Контейнерные классы

Контейнерные классы — это классы, предназначенные для хранения данных, организованных определенным образом. Примерами контейнеров могут служить массивы, линейные списки или стеки. Для каждого типа контейнера определены методы для работы с его элементами, не зависящие от конкретного типа данных, которые хранятся в контейнере, поэтому один и тот же, вид контейнера можно использовать для хранения, данных различных типов. Эта возможность реализована с помощью шаблонов классов, поэтому часть библиотеки C++, в которую входят контейнерные классы, а также алгоритмы и итераторы, о которых будет рассказано в следующих разделах, называют *стандартной библиотекой "шаблонов* (STL — Standard Template library).

STL содержит контейнеры, реализующие основные структуры данных, используемые при написании программ — *векторы, двусторонние очереди, списки и их разновидности, словари и множества.* Контейнеры можно разделить на два тала: последовательные и ассоциативные.

Итератор является аналогом указателя на элемент. Он используется для просмотра контейнера в прямом или обратном направлении. Все, что требуется от итератора — уметь ссылаться на элемент контейнера и реализовывать операцию - перехода к его следующему элементу. Константные итераторы используются тогда, когда значения, соответствующих элементов контейнера не изменяются.

При помощи итераторов просматривать контейнеры не заботясь о фактических типах данных, используемых для доступа к элементам.

Для этого в каждом контейнере определено несколько методов, перечисленных ниже.

Метод	Пояснение
iterator begin(), const_iterator begin() const	Указывают на первый элемент
iterator end(), const_itepator end() const	Указывают на элемент, следующей за последним:
reverse iterator rbegin(), Const_reverse_iterator rbegin() const	Указывают на первый элемент в обратной последовательности
reverse_iterator rend(), const_reverse_iterator rend() const	Указывают на элемент, следующий за последним, в обратной последо-вательности

Во всех контейнерах определены методы, позволяющие получить сведения о размере контейнеров:

Метод	**Пояснение**
size()	Число элементов
maxsize()	Максимальный размер контейнера (порядка миллиарда элементов)
empty()	Булевская функция, показывающая, пуст ли контейнер

Последовательные контейнеры

Векторы (vector), двусторонние очереди (deque) и списки (1ist) поддерживают разные наборы операций, среди которых есть совпадающие операции. Они, могут быть реализованы с разной эффективностью:

Операция	Метод	vector	deque	list
Вставка в начало	push_front	-	+	+
Удаление из начала	pop_front	-	+	+
Вставка в конец	push_back	+	+	+
Удаление из конца	pop_back	+	+	+
Вставка в произвольное место	insert	(+)	(+)	+
Удаление из произвольного места	erase	(+)	(+)	+
Произвольный доступ к элементу	[].at	+	+	-

Знак + означает, что соответствующая операция реализуется за постоянное время, не зависящее от количества n элементов в контейнере. Знак (+) означает, что соответствующая операция реализуется за время, пропорциональное n. Для малых n время операций, обозначенных +, может превышать время операций, *обоз*наченных (+). но для большого количества элементов последние могут оказаться очень дорогими.

Итак, **вектор —** это структура, эффективно реализующая произвольный доступ к элементам, добавление, в конец и удаление из конца.

Двусторонняя очередь эффективно реализует произвольный доступ к элементам, добавление в оба конца и удаление из обоих, концов.

Список эффективно реализует вставку и удаление элементов в произвольное место, но не имеет произвольного доступа к своим элементам.

Вектор

Примеры конструкторов:

Создает вектор из **10** равных единице элементов:

vector <int> v2 (10,1);

Создается вектор, равный вектору v1:

vector <int> v4(v1);

Создается вектор из двух элементов, равных первым двум элементам v1:

vector <int> v3 (v1.begin(), v1.begin()+ 2);

Создается вектор из **10** объектов класса monstr (работает конструктор по умолчаний):

vector <monstr> m1(10);

Создается вектор из 5 объектов класса monstr с заданным именем (работает конструктор с параметров char*):

vector <monstr> m2 (5, monstr("Вася"));

