями. Но в случае с векторами всего этого делать не нужно.

Вообще, по стандарту пользоваться динамическим массивом через конструктор new — не есть правильно. Так как в компьютере могут происходить различные утечки памяти.

Как создать вектор (vector) в С++

Сначала для создания вектора нам понадобится подключить библиотеку — <vector>, в ней хранится шаблон вектора.

```
#include <vector>
```

Кстати, сейчас и в будущем мы будем использовать именно шаблон вектора. Например, очередь или стек, не созданные с помощью массива или вектора, тоже являются шаблонными.

Далее, чтобы объявить вектор, нужно пользоваться конструкцией ниже:

```
vector < тип данных > <имя вектора>;
```

- Вначале пишем слово vector.

- Далее в угольных скобках указываем тип, которым будем заполнять ячейки.
- И в самом конце указываем имя вектора.

```
vector <string> ivector;
```

В примере выше мы создали вектор строк.

Кстати, заполнить вектор можно еще при инициализации (другие способы мы пройдем позже — в методах вектора). Делается это также просто, как и в массивах. Вот так:

```
vector <int> ivector = {<элемент [0]>, <[1]>, <[2]>};
```

После имени вектора ставим знак равенства и скобки, в которых через пробел указываем значение элементов.

Такой способ инициализации можно использовать только в C++!

Так, чтобы заполнить вектор строками, нам нужно использовать кавычки — "строка".

```
... = {"C", "+", "+"};
```

Но мы можем инициализировать вектор одним из следующих способов:

```
vector<int> v1;            // пустой вектор
vector<int> v2(v1);        // вектор v2 - копия вектора v1
vector<int> v3 = v1;       // вектор v3 - копия вектора v1
```

```
vector<int> v4(5);          // вектор v4 состоит из 5 чисел
vector<int> v5(5, 2);       // вектор v5 состоит из 5 чисел, каждое
число равно 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, 4, 5
```

Важно понимать отличие в данном случае круглых скобок от фигурных:

```
vector<int> v1(5); // вектор состоит из 5 чисел, каждое число в векторе
равно 0
vector<int> v2{5} //вектор состоит из одного числа, которое равно 5
vector<int> v3(5, 2); // вектор состоит из 5 чисел, каждое число равно
2
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:

```
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 возвращает элемент по индексу
- **front()**: возвращает первый элемент
- **back()**: возвращает последний элемент

Выполним перебор вектора и получим некоторые его элементы:

Пример

```
#include <iostream>
#include <vector>
int main()
{vector<int> numbers {1, 2, 3, 4, 5};
    int n1 = numbers.front(); // n1 = 1
  int n2 = numbers.back(); // n2 = 5
  int x = numbers[1]; // x = 2
  numbers[0] = 6;
    for(int n : numbers)
    cout << n << " "; // 6 2 3 4 5
    cout << std::endl;
  return 0;
}
```

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

```
vector<int> numbers {1, 2, 3, 4, 5};
numbers[5] = 9;
```

При таком обращении результат неопределен. Некоторые компиляторы 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:

Пример

```
#include <iostream>
#include <vector>
int main()
{   vector<int> numbers {1, 2, 3, 4, 5};
    try {
        int n = numbers.at(8);
    }
    catch (out_of_range e)
    {   cout << "Incorrect index" << endl;  }
    return 0;  }
```

Операции с векторами

Добавление элементов в вектор

Для добавления элементов в вектор применяется функция **push_back()**, в который передается добавляемый элемент:

Пример

```
#include <iostream>
#include <vector>
using namespace std;
int main()
{   vector<int> numbers;        пустой вектор
    numbers.push_back(5);
    numbers.push_back(3);
    numbers.push_back(10);
    for(int n : numbers)
        cout << n << " ";       5  3  10
```

```
    cout << endl;
    return 0;
}
```

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:

```
vector<int> numbers1 = { 1, 2, 3, 4, 5 };
numbers1.emplace_back(8);   // numbers1 = { 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**:

Пример

```
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**:

Пример

```
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};
 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};
 vector<int> values = { 10, 20, 30, 40, 50 };
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};
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:
vector<int> v = { 1,2,3,4 };
v.clear();
Функция pop_back() удаляет последний элемент вектора:
    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 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:

Пример

```
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 }
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}
```

Размер вектора

С помощью функции size() можно узнать размер вектора, а с помощью функции empty() проверить, пустой ли вектор.

Пример

```
#include <iostream>
using namespace std;
int main()
{
   vector<int> numbers {1, 2, 3};
   if(numbers.empty())
      cout << "Vector is empty" << endl;
   else
      cout << "Vector has size " << numbers.size() << endl;
   return 0;}
```

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:

Пример

