Тест с отчетом 3

Что будет выведено на экран в результате компиляции и выполнения следующего кода?

```
#include <iostream>
 using namespace std;

 class A {
    int i;
public:
    A() { cout << "in A::A()" << endl; i = 0; }
    A operator = (A a) { cout << "in A::operator=(A)" << endl; i = a.i;  }
};


int main() {
    A a;
    A b = a;
    return 0;
}
```

При A b = a; вызывается автоматически сгенерированный конструктор копирования, а не оператор присваивания, поэтому оператор присваивания ничего вывести не может, т.к. никогда не вызывается.

Варианты ответа:

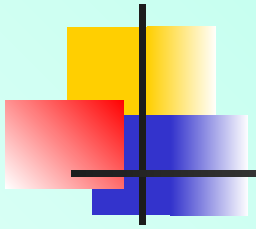
(1) - возникнет ошибка времени выполнения

(2) - in A::A()
in A::operator=(A)

(3) - in A::operator=(A)

(4) - in A::A()

(5) - возникнет ошибка компиляции



STL – основные понятия

В основе библиотеки STL лежат шесть понятий:

- контейнеры,
- итераторы,
- алгоритмы,
- объекты_функции,
- адаптеры,
- распределители (памяти).

STL – основные понятия - пояснения

В *контейнерах* хранятся объекты.

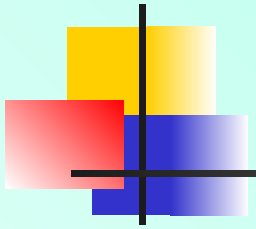
Итератор – это не зависящая от структуры данных абстракция, позволяющая получать доступ к элементам контейнера и обходить диапазоны элементов.

В *алгоритмах* итераторы применяются для обобщенного манипулирования диапазонами элементов, ничего не зная о типах самих элементов и структурах данных, в которых они размещаются.

С помощью *объектов_функций* определяются обобщенные операции, применяемые к элементам, которыми манипулируют алгоритмы.

Адаптеры позволяют согласовать между собой несопоставимые типы путем изменения их интерфейсов. Различают ***адаптеры класса*** (они адаптируют сами типы) и ***адаптеры экземпляра*** (они адаптируют экземпляры типов).

Распределители абстрагируют операции распределения памяти и конструирования объектов, выполняемые контейнерами.



STL – контейнеры (Sgi)

Согласно реализации STL фирмы Silicon Graphics определено следующее множество контейнерных классов:

Последовательные контейнеры:
- **vector** – вектор
- **deque** – дека
- **list** – список
- **slist** – односвязный список
- bit_vector – vector<bool>

Контейнерные адапторы:
- **stack** - стек
- **queue** – очередь
- priority_queue – очередь с порядком

Упаковка строк (string package):
- char_traits – трактовка символов
- basic_string – базисная строка

Ассоциативные контейнеры:
- **set** – множество
- **map** – отображение (словарь)
- **multiset** – мультимножество
- **multimap** – мультиотображение
- hash_set – хешированное множество
- hash_map – хешированное отображение (словарь)
- hash_multiset – хешированное мультимножество
- hash_multimap – хешированное мультиотображение
- hash – хеш-функция

Прочие :
- rope – специальная строка
- bit_set – множество битов

STL – итераторы

Модель итератора построена по образцу указателей в C/C++.

В общем случае к итератору можно применять операции инкремента, разыменования и сравнения. К некоторым итераторам применимы также операции декремента и указательная арифметика. Категория итератора определяется набором осмысленных операций, которые можно к нему применить.

В настоящее время в стандартной библиотеке определено пять категорий итераторов:

- итератор ввода (input – InIter),
- итератор вывода (output – OutIter),
- однонаправленный итератор (forward – ForIter),
- двунаправленный итератор (bidirectional – BiIter),
- итератор с произвольным доступом (random access – RandIter).

STL – алгоритмы

В STL алгоритмы – это обобщенные шаблоны функций, которые применяются к диапазонам, определяемым итераторами, и выполняют некоторые операции над самими диапазонами или над хранящимися в них элементами.