В шаблоне vector определены операция ***присваивания*** и функция ***копирования***:

vector<T>& operator=(const vector<***T***> &x);
void ***assign***(size_type n, const ***T*** &value);
template «class InputIter>
void ***assign***(InputIter first, InputIter last);
Здесь через ***T*** обозначен тип элементов вектора. Вектора можно присваивать друг другу точно так же, как стандартные типы данных или строки. После присваивания размер вектора становится равным новому значению, все старые элементы удаляются.
Примеры:
vector <int> v1,v2;
Первым 10 элементам вектора v1 присваивается значение 1**:**
v1.assign(10,1)**;**
Первым 3 элементам вектора v2 присваиваются значения v1[5], v1[6], vl[7]: v2.assign(v1.begin()+5,v1.begin()+8);
Для векторов определены ***операции сравнения*** ==, !=, <, <= и =.

Пример. В файле находится произвольное количество целых чисел. Программа считывает их в вектор и выводит на экран в том же порядке.

```
#include <fstream>
#include <vector>
#include <iostream>
#include <algorithm>
using namespace std;
void main(){
  ifstream in("input.txt");
  vector<int> v;
  vector <int>::iterator i;
  int x;
  do { in>>x; v.push_back(x);
  } while (!in.eof());
  sort(v.begin(),v.end());
  for (i= v.begin(); i!=v.end();i++)
  cout<<*i<<" ";
}
```

Двусторонние очереди (deque)

Двусторонняя очередь – это последовательный контейнер, который, наряду с вектором, поддерживает произвольный доступ к элементам и обеспечивает вставку и удаление из обоих концов очереди за постоянное время. Те же операции с элементами внутри очереди занимают время, пропорциональное количеству перемещаемых элементов. Распределение памяти выполняется автоматически. Доступ к элементам очереди осуществляется за постоянное время, хотя оно и несколько больше, чем для вектора.

Примеры конструкторов (см. примеры для вектора):

deque <int> *d2* (10, 1);

deque <int> d4 (v1);

deque <int> d3 (v1.begin(), v1.begin() + 2);

deque <monstr> m1(10);

deque <monstr> m2 (5, Monstr(“Вася в очереди”));

В шаблоне deque определены операция присваивания, функция копирования, итераторы, операции сравнения, операции и функции доступа к элементам и изменения объектов, аналогичные соответствующим операциям и функциям вектора.
Вставка и удаление так же, как и для вектора, выполняются за пропорциональное количеству элементов время. Если эти операции выполняются над внутренними элементами очереди, все значения итераторов и ссылок на элементы очереди становятся недействительными. Кроме перечисленных, определены функции добавления и выборки из начала очереди:

```
void push_front(const T& value);
void pop_front();
```

При выборке элемент удаляется из очереди. Для очереди определены resize и size. К очередям можно применять алгоритмы стандартной библиотеки.

Списки (list)

Список не предоставляет произвольного доступа к своим элементам, зато вставка и удаление выполняются за постоянное время. Класс ***list*** реализован в ***STL*** в виде двусвязного списка, каждый узел которого содержит ссылки на последующий и предыдущий элементы. Поэтому операции инкремента и декремента для итераторов списка выполняются за постоянное время, а передвижение на ***n*** узлов требует времени, пропорционального ***n***.

После выполнения операций вставки и удаления значения всех итераторов и ссылок остаются действительными.

Список поддерживает конструкторы, операцию присваивания, функцию копирования, операции сравнения и итераторы.

Для занесения в начало и конец списка определены методы, аналогичные соответствующим методам очереди:

void push_front(const T& value); void pop_front();	void push_back(const T& value); void pop_back();

В общем случае для поиска элемента в списке используется функция **find**. Для удаления элемента по его значению применяется функция remove:

void remove(const T& value);

Если элементов со значением value в списке несколько, все они будут удалены. Для сортировки элементов списка используется метод sort:

void sort();

template <class Compare> void sort(Compare comp);