```
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:

Пример

```
vector<string> names = { "Tom", "Bob", "Kate"};
names.assign(4, "Sam"); // numbers = {"Sam" "Sam", "Sam", "Sam"}
```

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

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

Пример

```
vector<string> clangs = { "C++", "C#", "Java" };
vector<string> ilangs = { "JavaScript", "Python", "PHP"};
clangs.swap(ilangs);   // clangs = { "JavaScript", "Python", "PHP"};
for(string n : clangs)
 cout << n << "\t";
cout << endl;
```

Сравнение векторов

Векторы можно сравнивать. 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:

Пример

```
vector<int> v1 = {1, 2, 3};
vector<int> v2 = {1, 2, 3};
vector<int> v3 = {3, 2, 1};
 bool v1v2 = v1 == v2;  // true
bool v1v3 = v1 != v3;  // true
bool v2v3 = v2 == v3;  // false
```

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

```
#include <vector>
```

Определим простейший вектор:

```
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:

Пример

```
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
std::vector<int> v5(5, 2);        // вектор v5 состоит из 5 чисел, каждое число равно 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, 4, 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 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
```

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:

```
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 возвращает элемент по индексу
- **front():** возвращает первый элемент
- **back():** возвращает последний элемент

Выполним перебор вектора и получим некоторые его элементы:

Пример

```
#include <iostream>
#include <vector>
int main()
{ vector<int> numbers {1, 2, 3, 4, 5};
  int n1 = numbers.front();     n1   1
  int n2 = numbers.back();      n2   5
  int x = numbers[1];           x    2
  numbers[0] = 6;
     for(int n : numbers)
     cout << n << " ";    6 2 3 4 5
     cout << endl;
  return 0;}
```

Векторный класс стандартной библиотеки C ++ - это шаблон класса для контейнеров последовательностей. Вектор хранит элементы данного типа в линейном порядке и обеспечивает быстрый произвольный доступ к любому элементу. Вектор является предпочтительным контейнером для последо - вательности, когда производительность произвольного доступа высока.

Класс deque (или просто «дек») — это двусторонняя очередь, реализованная в виде динамического массива, который может расти с обоих концов. Например:

Пример

```
#include <iostream>
#include <deque>
 int main()
{
   std::deque<int> deq;
   for (int count=0; count < 4; ++count)
   {
      deq.push_back(count); // вставляем числа в конец массива
      deq.push_front(10 - count); // вставляем числа в начало массива
   }
   for (int index=0; index < deq.size(); ++index)
      std::cout << deq[index] << ' ';
   std::cout << '\n';
}
```

Результат выполнения программы:

7 8 9 10 0 1 2 3

List (или просто «список») — это двусвязный список, каждый элемент которого содержит 2 указателя: один указывает на следующий элемент списка, а другой — на предыдущий элемент списка. List предоставляет доступ только к началу и к концу списка — произвольный доступ запрещен. Если вы хотите найти значение где-то в середине, то вы должны начать с одного конца и перебирать каждый элемент списка до тех пор, пока не найдете то, что ищете. Преимуществом двусвязного списка является то, что добавление элементов происходит очень быстро, если вы, конечно, знаете, куда хотите добавлять. Обычно для перебора элементов двусвязного списка используются итераторы.

Пример

```
#include <iostream>
#include <deque>
 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, 4, 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}
```

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()**:

Пример

```
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 }
```

Хотя о классе string (и wstring) обычно не говорят, как о последовательном контейнере, но он, по сути, таковым является, поскольку его можно рассматривать как вектор с элементами типа char (или wchar).

Ассоциативные контейнеры

Ассоциативные контейнеры — это контейнерные классы, которые автоматически сортируют все свои элементы (в том числе и те, которые добавляете вы). По умолчанию ассоциативные контейнеры выполняют сортировку элементов, используя оператор сравнения <.

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

multiset — это set, но в котором допускаются повторяющиеся элементы.

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

multimap (или «словарь») — это map, который допускает дублирование ключей. Все ключи отсортированы в порядке возрастания, и вы можете посмотреть значение по ключу.