Например, алгоритм std::distance() не разыменовывает переданные ему итераторы, а просто вычисляет число элементов в представленном с их помощью диапазоне.

Напротив, алгоритм std::transform() присваивает элементам из одного диапазона преобразованные значения элементов из другого диапазона.

```
std::vector<int> v1 = . . . ;
std::vector<int> v2(v1.size());
std::transform(v1.begin(), v1.end(), v2.begin(), ::abs);
```

STL – объекты_функции

Любой класс, перегружающий оператор вызова функции (то есть operator()), является *классом функтора*. Объекты, созданные на основе таких классов, называются *объектами функций,* или *функторами*. Как правило, в STL объекты функций могут свободно заменяться «обычными» функциями, поэтому под термином «объекты функций» часто объединяются как функции C++, так и функторы.

Можно сказать, что объектом_функцией, или **функтором** называется объект, который можно вызывать. Объекты_функции используются прежде всего в сочетании с алгоритмами, где встречаются в виде предикатов или функций. Предикат принимает один или несколько аргументов и возвращает булевское значение, позволяя тем самым управлять выбором элементом из диапазона.

Функция принимает нуль или более аргументов и возвращает значение, которое можно использовать для трансформации или присваивания элементам.

STL – контейнерные адаптеры

Помимо основных контейнерных классов стандартная библиотека C++ содержит специальные **контейнерные адаптеры**, предназначенные для особых целей. В их реализации применяются основные контейнерные классы.

Ниже перечислены стандартные контейнерные адаптеры, определенные в библиотеке.

Стеки – контейнеры, элементы которых обрабатываются по принципу LIFO (последним прибыл, первым обслужен).

Очереди – контейнеры, элементы которых обрабатываются по принципу FIFO (первым прибыл, первым обслужен). Иначе говоря, очередь представляет собой обычный буфер.

Приоритетные очереди - контейнеры, элементам которых назначаются приоритеты. Приоритет определяется на основании критерия сортировки, переданного программистом (по умолчанию используется оператор <). В сущности, приоритетная очередь представляет собой буфер, следующий элемент которого всегда обладает максимальным приоритетом в очереди. Если максимальный приоритет назначен сразу нескольким элементам, порядок следования элементов не определен.

STL – итераторные адаптеры

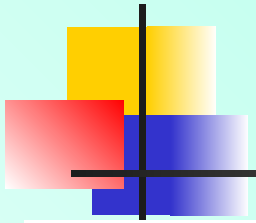
Стандартная библиотека C++ содержит несколько готовых специализированных итераторов, называемых **итераторными адаптерами**. Итераторные адаптеры являются не обычными вспомогательными классами; они наделяют саму концепцию итераторов рядом новых возможностей.

Выделяют три разновидности итераторных адаптеров:

- **итераторы вставки;**
  - **back_inserter** (контейнер) – элементы присоединяются с конца в прежнем порядке с использованием функции **push_back()**
  - **front_inserter** (контейнер) – элементы вставляются в начало в обратном порядке с использованием функции **push_front()**
  - **inserter** (контейнер, позиция) – элементы вставляются в заданной позиции в прежнем порядке с использованием функции **insert()**
- **потоковые итераторы;**
  - **istream_iterator** и **ostream_iterator**
- **обратные итераторы.**

Эти итераторы работают в противоположном направлении: вызов оператора **++ на внутреннем уровне преобразуется в вызов оператора --, и наоборот.**

Все контейнеры поддерживают создание обратных итераторов функциями rbegin() и rend().



STL – распределители (памяти)

Концепция распределителя описывает требования к типам, предоставляющим сервисы по управлению памятью. При этом подразумевается, что в типе определены некоторые члены, а также методы для выделения и освобождения неформатированной памяти и конструирования (по месту) и уничтожения объектов того типа, который задан в параметре распределителя.

