**1) STL**

**Стандартная библиотека шаблонов (STL)** (англ. Standard Template Library) — набор согласованных обобщённых алгоритмов, контейнеров, средств доступа к их содержимому и различных вспомогательных функций в C++.

Стандартная библиотека шаблонов до включения в стандарт C++ была сторонней разработкой, в начале — фирмы HP, а затем SGI. 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.).

Проект под названием STLPort, основанный на SGI STL, осуществляет постоянное обновление STL, iostream и строковых классов. Некоторые другие проекты также занимаются разработкой частных применений стандартной библиотеки для различных конструкторских задач. Каждый производитель компиляторов C++ обязательно поставляет какую-либо реализацию этой библиотеки, так как она является очень важной частью стандарта и широко используется.

Архитектура STL была разработана Александром Степановым и Менг Ли.

Основные компоненты, составляющие внутреннюю структуру STL.

1. Контейнер - блок для хранения данных, управления данными, размещение данных. Иными словами блок представляет собой объект, который используется для хранения других объектов.
2. Алгоритм - специальная функция для работы с данными содержащимися в контейнере.
3. Итератор - специальный внутренний указатель позволяющий алгоритмам перемещаться по данным конкретного контейнера.
4. Функторы - это механизм для инкапсуляции функций в конкретном объекте с целью использования его другими компонентами.

**2) Итераторы**

Для доступа к элементам некоторого множества элементов алгоритмы stl используют специальные объекты, называемые *итераторами*. В контейнерных типах stl они доступны через методы класса (например, begin() в шаблоне класса vector). Функциональные возможности указателей и итераторов близки, так что обычный указатель тоже может использоваться как итератор.

Вот как примерно может выглядеть внутреннее устройство итератора:

**class Iterator {**

**T\* pointer;**

**public:**

**T\* GetPointer (){**

**return this - >pointer;}**

**void SetPointer (T\* pointer){**

**this - >pointer = pointer;}**

**};**

Итераторы в основном используются для операций с элементами контейнеров: сортировка, поиск, копирование и т.д. Для создания итератора необходимо написать имя контейнера <тип данных> :: iterator и имя итератора.

Например,

**vector <float>::iterator begin;**

**string::iterator end,cur;**

Строки это тоже контейнеры. Теперь что мы можем делать с итераторами.

Мы можем получить элемент, на который они ссылаются:

**cout<<\*cur<<endl;**

Мы также можем перейти как на один элемент вперед так и на несколько:

**cur++;**

**cur +=10;**

Заполнение вектора данными, и вывод этих данных на экран при помощи итератора:

|  |
| --- |
| #include <iostream>  #include <iterator>  #include <vector>  using namespace std;  int main(){  vector <int> test;  vector<int>::iterator cur;  test.push\_back(10);  test.push\_back(20);    for (cur = test.begin();cur<test.end();cur++)  cout << \*cur<<endl;  } |

Есть несколько разных типов итераторов:

|  |  |
| --- | --- |
| **Итератор** | **Описание** |
| input\_iterator (для чтения) | Читают значения с движением вперед. Могут быть инкрементированы, сравнены и разыменованы. |
| output\_iterator (для записи) | Пишут значения с движением вперед. Могут быть инкрементированы и разыменованы. |
| forward\_iterator (однонаправленные) | Читают или пишут значения с движением вперед. Комбинируют функциональность предыдущих двух типов с возможностью сохранять значение итератора. |
| bidirectional\_iterator (двунаправленные) | Читают и пишут значения с движением вперед или назад. Похожи на однонаправленные, но их также можно инкрементировать и декрементировать. |
| random\_iterator (с произвольным доступом) | Читают и пишут значения с произвольным доступом. Самые мощные итераторы, сочетающие функциональность двунаправленных итераторов и возможность выполнения арифметики указателей и сравнений указателей. |
| reverse\_iterator (обратные) | Или итераторы с произвольным доступом, или двунаправленные, движущиеся в обратном направлении. |

Каждый класс контейнеров связан с типом итератора, и каждый из алгоритмов STL использует определенный тип итератора.

#### 3) Итераторы ввода

Итераторы ввода (input iterator) стоят в самом низу иерархии итераторов. Это наиболее простые из всех итераторов STL, и доступны они только для чтения.

|  |
| --- |
| #include <iostream>  #include <algorithm>  #pragma warning (disable: 4550)  using namespace std;  void printValue(int num)  {  cout << num << "\n";  }  void main()  {  int init[] = {1, 2, 3, 4, 5};  for\_each(init, init + 5, printValue);  } |

Программа чрезвычайно проста. Есть массив init с пятью числами. Вызывается алгоритм for\_each, ему в качестве входных итераторов передаются указатели на начало массива и на адрес, следующий за концом массива, т. е. на значение "за пределами". Третьим параметром является указатель на функцию printValue, которая печатает элементы массива.