Адаптеры

Адаптеры — это специальные предопределенные контейнерные классы, которые адаптированы для выполнения конкретных заданий. Самое

интересное заключается в том, что вы сами можете выбрать, какой последовательный контейнер должен использовать адаптер.

stack (стек) — это контейнерный класс, элементы которого работают по принципу LIFO (англ. «Last In, First Out» = «последним пришел, первым ушел»), т.е. элементы добавляются (вносятся) в конец контейнера и удаляются (выталкиваются) оттуда же (из конца контейнера). Обычно в стеках используется deque в качестве последовательного контейнера по умолчанию (что немного странно, поскольку vector был бы более подходящим вариантом), но вы также можете использовать vector или list.

queue (очередь) — это контейнерный класс, элементы которого работают по принципу FIFO (англ. «First In, First Out» = «первым пришел, первым ушел»), т.е. элементы добавляются (вносятся) в конец контейнера, но удаляются (выталкиваются) из начала контейнера. По умолчанию в очереди используется deque в качестве последовательного контейнера, но также может использоваться и list.

priority_queue (очередь с приоритетом) — это тип очереди, в которой все элементы отсортированы (с помощью оператора сравнения <). При добавлении элемента, он автоматически сортируется. Элемент с наивысшим приоритетом (самый большой элемент) находится в самом начале очереди с приоритетом, также, как и удаление элементов выполняется с самого начала очереди с приоритетом.

ПРАКТИЧЕСКАЯ РАБОТА .

ПОСЛЕДОВАТЕЛЬНЫЕ КОНТЕЙНЕРЫ БИБЛИОТЕКИ STL

Цель работы: Изучить последовательные контейнеры библиотеки stl и разработка алгоритмов выполнения задач с этими списками.

Поставленная задача: Создание элементов последовательного контейнера на языке Visual C++, разработка программы выполнения различных действий на нем.

Порядок работы:

Изучение теоретических данных лабораторной работы.

Разработка алгоритма заданной задачи.

Создание программы в среде программирования Visual C++.

Проверка результатов.

Подготовка и представление отчета для лабораторной работы

ТЕОРЕТИЧЕСКИЕ МАТЕРИАЛЫ

Статический непрерывный массив *array*

Объекты типа array представляют из себя массивы фиксированного размера: они имеют определенное количество элементов, которые идут друг за другом, в строгой линейной последовательности, как у обычных массивов в C++. В рамках данной статьи вместо объекта типа array, буду говорить — массив, но имею ввиду необычный массив C++, а, именно, экземпляр класса array

Внутри объекта array хранятся только его элементы, никаких других данных объект array не хранит (даже свой размер, который является параметром шаблона). Это связано с эффективным использованием памяти, объект array не должен занимать больше памяти, чем обычный массив. Этот класс просто добавляет свой член и глобальными функциями, так что массивы могут быть использованы в качестве стандартных контейнеров. В отличие от других стандартных контейнеров, контейнер array имеет фиксированный размер и не позволяет управлять распределением памяти под свои элементы через allocator. Экземпляр класса-контейнера array представляет собой —

массив элементов фиксированного размера. таким образом. размер массива не может быть изменен динамически (см. vector — на аналогичный контейнер, размер которого может и меняться динамически). Нулевой размер экземпляра array является действительным и не будет считаться ошибкой, но такой массив не может быть разименован(методы front. back. data).

Все элементы класса-контейнера array расположены в строгой линейной последовательности. друг за другом. Отдельные элементы доступны в своих позициях в этой последовательности. Элементы массива хранятся в смежных ячейках памяти. что позволяет в любой момент времени получить доступ к любому элементу. используя указатели и смещение относительно текущего элемента, можно получить доступ к другим элементам массива.

Контейнер array использует неявный конструктор для выделения необходимой. статической области в памяти. Размер этой области является константным значением времени компиляции. Контейнер array представляет аналог массива. Он также имеет фиксированный размер.

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

```
std::array<int, 6> numbers;      // состоит из 6 чисел
std::array<std::string. 5> strings;  // состоит из 5 строк "}
```

Фиксированный размер накладывает ограничение на инициализацию и использование подобных контейнеров. В частности. для инициализации мы не можем использовать стандартные для большинства контейнеров конструкторы. А при списочной инициализации количество передаваемых контейнеру элементов не должно превышать его размер:

```
#include <iostream>
```

```
#include <array>
#include <string>
int main()
{
    std::array<int, 6> numbers = { 1, 2, 3, 4, 5, 6 };
    std::array<std::string, 5> names = { "Tom", "Alice", "Kate", "Bob",
"Sam" };
    for (auto n : numbers)
    {
        std::cout << n << "\t";
    }
    std::cout << std::endl;
    return 0;
}
```

Если при инициализации мы передадим меньшее количество элементов, то для недостающих элементов будут использоваться значения по умолчанию (например, для целочисленных типов это число 0). Однако если при инициализации мы предадим большее количество элементов, нежели размер контейнера, то мы столкнемся с ошибкой.

