STL

STL (Standard Template Library, стандартная библиотека шаблонов) - набор согласованных обобщённых алгоритмов, контейнеров, средств доступа к их содержимому и различных вспомогательных функций. Является одной из самых важных составляющих языка C++. Стандартная библиотека шаблонов до включения в стандарт C++ была сторонней разработкой, в начале — фирмы HP, а затем SGI (Silicon Graphics, Inc.).

Стандарт языка не называет её «STL», так как эта библиотека стала неотъемлемой частью языка, однако многие люди до сих пор используют это название, чтобы отличать её от остальной части стандартной библиотеки (потоки ввода/вывода (iostream), подраздел Си и др.).

Библиотека содержит большое количество широко распространённых алгоритмов и структур данных. Например, в ней определены шаблонные классы векторов, списков, очередей и стеков, а так же многочисленные функции для работы с ними.

Поскольку STL состоит из шаблонных классов, её алгоритмы и структуры данных можно применять практически к любым типам данных.

Основу STL составляют пять основных компонентов:
1.Контейнер (container) - хранение набора объектов в памяти.
2.Итератор (iterator) - обеспечение средств доступа к содержи-мому контейнера.
3.Алгоритм (algorithm) - определение вычислительной процедуры.
4.Адаптер (adaptor) - адаптация компонентов для обеспечения различного интерфейса.
5.Функциональный объект (functor) - сокрытие функции в объекте для использования другими компонентами.
В дополнение к ним STL содержит один из наиболее важных классов — **string**. Этот класс определяет тип данных, позволяющих работать с символьными строками как обычно — с помощью операторов, а не функций.

Взаимодействие этих элементов обеспечивает стандартные решения очень широкого круга задач программирования.

Контейнеры — это объекты, хранящие внутри себя другие объекты. Контейнеры библиотеки STL можно разделить на четыре категории: последовательные, ассоциативные, контейнеры-адаптеры и псевдоконтейнеры.

Контейнер	Описание	Заголовок
Последовательные контейнеры		
vector	С-подобный динамический массив произвольного доступа с автоматическим изменением размера при добавлении/удалении элемента.	<vector>
list	Линейный список, элементы которого хранятся в произвольных кусках памяти, в отличие от контейнера vector, где элементы хранятся в непрерывной области памяти.	<list>
deque	Двусторонняя очередь. Контейнер похож на vector, но с возможностью быстрой вставки и удаления элементов на обоих концах.	<deque>
Ассоциативные контейнеры		
set	Множество, в которое каждый элемент может входить лишь один раз.	<set>
multiset	То же что и set, но позволяет хранить повторяющиеся элементы.	<set>

map	Хранит пары "ключ"-"значение", в которых каждому ключу соответствует лишь одно значение.	<map>
multimap	То же что и map, но каждому ключу может соответствовать несколько значений.	<map>
Контейнеры-адаптеры		
stack	Стек — контейнер, в котором добавление и удаление элементов осуществляется с одного конца.	<stack>
queue	Очередь — контейнер, с одного конца которого можно добавлять элементы, а с другого — вынимать.	<queue>
priority_queue	Очередь с приоритетом, организованная так, что самый большой элемент всегда стоит на первом месте.	<queue>
Псевдоконтейнеры		
bitset	Служит для хранения битовых масок.	<bitset>
basic_string	Контейнер, предназначенный для хранения и обработки строк. Хранит в памяти элементы подряд единым блоком, что позволяет организовать быстрый доступ ко всей последовательности.	<string>
valarray	Шаблон служит для хранения числовых массивов и оптимизирован для достижения повышенной вычислительной производительности. В некоторой степени похож на vector, но в нём отсутствует большинство стандартных для контейнеров операций.	<valarray>

Каждый контейнерный класс определяет набор функций, которые можно к нему применять. Например, класс list содержит функции для вставки, удаления и слияния элементов, а класс stack предусматривает функции для выталкивания и заталкивания элементов.

Алгоритмы

Алгоритмы применяются к контейнерам. Они позволяют манипулировать их содержимым: инициализировать, сортировать, искать и преобразовывать содержимое контейнера. Многие алгоритмы применяются к целому диапазону элементов, находящихся в контейнере.

Адаптеры

Адаптер - адаптация компонентов для обеспечения различного интерфейса. Проще говоря, адаптер превращает одну сущность в другую. Например, контейнер queue, создающий стандартную очередь, является адаптером по отношению к контейнеру deque.

