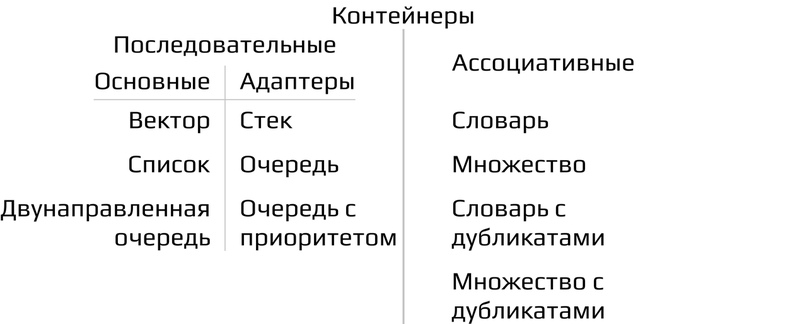
Последовательные контейнеры библиотеки STL

STL — Standard Template Library, стандартная библиотека шаблонов состоит из двух основных частей:

1. Набор контейнерных классов;
2. Набор обобщённых алгоритмов.

# 1 Контейнеры

*Контейнеры*— наборы однотипных объектов. Контейнерные классы являются шаблонными и могут хранить объекты любых типов данных. Эти объекты должны допускать копирование и присваивание. Встроенные типы этим требованиям удовлетворяют; то же самое относится к классам, если конструктор копирования или операция присваивания не объявлены в них закрытыми или защищенными. Контейнеры STL реализуют основные структуры данных, используемые при написании программ. Классификация контейнеров приведена на рисунке 1.

Рисунок 1 — Классификация контейнеров

## 1.1 Последовательные контейнеры

Последовательные контейнеры хранят конечное количество однотипных объектов в виде непрерывной последовательности. К базовым контейнерам относятся векторы (vector), списки (list) и двусторонние очереди (deque). Специализированные адаптеры реализованы на основе базовых — стеки (stack), очереди (queue) и очереди с приоритетом (priority\_queue).

Для использования контейнера необходимо подключить соответствующий заголовочный файл:

//Листинг 1  
//Объявление контейнерных классов

#include <vector>  
#include <list>  
#include "Time.h"  
/\*Some code here\*/  
vector <double> vect; //Вектор, содержащий объекты типа double  
list <Time> clock; //Список, содержащий объекты типа Time

# 2 Общие свойства контейнеров

Обратимся к таблице 1 для ознакомления с унифицированными типами, определёнными в STL.

|  |  |
| --- | --- |
| size\_type | Тип индексов, счётчиков и т.д. |
| iterator | Итератор |
| const\_iterator | Константный итератор (значение элемента изменять запрещено) |
| reverse\_iterator | Обратный итератор |
| const\_reverse\_iterator | Константный обратный итератор |
| reference | Ссылка на элемент |
| const\_reference | Константная ссылка на элемент |
| key\_type | Тип ключа в ассоциативных контейнерах |
| key\_compare | Тип функции сравнения двух ключей (в ассоц. конт.) |

Таблица 1. Унифицированные типы, определённые в STL

Обратимся к таблице 2 для ознакомления с операциями и методами, доступными для всех контейнеров.

|  |  |
| --- | --- |
| Операции равенства и неравенства | Возвращают true или false |
| Операция присваивания | Копирует один контейнер в другой |
| clear() | Удаление всех элементов |
| insert | Добавление одного элемента или диапазона |
| erase | Удаление одного элемента или диапазона |
| *size\_type* size() | Возвращает число элементов |
| *size\_type* max\_size() | Возвращает максимально допустимый размер контейнера |
| *bool* empty() | Возвращает true если контейнер пуст |
| *iterator* begin() | Возвращает итератор на начало контейнера |
| *iterator* end() | Возвращает итератор на конец контейнера |
| *reverse\_iterator* rbegin() | Возвращает обратный итератор на начало обратной последовательности |
| *reverse\_iterator* rend() | Возвращает обратный итератор на конец обратной последовательности |

## Таблица 2. Операции и методы, общие для всех контейнеров

# 3 Итераторы

Итераторы — это обобщение концепции указателей: они ссылаются на элементы контейнера. Для работы с итераторами необходимо подключить библиотеку <iterator>