По существу распределитель абстрагирует детали работы с памятью, вынося их из контейнера (и других компонентов, манипулирующих памятью) в отдельный четко определенный интерфейс, за счет чего компоненты можно специализировать такими сервисами управления памятью, которые необходимы для решения конкретной задачи.

Использование контейнеров

По сравнению с массивами контейнеры обладают большей гибкостью и функциональностью.

Главное – это освобождение программиста от проблем захвата и корректного освобождения памяти.

Они динамически увеличивают (а иногда и уменьшают) свои размеры, самостоятельно управляют памятью, следят за количеством хранящихся объектов, ограничивают алгоритмическую сложность поддерживаемых операций и обладают массой других достоинств.

Использование контейнеров STL сразу открывает широкие возможности по использованию готовых алгоритмов и функций STL.

Популярность контейнеров STL легко объяснима — просто они превосходят своих конкурентов, будь то контейнеры из других библиотек или самостоятельные реализации.

Классификации контейнеров по способу распределения памяти

В *блоковых контейнерах* (также называемых ***контейнерами со смежной памятью***) элементы хранятся в одном или нескольких динамически выделяемых блоках памяти, по несколько элементов в каждом блоке. При вставке нового или удалении существующего элемента другие элементы того же блока сдвигаются вверх или вниз, освобождая место для нового элемента или заполняя место, ранее занимаемое удаленным элементом. Подобные перемещения влияют как на скорость работы, так и на безопасность. К числу стандартных блоковых контейнеров относятся **vector**, **string** и **deque**. Нестандартный контейнер **rope** также является блоковым.

В *узловых контейнерах* каждый динамически выделенный фрагмент содержит ровно один элемент. Операции удаления и вставки выполняются только с указателями на узлы, не затрагивая содержимого самих узлов, и потому обходятся без перемещений данных в памяти. К этой категории относятся контейнеры связанных списков (такие как **list** и **slist**), а также все стандартные ассоциативные контейнеры, обычно реализуемые в форме сбалансированных деревьев. Реализация нестандартных хэшированных контейнеров тоже построена на узловом принципе.

Контейнер vector

Это простейший последовательный контейнер. Его прототип:

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

Здесь Type – тип хранящихся в контейнере данных, Allocator – распределитель памяти, по умолчанию – стандартного типа **allocator.**

Конструкторы:

vector(); – конструктор, создающий пустой вектор

vector(size_type **n**); – конструктор, создающий вектор из **n** элементов

vector(size_type **n**, const T& **t**); – конструктор, создающий вектор из **n** элементов, содержащих значение **t**

vector(const vector&); – конструктор копирования

template <**class** InputIterator> **vector** (InIterBegin, InIterEnd); – конструктор, создающий вектор по

Контейнер vector – члены функции

assign	Стирает вектор и копирует специфицированные элементы в пустой вектор.
at	Возвращает ссылку на элемент в специфицированной позиции вектора.
back	Возвращает ссылку на последний элемент вектора.
begin	Возвращает итератор произвольного доступа на первый элемент контейнера.
capacity	Возвращает объем памяти, захваченной под вектор.
clear	Стирает элементы вектора.
empty	Проверяет, пуст ли вектор.
end	Возвращает итератор произвольного доступа на точку за концом вектора.
erase	Удаляет элемент или группу элементов по специфицированной позиции.
front	Возвращает ссылку на первый элемент вектора.
get_allocator	Возвращает объект класса **allocator,** используемый вектором.
insert	Вставляет один или несколько элементов в заданную позицию вектора.
max_size	Возвращает максимальный размер вектора.
pop_back	Удаляет последний элемент вектора.
push_back	Добавляет в конец вектора новый элемент.
rbegin	Возвращает итератор на первый элемент в инверсном векторе.
rend	Возвращает итератор на конец инверсного вектора.
reserve	Резервирует минимальный размер памяти для объекта вектора.
resize	Устанавливает новый размер вектора.
size	Возвращает число элементов вектора.
swap	Меняет местами содержимое двух векторов.