Векторы

Наиболее универсальным контейнерным классом является vector, представляющий собой динамический массив, размеры которого могут меняться по мере необходимости. Несмотря на то, что вектор является динамическим массивом, для доступа к его элементам используется обычный способ индексации.

Шаблонная спецификация класса vector:

```
template <class Type, class Allocator=allocator<Type> > class vector
```

Type определяет тип данных, хранящихся в контейнере, а класс Allocator определяет механизм распределения памяти. По-умолчанию используется стандартный распределитель.

Класс vector содержит следующие конструкторы:

```
explicit vector(const Allocator &a = Allocator());
explicit vector(size_type num, const T &val = T(), const Allocator &a =
Allocator());
vector(const vector<Type, Allocator &ob);
template <class InIter> vector(InIter start, InIter end, const Allocator
&a = Allocator());
```

примечание:

explicit запрещает неявное преобразование и указывает на требования явной формы конструктора копий.

Первый конструктор создаёт пустой вектор. Второй — вектор, состоящий из num элементов., имеющих значение val, которое можно задавать по умолчанию. Третий конструктор создаёт вектор, содержащий элементы вектора ob. Четвёртый — вектор, состоящий из элементов, лежащих в диапазоне, определённом на интервале start и end.

Пример объявления векторов:

```
vector <int> iv;              //создаётся пустой вектор типа int
vector <char> cv(5);          //создаётся вектор типа char, состоящий из
                              //пяти элементов
vector <char> cv2(5, 'x');//инициализируется вектор типа char,
                          //состоящий из пяти элементов со значениями x
vector <int> iv2(iv);         //создаётся вектор типа int, который
                          //инициализируется другим целочисленным
                          //вектором
```

В классе vector определены следующие операторы сравнения:

==, <, <=, !=, >, >=

Для доступа к элементам вектора определён оператор [ ], работающий, как и в простом массиве.

В классе vector определено большое количество функций-членов и типов. Полный список можно получить из справки по используемой средой программирования стандартной библиотеке. Однако есть функции, общие для всех версий STL.

Функция-член	Описание
reference back( ); const_reference back( ) const;	Возвращает ссылку на последний элемент вектора.
const_iterator begin() const; iterator begin();	Возвращает итератор, установленный на первый элемент вектора.
void clear();	Удаляет из вектора все элементы.
bool empty() const;	Возвращает true, если вектор пуст и false в противном случае.
iterator end( ); const_iterator end( ) const;	Возвращает итератор, установленный на последний элемент вектора.
iterator erase(iterator i);	Удаляет элемент, на который ссылается итератор i. Возвращает итератор, установленный на элемент, следующий за удалённым.
iterator erase(iterator start, iterator end);	Удаляет все элементы из диапазона, заданного итераторами start и end. Возвращает итератор на элемент, следующий за последним удалённым
reference front( ); const_reference front( ) const;	Возвращает ссылку на первый элемент вектора
iterator insert(iterator i, const Type& val);	Вставляет элемент val перед элементом, на который указывает итератор i. Возвращает итератор, установленный на элемент val.
void insert(iterator i, size_type num, const Type& val);	Вставляет num копий элемента val перед элементом, на который ссылается итератор i.

template<class InputIterator> void insert(iterator i, InputIterator start, InputIterator end);	Вставляет перед элементом, на который ссылается итератор i, последовательность элементов, заданную итераторами start и end
void pop_back();	Удаляет последний элемент вектора
void push_back(const T& val);	Добавляет элемент val в конец вектора
size_type size() const;	Возвращает текущее количество элементов вектора

Рассмотрим пример, иллюстрирующий основные операции над векторами.

```
#include <vector>
#include <iostream>
#include <conio.h>
#include <ctype.h>

using namespace std;

int _tmain(int argc, _TCHAR* argv[])
{
    vector <char> v(10);   //создаём вектор из 10 элементов
    int i;

    //выводим на экран размер вектора
    cout << "Size of v = " << v.size() << '\n';

    //присваиваем элементам вектора значения
    for(i=0; i<10; i++)
            v[i] = i + 'a';
```

```
    //выводим на экран содержимое вектора
    cout << "Current contents:\n";
    for(i=0; i<10; i++)
            cout << v[i] << ' ';
    cout << "\n\n";

    cout << "Extended vector\n";
    //добавляем в конец вектора новые элементы,
    //размер вектора увеличивается автоматически
    for(i=0; i<10; i++)
            v.push_back(i+10+'a');

    //выводим на экран размер вектора
    cout << "New size = " << v.size() << '\n';

    //выводим на экран содержимое вектора
    cout << "Current contents:\n";
    for(i=0; i<v.size(); i++)
            cout << v[i] << ' ';
    cout << "\n\n";

    //изменяем содержимое вектора
    for(i=0; i<v.size(); i++)
            v[i]=toupper(v[i]);
    cout << "Modified contents:\n";
    for(i=0; i<v.size(); i++)
            cout << v[i] << ' ';

    _getch();
    return 0;

}
```