В первом случае список сортируется по возрастанию элементов (в соответствии с определением операции **<** для элементов), во втором — в соответствии с функциональным объектом Compare.

```
#include <iostream>
#include <list>
using namespace std;
bool comp(int &a, int &b){
return a>b;
}
int main(){
list<int> L1;
list<int>::iterator i;
int a;
setlocale(LC_ALL,"rus");
for (int i = 0;  i<10; i++) {
  a=rand() % 100;
  L1.push_back(a);
}
cout << "Исходный список:\n";
for (i = L1.begin(); i != L1.end();
++i)
cout << *i << " "; cout << endl;
```

```
L1.sort();
cout << «По возрастанию:\n";
for ( i = L1.begin(); i != L1.end(); ++i)
cout << *i << " ";
cout << endl;
L1.sort(comp);
cout << «По убыванию:\n";
for ( i = L1.begin(); i != L1.end(); ++i)
cout << *i << " ";
cout << endl;

cout << L1.front() << endl;
cout << L1.back() << endl;
cout << "Поиск числа 64:\n";
if(find(L1.begin(),L1.end(),64)!=L1.end())
cout<<*find(L1.begin(),L1.end(),64); else
cout<<"NO";
cout << endl;
}
```

Стеки (stack)

Как известно, в стеке допускаются только две операции, изменяющие его размер — добавление элемента в вершину стека и выборка из вершины. Стек можно реализовать на основе любого из рассмотренных контейнеров: вектора, двусторонней очереди или списка. Таким образом, стек является не новым типом контейнера, а вариантом имеющихся, поэтому он называется *адаптером* контейнера. Другие адаптеры (очереди и очереди с приоритетами) будут рассмотрены в следующих разделах.

В STL стек определен по умолчанию на базе двусторонней очереди:

template <class T, class Container = deque<T> >

При работе со стеком нельзя пользоваться итераторами и нельзя получить значение элемента из середины стека. Для стека, как и для всех рассмотренных выше контейнеров, определены операции сравнения.

Пример использования стека (программа вводит из файла числа и выводит их на экран в обратном порядке):

```
#include <fstream>
#include <iostream>
#include <vector>
#include <stack>
using namespace std;
int main(){
ifstream in("input.txt");
stack <int, vector<int> > s;
int x;
while ( in >>x, !in.eof())  s.push(x);
while (!s.empty()){
  x = s.top(); cout <<x << " ";    s.pop();
}
}
```

Содержимое файла input.txt:

56 34 54 0 76 23 51 11 51 11 76 88

Результат работы программы:

88 76 11 51 11 51 23 76 0 54 34 56

Очереди (queue)

Для очереди допускаются две операции**,** изменяющие ее размер - добавление элемента в конец и выборка из начала. Очередь является адаптером, который **можно** реализовать на основе двусторонней очереди или списка.

В STL очередь определена по умолчанию на базе двусторонней очереди:

template <class T, class Container = deque<T> >

Методы ***front*** и ***back*** используются для получения значений элементов, находящихся соответственно в начале и в конце очереди (при этом элементы остаются в очереди).

Пример работы с очередью (программа вводит из файла числа в очередь и выполняет выборку из нее, пока очередь не опустеет):

```
#include <fstream>
#include <iostream>
#include <deque>
#include <queue>
using namespace std;
int main(){
 ifstream in ("input.txt");
 queue <int, deque<int> > q;
 int x;
 while ( in >> x, !in.eof()) q.push(x);
 cout << "q.front(): " << q.front() << "    ";
 cout << "q.back(): " << q.back() << endl;
 while (!q.empty()){
    q.pop();
    if(q.empty()) break;
    cout << "q.front(): " << q.front() << "  ";
    cout << "q.back(): " << q.back() << endl;
 }
}
```

Содержимое файла input.txt:

56 34 54 0 76 23 51 11 51 11 76 88

Напечатайте результат работы:

Ассоциативные контейнеры

Как уже указывалось, ассоциативные контейнеры обеспечивают быстрый доступ к данным за счет того, что они, как правило, построены на основе сбалансированных деревьев поиска (стандартом регламентируется, только интерфейс контейнеров, а не их реализация).