Контейнер vector – переопределенные типы

allocator_type	Тип, который представляет класс **allocator** для объекта вектор.
const_iterator	Тип, который обеспечивает итератор произвольного доступа, который позволяет считывать **const**-элементы в векторе.
const_pointer	Тип, который обеспечивает указатель на **const**-элементы в векторе.
const_reference	Тип, обеспечивающий ссылку на **const-**элементы, сохраненные в векторе для чтения и исполнения **const**-операций.
const_reverse_ iterator	Тип, обеспечивающий итерирование вектора в обратном порядке.
difference_type	Тип, обеспечивающий расстояние между адресами двух элементов вектора.
iterator	Тип, обеспечивающий итератор произвольного доступа для чтения и модификации любого элемента вектора.
pointer	Тип, обеспечивающий указатель на элемент вектора.
reference	Тип, обеспечивающий ссылку на сохраненный в векторе элемент.
reverse_iterator	Тип, обеспечивающий произвольное итерирование в инверсном векторе.
size_type	Тип, определяющий число элементов в векторе.
value_type	Тип, определяющий тип сохраненных в векторе данных.

Контейнер vector – пример

```
#include <vector>
#include <iostream>
using namespace std;
int main( )
{  vector <int> vi;
   vector <int>::iterator It;
   int buf=0;                                          // Добавление элементов
   while(cout <<"Input: ", cin >> buf, buf!=0)     vi.push_back(buf);
   cout << "Vector size: "<<vi.size() <<"\n";
   cout << "Vector value:";
   for (int i=0; i < static_cast<int>(vi.size()); cout <<vi[i++]<< " ");
   cout << endl;
   It = vi.begin();
   while ( It != vi.end( ))
      cout << *It++ << " ";
   cout << endl;
   vi.erase(vi.begin(), vi.end());
   cout << "Vector size: "<<vi.size() <<"\n";
}
```

```
Input: 1
Input: 2
Input: 3
Input: 0
Vector size: 3
Vector value:1 2 3
1 2 3
Vector size: 0
```

Контейнер vector – пример «ручное управление» памятью

```
#include <iostream>
#include <vector>
using namespace std ;
typedef vector<int> INTVECTOR;
int main()
{    INTVECTOR V;
    cout << "Initial:\n0) size is: " << V.size() << endl;
    cout << "0) maximum size is: " << V.max_size() << endl;
    cout << "0) capacity is: " << V.capacity() << endl;
    V.push_back(54);      cout << endl;
    cout << "After adding:\n1) size is: " << V.size() << endl;
    cout << "1) capacity is: " << V.capacity() << endl;
    V.reserve(10);       cout << endl;
    cout << "After reserved:\n2) size is: " << V.size() << endl;
    cout << "2) capacity is: " << V.capacity() << endl;
    V.resize(20);    cout << endl;
    cout << "After resized:\n3) size is: " << V.size() << endl;
    cout << "3) capacity is: " << V.capacity() << endl;
}
```

```
Initial:
0) size is: 0
0) maximum size is: 1073741823
0) capacity is: 0

After adding:
1) size is: 1
1) capacity is: 1

After reserved:
2) size is: 1
2) capacity is: 10

After resized:
3) size is: 20
3) capacity is: 20
```

Контейнер deque (двусторонняя очередь)

Это еще один последовательный контейнер. Его прототип:

template < class Type**, class** Allocator **= allocator<**Type**> > class deque**

Здесь Type – тип хранящихся в контейнере данных, Allocator – распределитель памяти, по умолчанию – стандартного типа **allocator.**

Конструкторы:

deque(); – конструктор, создающий пустую очередь

deque(size_type **n**); – конструктор, создающий очередь из **n** элементов

deque(size_type **n**, const T& **t**); – конструктор, создающий очередь из **n** элементов, содержащих значение **t**

deque(const deque&); – конструктор копирования

template <**class** InputIterator> **deque**(InIterBegin, InIterEnd); – конструктор, создающий очередь по