Результат выполнения программы:
Size of v = 10
Current contents:
a b c d e f g h i j

Extended vector
New size = 20
Current contents:
a b c d e f g h i j k l m n o p q r s t

Modified contents:
A B C D E F G H I J K L M N O P Q R S T

Вставка и удаление элементов вектора

Для вставки элемента в произвольное место вектора используется функция insert(). Для удаления любого элемента — erase().

```
#include <vector>
#include <iostream>
#include <conio.h>


using namespace std;

int _tmain(int argc, _TCHAR* argv[])
{
	vector <char> v(10);
	vector <char> v2;
	char str[]="<Vector>";
	int i;

	for(i=0; i<v.size(); i++)
		v[i] = i + 'a';

	for(i=0; str[i]; i++)
		v2.push_back(str[i]);

	cout << "Starting contents of v:\n";
	for (i=0; i<v.size(); i++)
		cout << v[i] << ' ';
	cout << "\n\n";
```

```
vector <char>::iterator p = v.begin();
        p+=2;

        v.insert(p, 10, 'X');

        cout << "Size of v = " << v.size() << '\n';
        cout << "Contents after insert:\n";
        for (i=0; i<v.size(); i++)
                        cout << v[i] << ' ';
        cout << "\n\n";

        p=v.begin();
        p+=2;
        v.erase(p, p+10);

        cout << "Size of v = " << v.size() << '\n';
        cout << "Contents after erase:\n";
        for (i=0; i<v.size(); i++)
                        cout << v[i] << ' ';
        cout << "\n\n";

        p=v.begin()+2;
        v.insert(p, v2.begin(), v2.end());
        cout << "Size of v = " << v.size() << '\n';
        cout << "Contents after insert:\n";
        for (i=0; i<v.size(); i++)
                        cout << v[i] << ' ';
        _getch();
        return 0;

}
```

Результат выполнения программы

```
Starting contents of v:
a b c d e f g h i j

Size of v = 20
Contents after insert:
a b X X X X X X X X X X c d e f g h i j

Size of v = 10
Contents after erase:
a b c d e f g h i j

Size of v = 18
Contents after insert:
a b < V e c t o r > c d e f g h i j
```

Вектор, содержащий объекты класса

Так как вектор является шаблонным классом, из этого следует, что его элементами могут быть не только экземпляры стандартных типов (int, double, и т.п.), но и пользовательские классы. При использовании пользовательских объектов в качестве элементов вектора необходимо помнить, что вектор содержит встроенные операторы сравнения (==, <, <=, !=, >, >=) и присваивания. Поэтому важно *не забывать переопределять данные операторы в классе*, который будет использоваться в качестве элемента вектора.

Списки

Класс list создаёт *двунаправленный линейный список*. В отличие от вектора, предоставляющего произвольный доступ к своим элементам, доступ к элементам списка может быть только последовательным. Поскольку список является двунаправленным, можно перемещаться от начала к концу списка и наоборот.
Шаблонная спецификация класса list имеет следующий вид

```
template <class Type, class Allocator = allocator<Type> > class list;
```

Шаблонный параметр Type задаёт тип данных, хранящихся в списке. Распределитель памяти задаётся классом Allocator, причём по-умолчанию используется стандартный распределитель памяти.
Класс list имеет следующий конструкторы:

```
explicit list(const Allocator &a = Allocator());
explicit list(size_type num, const Type &val = Type(), const Allocator &a
= Allocator());
list(const list<Type, Allocator &ob);
template <class InIter> list(InIter start, InIter end, const Allocator &a
= Allocator());
```

Первый конструктор создаёт пустой список. Второй — список, состоящий из num элементов., имеющих значение val, которое можно задавать по-умолчанию.

Третий конструктор создаёт список, содержащий элементы объекта ob. Четвёртый — формирует список, состоящий из элементов, лежащих в диапазоне, заданном итераторами start и end.
Как видно из конструкторов — они имеют такой же вид, как и конструкторы векторов. Такая унификация — одна из особенностей STL, благодаря которой, программист может не задумываться об особенностях конструктора определённого контейнера.
Как и в векторе, в list определены следующие операторы сравнения: ==, <, <=, !=, >, >=
В классе list определены функции-члены для работы с элементами класса. Так как данный класс содержит в себе все функции, что и vector (кроме оператора [ ] - его список не поддерживает), внизу приведена таблица функций-членов, специфичных только для списка (т.е. для работы со списком необходимо знать принципы работы с вектором).