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, 5> names = { "Tom", "Alice", "Kate", "Bob",
"Sam" };
    std::string third = names.at(2);   // Kate
    std::string first = names.front(); // Tom
    std::string last = names.back();   // Sam
    std::cout << third << std::endl;
    std::cout << first << std::endl;
    std::cout << last << std::endl;
    names.fill("Tim"); // names = { "Tim", "Tim", "Tim", "Tim", "Tim" }
    for (int i = 0; i<names.size(); i++)
    {
        std::cout << names[i] << std::endl;
    }
    return 0;
}
```

Несмотря на то, что объекты array похожи на обычные массивы, однако 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 сделать
```

Динамический непрерывный массив *vector*

Вектор в C++ — 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 создания вектора

Управление элементами вектора

Методы класса vector

Пример создания вектора

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

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

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

Управление элементами вектора

Создадим вектор, в котором будет содержаться произвольное количество фамилий студентов.

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

int main()
{
  // Поддержка кириллицы в консоли Windows
  setlocale(LC_ALL, "");
  // Создание вектора из строк
  std::vector<std::string> students;
  // Буфер для ввода фамилии студента
  std::string buffer = "";
  std::cout << "Вводите фамилии студентов. "
        << "По окончание ввода введите пустую строку" << std::endl;

  do {
    std::getline(std::cin, buffer);
    if (buffer.size() > 0) {
```

```
        // Добавление элемента в конец вектора
        students.push_back(buffer);
    }
} while (buffer != "");
// Сохраняем количество элементов вектора
unsigned int vector_size = students.size();

// Вывод заполненного вектора на экран
std::cout << "Ваш вектор." << std::endl;
for (int i = 0; i < vector_size; i++) {
    std::cout << students[i] << std::endl;
}
return 0;
}
```

Результат работы программы:

```
Вводите фамилии студентов. По окончание ввода введите пустую строку
Иванов Иван Иванович
Петров Петр Петрович
Сидоров Александр Александрович
Попов Андрей Андреевич

Ваш вектор.
Иванов Иван Иванович
Петров Петр Петрович
Сидоров Александр Александрович
Попов Андрей Андреевич
```

Рисунок 1. Результат программы

Методы класса vector

Для добавления нового элемента в конец вектора используется метод push_back(). Количество элементов определяется методом size(). Для

доступа к элементам вектора можно использовать квадратные скобки [], также, как и для обычных массивов.

pop_back() — удалить последний элемент

push_back() — добавить элемент в конец

clear() — удалить все элементы вектора

empty() — проверить вектор на пустоту

vector :: pop_back

```
// vector::push_back
#include <iostream>
#include <vector>
int main ()
{
  std::vector<int> myvector;
  int myint;

  std::cout << "Please enter some integers (enter 0 to end):\n";

  do {
    std::cin >> myint;
    myvector.push_back (myint);
  } while (myint);

  std::cout << "myvector stores " << int(myvector.size()) << " numbers.\n";
  return 0;
}
```

Добавить элемент в конце:

Добавляет новый элемент в конец вектора после его текущего последнего элемента. Содержимое *val* копируется (или перемещается) в новый элемент. Это эффективно увеличивает размер контейнера на единицу, что вызывает автоматическое перераспределение выделенного пространства хранения, если - и только если - новый размер вектора превышает текущую емкость вектора

Пример

```
// vector::push_back
#include <iostream>
#include <vector>
int main ()
{
  std::vector<int> myvector;
  int myint;