Словарь построен на основе пар значений, первое из которых представляет собой ключ для идентификации элемента, а второе — собственно элемент. Можно сказать, что ключ ассоциирован с элементом, откуда и произошло название этих контейнеров. Например, в англо-русском словаре ключом является английское слово, а элементом — русское. Обычный массив тоже можно рассматривать как словарь, ключом в котором служит номер элемента. В словарях, описанных в STL, в качестве ключа, может использоваться значение произвольного типа. Ассоциативные контейнеры описаны в заголовочных файлах <map> и <set>.

Для хранения пары «ключ—элемент» используется шаблон pair, описанный в заголовочном файле <utility>:

```
template <class Tl, class T2> struct pair{
typedef  Tl  first_type;
typedef  T2  second_type;
Tl first;
T2 second;
pair();
pair (const T1& x, const T2& y);
template <class U, class V> pair(const pair<U, V> &p);
};
```

Шаблон ***pair*** имеет два параметра, представляющих собой типы элементов пары. Первый элемент имеет имя ***first***, второй — ***second***. Определено два конструктора: один должен получать два значения для инициализации элементов, второй (конструктор копирования) — ссылку на другую пару. Конструктора по умолчанию у пары нет, то есть при создании объекта ему требуется присвоить значение явным образом.

Пример формирования пар:

```
#include <iostream>
#include <utility>
using namespace std;
int main()  {
   pair<int, double> p1(10, 12.3), p2(p1);
   p2=make_pair(20, 12.3);
   // Эквивалентно p2=pair <int,double>(20,12.3>
   cout <<"p1: "<<p1.first<<" "<<p1.second<<endl;
   cout<<"p2: "<<p2.first<<" "<<p2.second<<endl;
   p2.first-=10;
   if (p1==p2) cout<<"p1==p2\n";
   p1.second -=1;
   if (p2 > p1) cout<<"p2 > p1\n";
}
```

Результат работы программы:

```
p1: 10 12.3
p2: 20 12.3
pl==p2
p2 > p1
```

Словари (map)

В словаре (map), в отличие от словаря с дубликатами (multimap), все ключи должны быть уникальны. Элементы в словаре хранятся в отсортированном виде, поэтому для, ключей должно быть определено отношение «меньше». Шаблон словаря содержит три параметра: тип ключа, тип элемента и тип функционально объекта, определяющего отношение «меньше»:

```
template <class Key, class T, class Compare = less<Key> >
class map{
public:
typedef  pair <const Key, T> value_type;
explicit map(const Compared &comp=Compare());
template <class  InputIter>
map(InputIter first, InputIter last, const Compare& comp=Compare ());
map(const map <Key, T, Compare>& x);
}
```

Как видно из приведенного описания (оно дано с сокращениями), тип элементов словаря value_type определяется как пара элементов типа Key и T.

Первый конструктор создает пустой словарь, используя указанный функциональный объект. ***Второй конструктор*** создает словарь и записывает в него элементы, определяемые диапазоном указанных итераторов. Время работы этого конструктора пропорционально количеству записываемых элементов, если они упорядочены, и квадрату количества элементов, если нет. ***Третий конструктор*** является конструктором копирования.

Как и для всех контейнеров, для словаря определены деструктор, операция присваивания и операции отношения.

Для доступа к элементам по ключу определена операция [ ]:

```
T& operator[](const Key & x);
```

С помощью этой операции можно не только получать значения элементов, но и добавлять в словарь новые. В качестве примера словаря рассмотрим телефонную книгу, ключом в которой служит фамилия, а элементом — номер телефона:

```
#include <fstream>
#include <iostream>
#include <string>
#include <map>
using namespace std;
typedef map <string, long, less <string>>
map_sl;
int main(){
map_sl ml;
ifstream in( "phonebook.txt");
string str;
long num;
while (in>>num, !in.eof()){ //Чтение номера
in.get();                   // Пропуск пробела
getline(in, str);           // Чтение фамилии
ml[str]=num;
cout<<str<<" "<<num<<endl;
}
ml["Petya P."]= 2134622;     // Дополнение
словаря
```

```
map_sl :: iterator i;
cout<<"ml:"<<endl;
// Вывод словаря:
for (i=ml.begin();i!=ml.end();i++)
cout<< (*i).first<<" "<<(*i).second<<endl;
i=ml.begin(); i++;
// Вывод второго элемента:
cout << "Second element: ";
cout << (*i).first<<" "<<(*i).second<<endl;
cout <<"Vasya: " << ml["Vasia"]<< endl;
return 0;
}
```

Словари с дубликатами <multimap)

Как уже упоминалось, словари с дубликатами (multimap) допускают хранение элементов с одинаковыми ключами. Поэтому для них не определена операция доступа по индексу [ ], а добавление с помощью функции insert выполняется успешно в любом случае. Функция возвращает итератор на вставленный элемент.

