### Коллекции

Для использования коллекции в своем коде используйте следующую директиву:

#include <T>,

где T — название коллекции

Итак, наиболее часто используются:

* vector — коллекция элементов, сохраненных в массиве, изменяющегося по мере необходимости размера (обычно, увеличивающегося);
* list — коллекция, хранящая элементы в виде двунаправленного связанного списка;
* map — коллекция, сохраняющая пары вида <const Key, T>, т.е. каждый элемент — это пара вида <ключ, значение>, при этом однозначная (каждому ключу соответствует единственное значение), где ключ — некоторая характеризующая значение величина, для которой применима операция сравнения; пары хранятся в отсортированном виде, что позволяет осуществлять быстрый поиск по ключу, но за это, естественно, придется заплатить: придется так реализовывать вставку, чтобы условие отсортированности не нарушилось;
* set — это отсортированная коллекция одних только ключей, т.е. значений, для которых применима операция сравнения, при этом уникальных — каждый ключ может встретиться во множестве (от англ. set — множество) только один раз;
* multimap — map, в котором отсутствует условие уникальности ключа, т.е. если вы произведете поиск по ключу, то получите не единственное значение, а набор элементов с одинаковым значением ключа; для использования в коде используется #include <map>;
* multiset — коллекция с тем же отличием от set’а, что и multimap от map’а, т.е. с отсутствием условия уникальности ключа; для подключения: #include <set>.

### Строки

Любая серьезная библиотека имеет свои классы для представления строк. В STL строки представляются как в формате ASCII, так и Unicode:

string — коллекция однобайтных символов в формате ASCII;

wstring — коллекция двухбайтных символов в формате Unicode; включается командой #include <xstring>.

### Строковые потоки

strstream — используются для организации STL-строкового сохранения простых типов данных.

Разбор примеров начнем именно с этого класса.

//stl.cpp: Defines the entry point for the console application&nbsp; #include "stdafx.h" #include <iostream> #include <strstream> #include <string> using namespace std; int \_tmain (int argc, \_TCHAR\* argv []) { strstream xstr; for (int i = 0; i < 10; i++) { xstr << "Demo " << i << endl; } cout << xstr.str (); string str; str.assign (xstr.str (), xstr.pcount ()); cout << str.c\_str (); return 0; }

**Строковый поток** — это буфер, поэтому получить реальный конец можно получив счетчик: pcount(). Затем “реальная часть” потока копируется в новую строку.

### Итераторы

Очень важное понятие в реализации динамических структур данных — итератор. Неформально итератор можно определить как абстракцию, которая ведет себя как указатель, возможно, с какими-то ограничениями. Строго говоря, итератор — более общее понятие, и является объектной оберткой для указателя, поэтому указатель является итератором. Примерно его устройство может выглядеть так:

class Iterator { T\* pointer; public: T\* GetPointer () { return this - >pointer; } void SetPointer (T\* pointer) { this - >pointer = pointer; } };

Вот несколько формализованных определений итератора:

* Итераторы обеспечивают доступ к элементам коллекции
* Для каждого конкретного класса STL итераторы определяются отдельно внутри класса этой коллекции.

Существуют три типа итераторов:

* (forward) iterator — для обхода коллекции от меньшего индекса к большему;
* reverse iterator — для обхода коллекции от большего индекс к меньшему;
* random access iterator — для обхода коллекции в любом направлении.

Вот пример использования итераторов для удаления половин элементов коллекции:

#include "stdafx.h"

#include <iostream>

#include <vector>

#include <algorithm>

using namespace std;

void printInt (int number);

int \_tmain (int argc, \_TCHAR\* argv [])

{

vector<int> myVec;

vector<int>::iterator first, last;

for (long i=0; i<10; i++)

{

myVec.push\_back (i);

}

first = myVec.begin ();

last = myVec.begin () + 5;

if (last >= myVec.end ())

{ return - 1; }

myVec.erase (first, last);

for\_each (myVec.begin (), myVec.end (), printInt);

return 0;

}

void printInt (int number)

{

cout << number << endl;

}

Важно понимать, что при получении итератора на какой-то элемент коллекции и последующем изменении коллекции итератор может стать непригоден для использования.