Тип iterator определён для всех контейнерных классов, но реализация для разных классов разная. Основные операции итераторов:

1. Разыменовывание: если p — итератор, то \*p — значение объекта, на который он ссылается.
2. Присваивание одного итератора другому.
3. Сравнение на равенство и неравенство (== и !=)
4. Перемещение итератора по элементам контейнера с помощью инкремента (p++ и ++p)

Из-за различий реализации объявление объектов типа iterator должно сопровождаться указанием области видимости в форме имя\_шаблона::, например:

//Листинг 2  
//Объявление итераторов

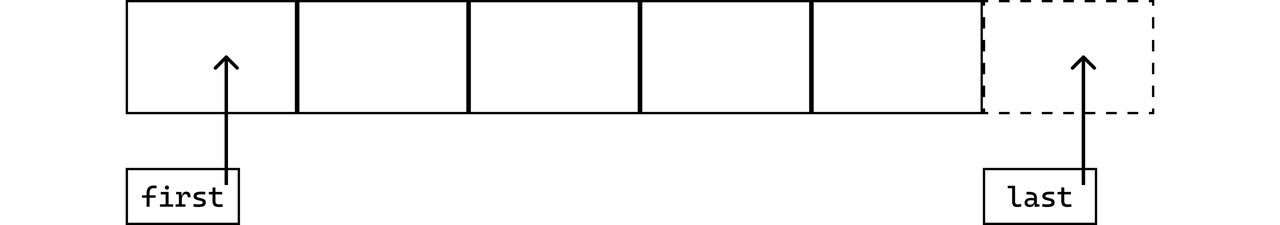
#include <vector>  
#include <list>  
#include <iterator> //подключение библиотеки итераторов

//Итератор для вектора с объектами типа double  
vector<double>::iterator vectIter;

//Реверсивный итератор для списка с объектами типа Time  
list<Time>::reverse\_iterator listIter;

Реализация циклов просмотра элементов контейнеров тоже имеет свою специфику. Так, если i — некоторый итератор, то вместо привычной формы:

for (i = 0; i < n; i++) используется:  
 for (i = first; i != last; i++), где first — значение итератора, указывающее на первый элемент в контейнере, а last — значение итератора, указывающее на воображаемый элемент за последним элементом.



Для всех контейнерных классов определены унифицированные методы begin() для получения адреса first и end() для получения адреса last.

Типы итераторов в STL:

1. Ввода – поддерживают операции равенства, разыменовывания и автоинкремента: ==, !=, ++i, i++, \*i++.  
   Могут использоваться для однопроходных алгоритмов, которые считывают данные в одном направлении.  
   Специальным случаем итератора ввода является istream\_iterator.
2. Вывода – поддерживают разыменовывание, допустимое только с левой стороны, присваивание, и инкремент:  
   ++i, i++, \*i=t, \*i++ = t.
3. Однонаправленные – поддерживают все операции итераторов ввода-вывода и позволяют без ограничений применять присваивание. Для них из i == j следует ++i == ++j, что не всегда истинно для итераторов ввода, то есть такие итераторы сохраняют свою позицию внутри контейнера, поэтому их можно использовать в алгоритмах с многократным проходом
4. Двунаправленные – поддерживают все операции однонаправленных итераторов, а также проход контейнера в обоих направлениях, т.е. декремент: --i, i–, \*i–.
5. Произвольного доступа – поддерживают все операции двунаправленных итераторов, а также операции сравнения и адресной арифметики, т.е. непосредственный доступ по индексу:  
   i+=n, a+n, i-=n, a-n, i-j, i[n], i<j, i<=j, i>=j, i>j.

## 3.1 Прямой и обратный итераторы

В STL библиотеках помимо обычных (прямых) итераторов iterator поддерживаются обратные итераторы reverse\_iterator. Обратный итератор применяется так же, как прямой. Разница состоит в реализации операторов перехода к следующему и предыдущему элементам. Для прямого итератора оператор ++ дает доступ к следующему элементу контейнера, тогда как для обратного – к предыдущему.