Контейнер deque – члены класса

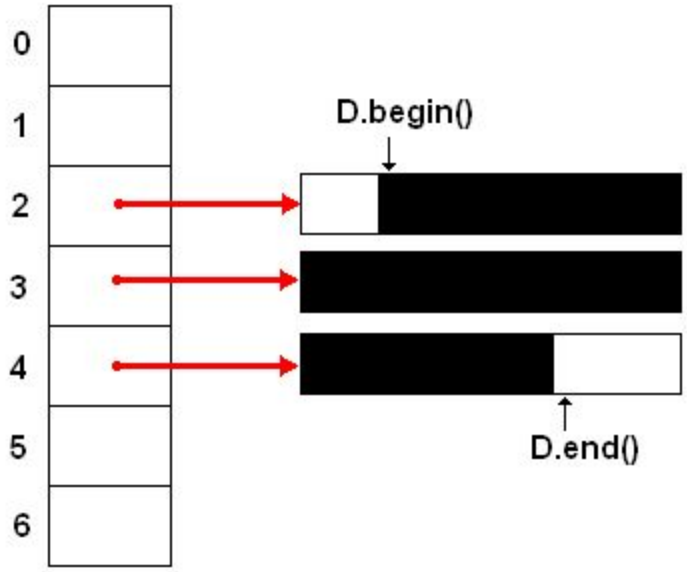
Все переопределяемые типы те же, что и у вектора:

allocator_type, const_iterator, const_pointer, const_reference, const_reverse_iterator, difference_type, iterator, pointer, reference, reverse_iterator, size_type, value_type

Члены-функции класса **deque** те же, что и для вектора, за исключением:

capacity	Отсутствует
reserve	Отсутствует
pop_front	Удаляет первый элемент очереди.
push_front	Добавляет в начало очереди новый элемент.

Контейнер deque - как она устроена?



Кажется, что поскольку предполагается вставка элементов с обоих концов очереди, то выгодно использовать в качестве основы просто список.

Но в STL используется другая схема реализации очереди, позволяющая обеспечить доступ к ее элементам за постоянное время.

Элементы двусторонней очереди размещаются в блоках фиксированного размера и хранится массив указателей на эти блоки. Черным цветом обозначены существующие элементы очереди, белым – захваченная память. Поскольку все блоки имеют одинаковый размер, то мы можем обеспечить вычисление позиции требуемого блока без перебора элементов. Фактически, для деки мы можем использовать выражение D[i] как для массива или вектора, равно как выражение *(D.begin()+i)

Контейнер deque – пример

```
#include <deque>
#include <iostream>
using namespace std;
int main( )
{ deque <int> vi;
   deque <int>::iterator It;
   int buf=0;                                          // Добавление элементов
   while(cout <<"Input: ", cin >> buf, buf!=0)     vi.push_back(buf);
   cout << "Deque size: "<<vi.size() <<"\n";
   cout << "Deque value:";
   for (int i=0; i < static_cast<int>(vi.size()); cout <<vi[i++]<< " ");
   cout << endl;
   It = vi.begin();
   while ( It != vi.end( ))
      cout << *It++ << " ";
   cout << endl;
   vi.erase(vi.begin(), vi.end());
   cout << "Deque size: "<<vi.size() <<"\n";
}
```

```
Input: 1
Input: 2
Input: 3
Input: 0
Deque size: 3
Deque value:1 2 3
1 2 3
Deque size: 0
```

Контейнер list

Это еще один последовательный контейнер. Его прототип:

template < class Type**, class** Allocator **= allocator<**Type**> > class list**

Здесь Type – тип хранящихся в контейнере данных, Allocator – распределитель памяти, по умолчанию – стандартного типа **allocator.**

Конструкторы:

list(); – конструктор, создающий пустую очередь

list(size_type **n**); – конструктор, создающий очередь из **n** элементов

list(size_type **n**, const T& **t**); – конструктор, создающий очередь из **n** элементов, содержащих значение **t**

list(const list&); – конструктор копирования

template <**class** InputIterator> **list**(InIterBegin, InIterEnd); – конструктор, создающий очередь по диапазону