Элементы с одинаковыми ключами хранятся в словаре в порядке их занесения. При удалении элемента по ключу функция erase возвращает количество удаленных элементов. Функция equal_range возвращает диапазон итераторов, определяющий все вхождения элемента с заданным ключом. Функция count может вернуть значение, большее 1. В остальном словари с дубликатами аналогичны обычным словарям.

Множества (set)

Множество — это ассоциативный контейнер, содержащий только значения ключей, то есть тип value_type соответствует типу Key. Значения ключей должны быть уникальны. Шаблон множества имеет два параметра: тип ключа и тип функционального объекта, определяющего отношение «меньше»:

template <class Key, class Compare less<Key>>

Из описания, приведенного с сокращениями, видно, что интерфейс аналогичен интерфейсу словаря. Ниже приведен простой пример, в котором создается множества целых чисел:

```
#include <iostream>
#include <set>
using namespace std;
typedef set<int, less<int> > set_i;
set_i::iterator i;
int main(){
int a[4] = {4, 2, 1, 2};
set_i si;                 // Создается пустое множество
set_i s2(a, a+4);    // Множество создается копированием массива
```

```
#include <iostream>
#include <set>
using namespace std;
typedef set<int, less<int> > set_i;
set_i::iterator i;
int main(){
  int a[4] = {4, 2, 1, 2};
  set_i si;
  // Множество создается
  //копированием  массива:
  set_i s2(a, a+4);
  // Конструктор копирования :
  s3(s2);
  // Вставка элементов:
  s3.insert(10);
  s3.insert(6);
  cout << "s2: ";
  for (i=s2.begin(); i!=s2.end(); i++) cout << *i << " ";
  cout << endl;
  cout << "s3: ";
  for (i=s3.begin(); i!=s3.end(); i++)
  cout << *i << " ";
  cout << endl;
  if(s2.find(10)!=s2.end()) cout<<"YES";
                            else  cout<<"NO";
  cout<< endl;
  return 0;
}
```

Результат работы программы:

s2: 1 2 4

S3: 1 2 4 6 10

NO

Как и для словаря, элементы в множестве хранятся отсортированными. Повторяющиеся элементы в множество не заносятся.

Класс bitset

Класс bitset предназначен для работы с отдельными битами. Т. е. экземпляр этого класса представляет из себя набор переменных булевского типа. Число же битов (булевских переменных) указывается при создании экземпляра класса bitset.

С переменными типа bitset можно производить стандартные побитовые операции. Кроме того, можно получать значения отдельных битов в bitset, число установленных битов, изменять все биты на противоположные и др.

Привет

```
#include <iostream>
#include <bitset>
using namespace std;
int main(){
bitset<3> b0;
   b0[0] = 1;
   b0[1] = 0;
   b0[2] = 1;
   bitset<3> b1(string("001"));
   cout<< b1 <<"\n";
   // Побитовые операции.
   cout<<"*****\n";
   bitset<3> res = b0 & b1; // Побитовое "и".
   cout << res << "\n";
   res = b0 | b1; // Побитовое "или".
   cout << res << "\n";
   res = b0 ^ b1; // Исключающее "или".
   cout << res << "\n";
   cout<<"*****\n";
```

```
 // Число установленный битов
   cout << b0.count() << "\n";
   // Общее число элементов.
   cout << b0.size() << "\n";
   // Обращение битов на
противоположные.
   b0.flip();
   cout << b0 << "\n";
   // Сдвиг битов влево.
   b1 = b1<<1;
   cout << b1 << "\n";
   // Обнуление всех битов.
   b1 = b1.reset();
   cout << b1 << "\n";
  system("pause");
   return 0;
}
```