# 4 Использование последовательных контейнеров

К основным контейнерам относятся векторы (vector), списки (list) и двусторонние очереди (deque). Для использования контейнера необходимо подключить соответствующий заголовочный файл:

#include<vector>  
 #include<list>  
 #include<deque>

*Вектор* является аналогом обычного массива, за исключением того, что он автоматически выделяет и освобождает память по мере необходимости. Контейнер эффективно обрабатывает произвольную выборку элементов с помощью операции индексации [] или метода at. Однако вставка элемента в любую позицию, кроме конца вектора, неэффективна. Для этого потребуется сдвинуть все последующие элементы путем копирования их значений. По этой же причине неэффективным является удаление любого элемента, кроме последнего.

*Список* организует хранение объектов в виде двусвязного списка. Вставка и удаление работают эффективно для любой позиции элемента в списке. Однако список не поддерживает произвольного доступа к своим элементам: например, для выборки n-го элемента нужно последовательно выбрать предыдущие п-1 элементов.

*Двусторонняя очередь* во многом аналогична вектору, элементы хранятся в непрерывной области памяти. Но в отличие от вектора двусторонняя очередь эффективно поддерживает вставку и удаление первого элемента (так же, как и последнего).

В листинге 3 представлены способы определить объект для последовательного контейнера.

//Листинг3  
//Способы определения объекта для  
//последовательного контейнера

//1. Создать пустой контейнер  
vector<int> vec1;  
list<double> lst1;

//2. Задать контейнер заданного размера и  
//инициализировать его элементы  
//значениями по умолчанию  
vector<int> vec2(50);  
list<double> lst2(75);

//3. Задать контейнер заданного размера и  
//инициализировать его элементы заданным значением  
vector<int> vec3(133, 7);  
list<double> lst3(14, 8.8);

//4. Задать контейнер и инициализировать его элементы  
//значениями диапазона (first, last) элементов  
//другого контейнера  
int arr[8] = {0, 20, 1, 22, 8 -144, 6, -50, 100, 45};  
vector<int> vec4(arr, arr+9);  
list<int> lst4(vec4.begin()+2, vec4.end());

//5. Задать контейнер и инициализировать его элементы  
//значениями элементов другого однотипного контейнера  
list<int> lst5(10, 6);  
list<int> lst6(lst5);

Обратимся к таблице 3.

|  |  |  |
| --- | --- | --- |
| push\_back(T& key) | Добавить в конец | vector, list, deque |
| pop\_back() | Удалить из конца | vector, list, deque |
| push\_front(T& key) | Добавить в начало | list, deque |
| pop\_front() | Удалить из начала | list, deque |
| insert | Добавить в произвольное место | vector, list, deque |
| erase | Удалить из произвольного места | vector, list, deque |
| [], at | Доступ к произвольному элементу | vector, deque |
| swap | Обмен списков/векторов | vector, list |
| clear() | Очистка списков/векторов | vector, list |
| splice | Сцепка двух списков | list |

Таблица 3. Методы, которые поддерживают последовательные контейнеры

Метод insert имеет несколько реализаций:

1. iterator insert(iterator pos, const T& key); – вставка элемента key в позицию, на которую указывает pos. Возвращает итератор на новый элемент.
2. void insert(iterator pos, size\_type n, const T& key); – вставка n элементов со значением key начиная с позиции, на которую указывает pos.
3. template <class InputIter>  
   void insert(iterator pos, InputIter first, InputIter last); – Вставка элементов из диапазона first...last начиная с позиции, на которую указывает pos.

Пример использования метода insert() представлен в листинге 4.

//Листинг 4  
//Использование метода insert()

void main()  
{  
 //создать вектор из 5 элементов, проинициализировать элементы нулями  
 vector <int>v1(5,0);  
 int m[5]={1,2,3,4,5}; //массив из 5 элементов

//вставить элемент со значением 100 в начало вектора:  
 v1.insert(v1.begin(),100);

//вставить два элемента со значением 200 после первого элемента вектора:  
 v1.insert(v1.begin()+1,2,200);

//вставить элементы из массива m после третьего элемента:  
 v1.insert(v1.begin()+3,m,m+5); //m - указатель на первый, +5 т.к. пять элементов

//вставить элемент 100 в конец вектора:  
 v1.insert(v1.end(),100);

//вывести вектор на печать:  
 for(int i=0;i<v1.size();i++)  
 cout << v1[i] <<' ';  
}

//Результат работы программы:  
//100 200 200 1 2 3 4 5 0 0 0 0 0 100

Метод erase также имеет несколько реализаций:

1. iterator erase(iterator pos); – Удаляет элемент в позиции, на которую указывает итератор pos.
2. iterator erase(iterator first, iterator last); – Удаляет диапазон элементов.