Функция-член	Описание
void merge(list<T, Allocator &ob); template <class Comp> void merge(list<T, Allocator &ob, Comp cmpfn);	Внедряет упорядоченный список, содержащийся в объекте ob, в заданный список. В результате возникает упорядоченный список. После внедрения список, содержащийся в ob, становится пустым. Вторая функция merge получает в качестве параметра функцию, позволяющую сравнивать элементы списка между собой
void pop_front();	Удаляет первый элемент списка
void push_front();	Добавляет элемент val в начало списка

void remove (const T &val);	Удаляет из списка все элементы, значения которых равно val
void reverse();	Меняет порядок следования элементов списка на противоположный

void sort();
template <class Comp>
void sort(Comp cmpfn);
	Упорядочивает список. Вторая версия упорядочивает список, используя для сравнения элементов функцию cmpfn

void splice(iterator i, list <T, Allocator> &ob);	Вставляет в позицию, указанную итератором i, содержимое вектора ob. После выполнения операции список ob становится пустым
void splice(iterator i, list <T, Allocator> &ob, iterator el);	Элемент, на который ссылается итератор el, удаляется из списка ob и вставляется в вызывающий список в позицию, заданную итератором i
void splice(iterator i, list <T, Allocator> &ob, iterator start, iterator end);	Элементы списка ob, расположенные в диапазоне, заданном итераторами start и end, удаляются из списка ob и вставляются в вызывающий список в позицию, заданную итератором i

Для достижения гибкости и независимости от компиляторов любой объект, помещаемый в список, должен иметь конструктор по умолчанию. Кроме того, в нём должны быть определены операторы сравнения, чтобы исключить конфликт с работой этих же операторов самого контейнера.

```cpp
#include <list>
#include <iostream>
#include <conio.h>
using namespace std;
int _tmain(int argc, _TCHAR* argv[])
{
    list <int> lst; // Создаём пустой список
    int i;
    for(i=0; i<10; i++)
        lst.push_back(i);
    cout << "Size of list = " << lst.size() << '\n';
    cout << "Contents: ";
    list <int>::iterator p = lst.begin();
    while (p!=lst.end()){
        cout << *p << ' ';
        p++;
    }
    cout << "\n\n";
    //Изменяем содержимое списка
    p=lst.begin();
    while (p!=lst.end()){
        *p=*p + 100;
        p++;
    }
    cout << "Modified contents: ";
    p=lst.begin();
    while (p!=lst.end()){
        cout << *p << ' ';
        p++;
    }
    _getch();
    return 0;
}
```

Результат выполнения программы

Size of list = 10
Contents: 0 1 2 3 4 5 6 7 8 9

Modified contents: 100 101 102 103 104 105 106 107 108 109

Эта программа создаёт список целых чисел. Сначала формируется пустой список, после чего в него с помощью push_back() записываются 10 целых чисел, причём каждое записывается в конец существующего списка. После этого на экран выводится размер и содержимое этого списка. Затем каждое число увеличивается на 100 и снова выводятся на экран.
Важно обратить внимание на работу с итератором. Он намного удобнее и универсальнее указателя. При прибавлении единицы к текущему итератору мы получим *следующий в контейнере объект* независимо от типа элементов контейнера.

Функция end()

Во всех предыдущих примерах можно обратить внимание, что для "обхода" контейнера используется цикл с предусловием, а в качестве условия — неравенство итератора концу контейнера. Важное свойство функции end() - она возвращает указатель не на последний элемент контейнера, а на *следующий за ним* элемент. Таким образом, последнему элементу соответствует значение end() - 1. Это позволяет создавать очень эффективные алгоритмы обхода всех элементов контейнера, включая последний элемент. Если значение итератора становится равным значению end(), значит, все элементы контейнера пройдены. Эту особенность следует помнить всегда, чтобы избежать ошибок в использовании данной функции.

Ассоциативные контейнеры

Класс map создаёт ассоциативный контейнер, в котором каждому ключу соответствует единственное значение. Сам ключ представляет собой имя, с помощью которого можно получить требуемое значение. Если в контейнере хранится некоторое значение, доступ к нему возможен только через ключ. Таким образом, *ассоциативный контейнер хранит пары "ключ-значение"*. Преимущество таких контейнеров — в доступе к значениям по ключам. К примеру, по имени или фамилии человека (ключ) можно узнать его номер телефона (значение).