Контейнер list – члены класса

Все переопределяемые типы те же, что и у вектора:

allocator_type, const_iterator, const_pointer, const_reference, const_reverse_iterator, difference_type, iterator, pointer, reference, reverse_iterator, size_type, value_type

Члены-функции класса **list** по сравнению с вектором:

at	Отсутствует
capacity	Отсутствует
reserve	Отсутствует
pop_front	Удаляет первый элемент списка.
push_front	Добавляет в начало списка новый элемент.
merge	Объединяет списки
remove	Удалет из списка все элементы, равные значению аргумента
remove_if	Удаляет из списка элементы, удовлетворяющие заданному предикату
reverse	Изменяет порядок следования элементов списка на обратный
sort	Сортирует элементы списка
splice	Удаляет элементы из списка-аргумента и встраивает их в другой
unique	Удаляет все последовательно расположенные одинаковые элементы

Контейнер list – пример

```
#include <list>
#include <iostream>
using namespace std;
int main( )
{  list <int> vi;
   list <int>::iterator It;
   int buf=0;                                          // Добавление элементов
   while(cout <<"Input: ", cin >> buf, buf!=0)     vi.push_back(buf);
   cout << "List size: "<<vi.size() <<"\n";
   cout << "List value:";
   It = vi.begin();
   while ( It != vi.end( ))
      cout << *It++ << " ";
   cout << endl;
   vi.erase(vi.begin(), vi.end());
   cout << "List size: "<<vi.size() <<"\n";
}
```

```
Input: 1
Input: 2
Input: 3
Input: 0
List size: 3
List value:1 2 3
List size: 0
```

Контейнер list – функции *sort* и *unique*

Алгоритм **sort** из STL не работает со списками. Вместо него включена функция-член класса **sort**.

Функция **unique** удаляет повторяющиеся последовательные элементы.

```
Initial:
123 123 123 123 123 100 123 123
After unique:
123 100 123
After sort:
100 123 123
```

```
#include <iostream>
using namespace std;
#include <list>
void out(char *s, const list<int> &L)
{ cout << s;
   copy(L.begin(), L.end(), ostream_iterator <int> (cout, " "));  cout << endl;
}

void main()
{  list<int> L(5,123);
  L.push_back(100); L.push_back(123); L.push_back(123); out ("Initial:\n",L);
  L.unique();  out ("After unique:\n",L);
  L.sort(); out ("After sort:\n",L);
}
```

Контейнер list – функции *сцепки*

Функция **splice** перемещает один или более элементов из одного списка в другой не перераспределяя память.

1) `void splice(iterator position, list<T, Alloc>& x);`

Если **i** является допустимым итератором для списка L, то следующая операция вставляет содержимое списка M перед i в L, оставляя M пустым:

```
L.splice(i,M);
```

Внимание! **L и M должны обязательно быть разными!**

2) `void splice(iterator position,list<T, Alloc>& x, iterator j);`

Если **i** является допустимым итератором для списка L, а j – для M, то следующая операция удаляет элемент, на который ссылается **j**, и вставляет его перед **i**. Возможно, что L и M – это один и тот же список.

```
L.splice(i,M,j);
```

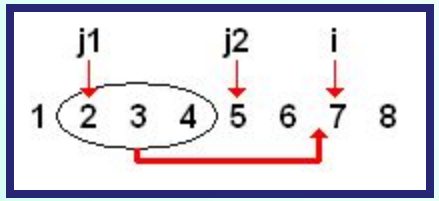
3) `void splice(iterator position, list<T, Alloc>& x, iterator f, iterator l);`

Если **i** является допустимым итератором для списка L, а [**j1,j2**] – допустимым диапазоном для M, то следующая операция удаляет элементы этого диапазона и вставляет их перед **i**. L и M могут быть одним и тем же списком.

```
L.splice(i,M,j1,j2);
```