Итерация вперед и аналогично назад происходит так:

for (iterator element = begin (); element < end (); element++)

{ t = (\*element); }

При использовании random access iterator, например, так:

for (iterator element = begin (); element < end (); element+=2)

{ t = (\*element);}

### Методы коллекций

Основными методами, присутствующими почти во всех коллекциях являются следующие:

* empty — определяет, пуста ли коллекция;
* size — возвращает размер коллекции;
* begin — возвращает прямой итератор, указывающий на начало коллекции;
* end — возвращает прямой итератор, указывающий на конец коллекции, т.е. на несуществующий элемент, идущий после последнего;
* rbegin — возвращает обратный итератор на начало коллекции;
* rend — возвращает обратный итератор на конец коллекции;
* clear — очищает коллекцию, т.е. удаляет все ее элементы;
* erase — удаляет определенные элементы из коллекции;
* capacity — возвращает вместимость коллекции, т.е. количество элементов, которое может вместить эта коллекция (фактически, сколько памяти под коллекцию выделено);

Вместимость коллекции, как было сказано в начале, меняется по мере надобности, т.е. если вся выделенная под коллекцию память уже заполнена, то при добавлении нового элемента вместимость коллекции будет увеличена, а все значения, бывшие в ней до увеличения, будут скопированы в новую область памяти — это довольно “дорогая” операция. Убедиться в том, что размер и вместимость — разные вещи, можно на следующем примере:

vector<int> vec;

cout << "Real size of array in vector: " << vec.capacity () << endl;

for (int j = 0; j < 10; j++)

{ vec.push\_back (10); }

cout << "Real size of array in vector: " << vec.capacity () << endl;

### 

### Vector

Самая часто используемая коллекция — это вектор. Очень удобно, что у этой коллекции есть такой же оператор operator [], что и у обычного массива. Такой же оператор есть и у коллекций map, deque, string и wstring.

Важно понимать, что вместимость vector'а изменяется динамически. Обычно для увеличения размера используется мультипликативный подход: выделенная под vector память увеличивается при необходимости в константное число раз, т.е. если добавление нового элемента приведет к тому, что размер массива превысит вместимость, то операционной системой для программы будет выделен новый участок памяти, например, в два раза больший, в который будут скопированы все значения из старого участка памяти и к которому будет дописано новое значение.

### Алгоритмы

Все реализованные функции можно поделить на три группы:

* Методы перебора всех элементов коллекции и их обработки: count, count\_if, find, find\_if, adjacent\_find, for\_each, mismatch, equal, search copy, copy\_backward, swap, iter\_swap, swap\_ranges, fill, fill\_n, generate, generate\_n, replace, replace\_if, transform, remove, remove\_if, remove\_copy, remove\_copy\_if, unique, unique\_copy, reverse, reverse\_copy, rotate, rotate\_copy, random\_shuffle, partition, stable\_partition
* Методы сортировки коллекции: sort, stable\_sort, partial\_sort, partial\_sort\_copy, nth\_element, binary\_search, lower\_bound, upper\_bound, equal\_range, merge, inplace\_merge, includes, set\_union, set\_intersection, set\_difference, set\_symmetric\_difference, make\_heap, push\_heap, pop\_heap, sort\_heap, min, max, min\_element, max\_element, lexographical\_compare, next\_permutation, prev\_permutation
* Методы выполнения определенных арифметических операций над членами коллекций: Accumulate, inner\_product, partial\_sum, adjacent\_difference

### Предикаты

Для многих алгоритмов STL можно задать условие, посредством которого алгоритм определит, что ему делать с тем или иным членом коллекции. Предикат — это функция, которая принимает несколько параметров и возвращает логическое значение (истина/ложь). Существует и набор стандартных предикатов.

### Потокобезопасность

Важно понимать, что STL — не потокобезопасная библиотека. Но решить эту проблему очень просто: если два потока используют одну коллекцию, просто реализуйте критическую секцию.