Пример использования метода erase() представлен в листинге 5.

//Листинг 5  
//Использование метода erase()

int main()  
{  
 vector<int> v1;//создать пустой вектор

int m[5] = {1,2,3,4,5};  
 int n,a;

cout << "insert n: "; cin >> n;

for(int i=0; i < n; i++)  
 {  
 cin >> a;  
 //добавить в конец вектора элемент со значением а  
 v1.push\_back(a);  
 }

//вывод вектора  
 for (int i=0; i < v1.size(); i++)  
 cout << v1[i] << ' ';

cout << endl;

/\*удалить элемент из начала вектора и итератор поставить на начало вектора\*/  
 vector<int>::iterator iv=v1.erase(v1.begin());

cout << \*iv << '\n';//вывод первого элемента

//вывод вектора  
 for (int i=0; i < v1.size(); i++)  
 cout << v1[i] << ' ';  
}

## 4.1 Адаптеры контейнеров

Стек, очередь и очередь с приоритетами не являются самостоятельными классами, а реализованы на основе базовых, поэтому они называются адаптерами контейнеров.

По умолчанию прототипом является класс deque.

## 4.1.1 Стек

Заголовочный файл <stack>

Объявление stack<int> s; создаёт стек на основе двусторонней очереди. Если необходимо создать стек на основе списка, то используется stack<int, list<int>> s;

Такая реализация необходима для того, чтобы ограничить функционал класса только теми методами, которые необходимы новому классу. Стек не поддерживает произвольный доступ к элементам, а также итераторы.

Доступные методы:

1. push() – добавление в конец;
2. pop() – удаление из конца;
3. top() – значение текущего элемента стека;
4. size() – количество элементов;
5. empty() – проверяет пуст ли стек.

## 4.1.2 Очередь

Заголовочный файл <queue>

Объявление queue<int> q; создаёт очередь на основе двусторонней очереди. Возможна реализация на основе списка

Доступные методы:

1. push() – добавление в конец;
2. pop() – удаление из начала;
3. front() – значение первого элемента очереди;
4. back() – значение последнего элемента очереди;
5. size() – количество элементов;
6. empty() – проверяет пуста ли очередь.

## 4.1.3 Очередь с приоритетами

Заголовочный файл <queue>

Объявление priority\_queue<int> p\_q; создаёт очередь с приоритетами на основе вектора. Возможна реализация на основе списка.

Отличие от обычной очереди в том, что для извлечения выбирается максимальный элемент из хранимых. Поэтому после каждого изменения максимальный элемент сдвигается в начало контейнера.

Доступные методы:

1. push() – добавление в конец;
2. pop() – удаление из начала;
3. top() – значение первого элемента очереди;
4. size() – количество элементов;
5. empty() – проверяет пуста ли очередь.

Пример использования очереди с приоритетами представлен в листинге 6.

//Листинг 6  
//Использование priority\_queue

int main()  
{  
 priority\_queue<int> pq; //Очередь с приоритетами

//добавление элементов  
 pq.push(50);  
 pq.push(3000);  
 pq.push(-5);  
 pq.push(90);  
 pq.push(0);

while(!pq.empty()) //пока очередь не пуста  
 {  
 cout << pq.top() << ' '; //вывести первый элемент  
 pq.pop(); //удалить первый элемент  
 }  
}

//Результат работы:  
//3000 90 50 0 -5

# Список литературы

1. Ноткин А.М. Объектно-ориентированное программирование: ООП в C++ : учебное пособие, 2013 г. —230 с.
2. Стенли Б. Липпман. C++ для начинающих. —1195 c.
3. Павловская Т.А. C/C++. Программирование на языке высокого уровня, 2007 г. —461 с.