Контейнер list – функции *сцепки* - пример

```
#include <iostream>
using namespace std;
#include <list>
void main()
{ list <int> L;
  list <int>::iterator i,j1,j2,j;
  for (int k=1; k<=8; k++) {
      L.push_back(k);
      j = L.end();
      if(k==2) j1 = --j; else
          if (k==5) j2 = --j; else
              if (k==7) i = --j;
  }
  L.splice(i,L,j1,j2);
  copy(L.begin(), L.end(),
    ostream_iterator<int>(cout," "));
  cout << endl;
  char s; cin >>s;
}
```



Результат сцепки:
1 5 6 2 3 4 7 8

Важно!
Можно использовать —j, но нельзя : j2 = j – 1 (**для итератора такой операции нет!**)

Контейнер list – функция *remove*

Для удаления из списка всех элементов с заданным значением лучше всего использовать функцию-член класса **remove**.

```
Initial:
1 4 1 3 1 2
After remove:
4 3 2
```

```
#include <iostream>
using namespace std;
#include <list>
void out(char *s, const list<int> &L)
{ cout << s;
   copy(L.begin(), L.end(), ostream_iterator <int> (cout, " "));  cout << endl;
}

void main()
{ list <int> L;
  L.push_back(1); L.push_back(4); L.push_back(1);
  L.push_back(3); L.push_back(1); L.push_back(2);
  out ("Initial:\n",L);
  L.remove(1);
  out ("After remove:\n",L);
}
```

Контейнер list – функция *merge*

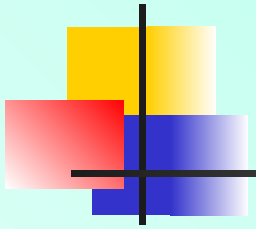
Для слияния элементов однотипных списков используется функция-член класса **merge**.

```
#include <iostream>
using namespace std;
#include <list>
void out(char *s, const list<int> &L)
{ cout << s;
   copy(L.begin(), L.end(), ostream_iterator <int> (cout, " "));  cout << endl;
}

void main()
{  list <int> L1, L2;
    L1.push_back(1); L1.push_back(4); L1.push_back(6);
    L2.push_back(3); L2.push_back(5); L2.push_back(8);
    out ("Initial 1:\n",L1);    out ("Initial 2:\n",L2);
    L1.merge(L2);
    out ("After merge:\n",L1);
}
```

```
Initial 1:
1 4 6
Initial 2:
3 5 8
After merge:
1 3 4 5 6 8
```

Все списки должны быть упорядочены в соответствии с некоторым отношением



Контейнер slist

Контейнер **slist** реализует односвязный список, в котором каждый элемент связан со следующим и не связан с предыдущим.

Этот контейнер реализован не во всех библиотеках STL, в частности, в Microsoft – версии, представленной в Studio, его нет.

Главное отличие этого списка от контейнера **list** состоит в том, что итераторы **list** являются двунаправленными (**bidirectional**) , а итераторы **slist** – однонаправленными итераторами (**forward**).

С точки зрения эффективности следует заметить, что функции вставки (**insert**), сцепки (**splice**) и удаления (**erase**) выполняются значительно медленнее, поскольку основаны на дополнительном переборе от начала списка перед выполнением собственно операции вставки и удаления.

Для ликвидации этого «перекоса» в интерфейс добавлены функции-члены **insert_after, splice_after** и **erase_after** , которыми и рекомендуется пользоваться всякий раз вместо общих функций **insert, splice** и **erase**.

Следует отметить, что в ряде случаев **slist** превосходит **list** и по затратам памяти, и по скорости работы.

Контейнер slist – пример

```
int main()
{
   slist<int> L;
   L.push_front(0);
   L.push_front(1);
   L.insert_after(L.begin(), 2);
   copy(L.begin(), L.end(),                    // The output is 1 2 0
         ostream_iterator <int>(cout, " "));
   cout << endl;
   slist<int>::iterator back = L.previous(L.end());
   back = L.insert_after(back, 3);
   back = L.insert_after(back, 4);
   back = L.insert_after(back, 5);
   copy (L.begin(), L.end(),                     // The output is 1 2 0 3 4 5
          ostream_iterator <int>(cout, " "));
   cout << endl;
}
```