  std::cout << "Please enter some integers (enter 0 to end):\n";
  do {
    std::cin >> myint;
    myvector.push_back (myint);
  } while (myint);
  std::cout << "myvector stores " << int(myvector.size()) << " numbers.\n";
  return 0;
}
```

Практическая работа . Библиотека STL. Контейнеры. Последовательные контейнеры array, vector, list, deque.

Задание 1. Выполнить следующее:

1. Создать объект-контейнер в соответствии с вариантом задания и заполнить его данными в указанном количестве и в интервале с помощью генератора случайных чисел , тип которых определяется вариантом задания.

2. Просмотреть контейнер.

3. Изменить контейнер, удалив из него (см. таб.) элемент

4. Затем в конец добавит (см. таб.) элемент.

5. Просмотреть контейнер, используя для доступа к его элементам итераторы.

6. Изменить первый контейнер, удалив из него N-7 элементов до заданного N (если N>8) или N+3 элементов после заданного N.

7. Создать второй контейнер этого же класса (с размерностью 1-го после выполнения 6 пункта) и заполнить его данными того же типа, что и первый контейнер.

8. Сравнить оба контейнера.

9. Меньшее из них добавит к в конец большего.

10. Просмотрет оба контейнера.

Примечание. N – соответсвует порядковому номеру в таблице.

№	1-й контейнер	Тип данных	Кол-во элементов	Диапазон значений	Пункт 3	Пункт 4
1	vector	int	20	[0;100]	Минимальный	Среднее значение
2	list	long	30	[100;1000]	Максимальный	Среднее значение
3	deque	float	25	[1;50] с точностью 0.1	6-й	Максимальный

4	vector	double	30	[100;500] с точностью 0.01	16-й	Минимальный
5	list	long	25	[0;100]	Максимальный	Среднее значение
6	deque	float	20	[100;1000]	4-й	Среднее значение
7	deque	int	30	[-100;100]	12-й	Минимальный
8	vector	float	30	100;500] с точностью 0.01	Максимальный	Среднее значение
9	list	int	25	[0;100]	17-й	Минимальный
10	list	float	20	100;500] с точностью 0.01	9-й	Максимальный
11	deque	double	30	[100;1000]	5-й	Среднее значение
12	vector	long	25	[-100;100]	10-й	Минимальный
13	deque	float	20	[1;50] с точностью 0.1	Минимальный	Среднее значение
14	vector	int	45	[-50;50]	11-й	Среднее значение
15	list	float	25	[1;50] с точностью 0.1	2-й	Минимальный

Задание 2. В бухгалтерии предприятия хранится сведение о заработанной платы каждого сотрудника за 3 года. Создайте программу, для определения максимального среднего заработка за N месяцев (последовательно идущее), используя контейнер array. (Оклад работника сдельная, и меняется от 800 000 до 3 500 000)

Задание 3 Ту же №2 задачу решите используя контейнер указаной в таблице . Номер варианта соответевует номеру зашего в журнале.

№	контейнер
1	deque
2	vector
3	list
4	deque
5	vector
6	list
7	deque
8	deque
9	vector
10	list
11	list
12	deque
13	vector
14	deque
15	vector

4. Контрольные вопросы.

1. Контейнер (сборник)

2. Объясните разницу между массивом и контейнером.

3. Какой тип библиотеки является STL (Стандартная библиотека шаблонов)?

4. Что входит в серию контейнеров?

5. Что вы знаете о классе контейнера Array и его методах?

6. Что вы знаете о классе контейнера Vector и его методах?

7. Что вы знаете о классе deque container и его методах?

8. Что вы знаете о классе контейнера списка и его методах?

9. Что вы знаете о классе контейнера Forward_list и его методах?

10. Какие библиотеки используются для использования последовательных контейнеров?

11. Что такое стандартная библиотека шаблонов?

12. Что такое контейнеры или коллекции?

13. Какие контейнеры являются последовательными?

14.Что такое array?

15. Что такое vector?

16. Что такое deque?

17. Как инициализировать вектор C ++

18. Использование массива

19. Перемещение значений по одному

20. Использование перегруженного конструктора векторного класса

Литература

1. Mo'minov.B.B, Dasturlash I(Darslik).-T."Nihol print" ok, 2021, 280 b.
2. Madraximov Sh.F., A.M.Ikramov., Babajanov M.R. C++ tili programmalash bo'yicha masalalar to'plami. Toshkent. "Universitet" nashiryoti -2014. -160 bet.
3. Nazirov Sh A., Qobulov R.V., Bobojanov M.R., Raxmanov Q.S. S va C++ tili. "Voris- nashriyot" MChJ, Toshkent 2013, 488 b.
4. Mo'minov.B.B. Dasturlash II(Darslik).-T."Nihol print" ok, 2021, 604 b.
5. Herbert Schildt, Java the complete reference ninth edition, oracle press, 2014.
6. Алгоритмы. Справочник с примерами на C, C++, Java и Python. Джордж Хайнеман, Гэри Поллис, Стэнли Селков. Москва «Альфа-Книга», 2017. – С254.
7. ASP.NET MVC Framework с примерами на C# для профессионалов. Стивен Сандерсон. Москва 2010. –С. 562.
8. Современные технологии разработки программ, взаимодействующих с базами данных. С.А. Минеев. Ю.Е. Чуманкин. Нежний Новгород, 2018.
9. C++ High Performance: Master the art of optimizing the functioning of your C++ code, 2nd Edition 2nd ed. Edition. Author: Bjorn Andrist, Ben Garney. Published 2022

Содержание.

Формат 60x84 1/16. Печ.лист 4,25.
Заказ №59. Тираж 20.
Отпечатано в «Редакционно издательском»
отделе при ТУИТ.
Ташкент ул. Амир Темур, 108.