Как указывалось ранее, map может содержать только уникальные ключи, дубликаты не допускаются. Если необходимо создать ассоциативный контейнер, который может хранить дубликаты, применяется класс multimap или multiset.

Шаблонная спецификация класса map имеет следующий вид:

```
template <class Key, class Type, class Comp = less<Key>, class Allocator =
allocator< pair<const Key, Type> > class map
```

Класс Key определяет тип ключа, шаблонный параметр Type задаёт тип данных, хранящихся в ассоциативном массиве, а функтор Comp позволяет сравнивать два ключа. По-умолчанию в качестве функции Comp применяется стандартный функтор less(). Распределитель задаётся классом Allocator.

Класс map имеет следующие конструкторы

```
explicit map(const Comp &cmpfn=Comp(), const Allocator &a = Allocator());
map(const map<Key, Type, Comp, Allocator> &ob);
template <class InIter> map (InIter start, InIter end, const Comp &cmpfn =
Comp(), const Allocator &a = Allocator());
```

Первый конструктор создаёт пустой ассоциативный массив. Второй — контейнер, содержащий элементы объекта ob. Третий — создаёт массив, состоящий из элементов, лежащих в диапазоне, заданном итераторами start и end. Функция cmpfn определяет порядок следования элементов массива.
Как правило, любой объект, использующийся в качестве ключа, должен определять конструктор по-умолчанию, а так же операторы сравнения. Для каждого компилятора имеются свои требования к ключам.
В классе map определены следующие операторы сравнения:
==, <, <=, !=, >, >=

Функции-члены, определённые в этом классе перечислены в таблице ниже. Класс key_type представляет собой тип ключа, а класс value_type определяет тип pair<Key,Type>

Функция-член	Описание
iterator begin(); const_iterator begin() const;	Возвращает итератор, установленный на первый элемент контейнера
void clear();	Удаляет из ассоциативного массива все элементы
size_type count (const key_type &k) const;	Возвращает текущее количество дубликатов элемента со значением k в контейнере
bool empty() const;	Возвращает true, если контейнер пуст, и false в обратном случае
iterator end(); const_iterator end() const;	Возвращает итератор, установленный на последний элемент ассоциативного массива
iterator erase(iterator i);	Удаляет элемент, на который ссылается итератор i. Возвращает итератор, установленный на элемент, следующий за удалённым
iterator erase (iterator start, iterator end);	Удаляет все элементы из диапазона, заданного итераторами start и end. Возвращает итератор, установленный на элемент, следующий за последним удалённым
size_type erase (const key_type &k);	Удаляет из контейнера элементы, имеющие значение k
iterator find(const key_type &k); const_iterator find (const key_type &k);	Возвращает итератор, установленный на указанный ключ. Если ключ не найден, возвращает итератор, установленный на конец массива
iterator insert(iterator i, const value_type &val);	Вставляет элемент со значением val на место или сразу после элемента, на который ссылается итератор i. Возвращает итератор, установленный на этот элемент
template <class InIter> void insert(InIter start, InIter end);	Вставляет элементы из диапазона, заданного итераторами start и end

pair <iterator, bool> insert(const value_type &val);	Вставляет элемент со значением val в вызывающий ассоциативный контейнер. Возвращает итератор, ссылающийся на вставленный элемент. Элемент вставляется, только если его не было в ассоциативном массиве. В случае успеха возвращается объект класса pair <iterator, true>, иначе — pair <iterator, false>
mapped_type& operator[ ] (const key_type &i);	Возвращает ссылку на элемент, заданный итератором i. Если элемента в массиве не было, он вставляется туда.
size_type size() const;	Возвращает текущее количество элементов контейнера

Пары "ключ-значение" хранятся в ассоциативном массиве как объекты типа pair. Его шаблонная спецификация имеет следующий вид:

```
template <class Ktype, class Vtype> struct pair{
    typedef Ktype first_type;
    typedef Vtype second_type;
    Ktype first;
    Vtype second;
    //Конструкторы
    pair();
    pair(const Ktype &k, const Vtype &v);
    template <class A, class B>
    pair (const <A, B> &ob);
}
```

Значение, хранящееся в поле first, содержит ключ, а поле second хранит значение, соответствующее этому ключу.

Пару можно создать, вызвав либо конструкторы типа pair, либо функцию make_pair(), создающую объекты класса pair на основе информации о типе параметров.