#### 4) Итераторы вывода

Если итератор ввода предназначен для чтения данных, то итератор вывода (output iterator) служит для ссылки на область памяти, куда выводятся данные. Итераторы вывода можно встретить повсюду, где происходит хоть какая-то обработка информации средствами STL

|  |
| --- |
| #include <iostream>  #include <algorithm>  #include <vector>  #include <iterator>  using namespace std;  void printValue(int num)  {  cout << num << "\n";  }  void main()  {  int init1[] = {1, 2, 3, 4, 5};  int init2[] = {6, 7, 8, 9, 10};  vector<int> v(10);  merge(init1, init1 + 5, init2, init2 + 5, v.begin());  copy(v.begin(), v.end(), ostream\_iterator<int>(cout, "\n"));  } |

 В приведенном нами примере создаются и инициализируются два массива - init1 и init2. Далее их значения соединяются вместе алгоритмом merge и записываются в вектор. А для проверки полученного результата мы пересылаем данные из вектора в поток вывода, для чего вызываем алгоритм копирования copy и специальный итератор потока вывода ostream\_iterator. Он перешлет данные в поток cout, разделив каждое пересылаемое значение символом окончания строки. Для шаблонного класса ostream\_iterator требуется указать тип выводимых значений. В нашем случае это int.

#### 5) Однонаправленные итераторы

Если соединить итераторы ввода и вывода, то получится однонаправленный итератор (forward iterator), который может перемещаться по цепочке объектов в одном направлении, за что и получил такое название.

Чтобы убедиться в правильности работы всех операторов однонаправленных итераторов, составим программу, заменяющую в исходном массиве все единицы на нули и наоборот, т. е. произведем инверсию

|  |
| --- |
| #include <iostream>  #include <algorithm>  #include <vector>  #include <iterator>  using namespace std;  void printValue(int num)  {  cout << num << "\n";  }  void main()  {  int init1[] = {1, 2, 3, 4, 5};  replace(init1, init1 + 5, 0, 2);  replace(init1, init1 + 5, 1, 0);  replace(init1, init1 + 5, 2, 1);  copy(init1, init1 + 5, ostream\_iterator<int>(cout, "\n"));  } |

Как видно из примера, алгоритм replace умело используя однонаправленные итераторы, читает и заменяет значения, а также перемещается от одного к другому.

#### 6) Двунаправленные итераторы

Двунаправленный итератор (bidirectional iterator) аналогичен однонаправленному итератору. В отличие от последнего двунаправленный итератор может перемещаться не только из начала в конец цепочки объектов, но и наоборот.

На двунаправленных алгоритмах базируются различные алгоритмы, например reverse. Этот алгоритм меняет местами все объекты в цепочке. Следующий пример был бы невозможен без двунаправленных итераторов:

|  |
| --- |
| #include <iostream>  #include <algorithm>  #include <vector>  #include <iterator>  using namespace std;  void printValue(int num)  {  cout << num << "\n";  }  void main()  {  int init[] = {1, 2, 3, 4, 5};  reverse(init, init + 5);  copy(init, init + 5, ostream\_iterator<int>(cout, "\n"));  } |

#### 7) Итераторы произвольного доступа

Итераторы произвольного доступа - самые "умелые" из основных итераторов. Они не только реализуют все функции, свойственные итераторам более низкого уровня, но и обладают большими возможностями. Например получить элемент из середины вектора.

|  |
| --- |
| #include <algorithm>  #include <iostream>  #include <vector>  #include <iterator>  using namespace std;  int main(void)  {  const int init[] = {1, 2, 3, 4, 5};  vector<int> v(5);  typedef vector<int>::iterator vectItr;  vectItr itr ;  copy(init, init + 5, itr = v.begin());  cout << \*( itr + 4 ) << endl;  cout << \*( itr += 3 ) << endl;  cout << \*( itr -= 1) << endl;  cout << \*( itr = itr - 1) << endl;  } |

Как видно из примера, операции с итераторами произвольного доступа реализованы так, чтобы программист не чувствовал разницы между использованием обычных указателей и итераторов.

#### 8) Реверсивные итераторы

Некоторые классы-контейнеры спроектированы так, что по хранимым в них элементам данных можно перемещаться в заданном направлении. В одних контейнерах это направление от первого элемента к последнему, а в других - от элемента с самым большим значением к элементу, имеющему наименьшее значение. Однако существует специальный вид итераторов, называемых реверсивными. Такие итераторы работают "с точностью до наоборот": т. е. если в контейнере итератор ссылается на первый элемент данных, то реверсивный итератор ссылается на последний.

|  |
| --- |
| #include <algorithm>  #include <iostream>  #include <vector>  #include <iterator>  using namespace std;  void main()  {  const int init[] = {1, 2, 3, 4, 5};  vector<int> v(5);  copy(init, init + 5, v.begin());  copy(v.begin(), v.end(), ostream\_iterator<int>(cout, " "));  copy(v.rbegin(), v.rend(), ostream\_iterator<int>(cout, " "));  } |

Данный пример использует нормальный итератор для вывода значений от 1 до 5 и реверсивный итератор для вывода этих же значений, но в обратном порядке