Тест с отчетом 4

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

```
#include <iostream>
struct A {
    char foo() { return 'A';}
};
template<class T> struct B : public T {     virtual char foo() {return 'B';}  };
template<class T> struct C : public T {    virtual char foo() {return 'C';}  };
int main(int argc, char* argv[])
{
    A* a = new A();
    A* b = new B< A >();
    A* c = new C< A >();
    A* d = new B< C< A > >();
    std::cout << a->foo() << b->foo() << c->foo() << d->foo();
    return 0;
}
```

Ответ : AAAA

Параметры шаблонов B и C определяют родителя генерируемого класса. Так как метод foo() базового класса (класса A) невиртуальный, то в данном случае позднее связывание использовано не будет и во всех случаях будет вызван метод foo() класса A.

STL – прямой (однонаправленный) итератор

Прямые итераторы представляют собой комбинацию итераторов ввода и вывода. Они обладают всеми свойствами итераторов ввода и большинством свойств итераторов вывода. В отличие от итераторов ввода и вывода прямые итераторы могут ссылаться на один и тот же элемент коллекции и обрабатывать его по несколько раз.

Выражение	*Описание*
*iter	Обращение к элементу
iter->member	Обращение к переменной или функции элемента
++iter	Смещение вперед (возвращает новую позицию)
iter++	Смещение вперед (возвращает старую позицию)
iter1 == iter2	Проверка двух итераторов на равенство
iter1 !=iter2	Проверка двух итераторов на неравенство
TYPE()	Копирование итератора (конструктор по умолчанию)
TYPE(iter)	Копирование итератора (копирующий конструктор)
iter1 = iter2	Присваивание итератора

Почему прямой итератор не обладает всеми свойствами итератора вывода?

STL – прямой (однонаправленный) итератор

Существует одно важное ограничение, из-за которого код, действительный для итераторов вывода, оказывается недействительным для прямых итераторов.

- Итераторы вывода позволяют записывать данные без проверки конца последовательности. Более того, нельзя сравнить итератор вывода с конечным итератором, потому что итераторы вывода не поддерживают операцию сравнения.

Следовательно, для итератора вывода **pos** следующий цикл правилен, а для прямого итератора - нет:

```
// ОК для итераторов вывода
// ОШИБКА для прямых итераторов
while (true) {
   *pos = foo();
  ++pos;
 }
```

STL – прямой (однонаправленный) итератор

- При работе с прямыми итераторами перед разыменованием(обращением к данным) необходимо заранее убедиться в том, что это возможно. Приведенный цикл неправилен для прямых итераторов, поскольку он приводит к попытке обращения по итератору end() в конце коллекции с непредсказуемыми последствиями.

  Для прямых итераторов цикл необходимо изменить следующим образом:

```
// ОК для прямых итераторов
// ОШИБКА для итераторов вывода
whi1e (pos != col1.end() ) {
  *pos = foo();
  ++pos;
 }
```

Для итераторов вывода цикл не компилируется, потому что для них не определен оператор !=.

STL – двунаправленный итератор

Двунаправленными итераторами называются прямые итераторы, поддерживающие возможность перебора элементов в обратном направлении. Для этой цели в них определяется дополнительный оператор --.

Выражение	*Описание*
--iter	Смещение назад (возвращает новую позицию)
iter--	Смещение назад (возвращает старую позицию)

STL – итератор произвольного доступа

Итераторами произвольного доступа называются двунаправленные итераторы, поддерживающие прямой доступ к элементам. Для этого в них определяются «вычисления с итераторами» (по аналогии с математическими вычислениями с обычными указателями) - можно складывать и вычитать смещения, обрабатывать разности и сравнивать итераторы при помощи операторов отношения (таких, как < и >). Итераторы произвольного доступа поддерживаются следующими объектами и типами:

- контейнеры с произвольным доступом (**vector**, **deque**);
- строки (**string**, **wstring**);
- обычные массивы (указатели).

STL – итераторы вставки

Итератор вставки представляет собой итераторный адаптер, преобразующий присваивание нового значения во вставку нового значения. С помощью итераторов вставки алгоритмы вставляют новые значения вместо того, чтобы записывать их на место старых.

Все итераторы вставки относятся к категории итераторов вывода, то есть поддерживают только вставку новых значений. Обычно алгоритмы присваивают значения элементам, на которые ссылаются приемные итераторы. Для примера - алгоритм copy():

```
namespace std {
template <class InputIterator, class OutputIterator>
OutputIterator copy
   (Inputlterator from_pos, // Начало источника
    InputIterator from end, // Конец источника
    Outputlterator to pos) // Начало приемника
{
  while (from_pos != from_end) {
    *to_pos = *from_pos; // Копирование значений
    ++from pos;
    ++to_pos;
  }
  return to_pos:
}
```

STL – итератор вставки - пример

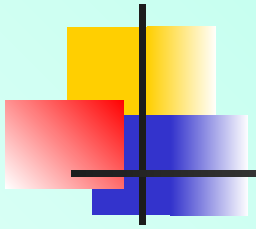
```
#include <iterator>
#include <list>
#include <iostream>
int main( )
{
   using namespace std;
   int i;       list <int>::iterator L_Iter;       list<int> L;
   for ( i = -2 ; i < 4 ; ++i )    {       L.push_back ( i );    }
                        // Вывод содержимого списка
   cout << "The list L is:\n ( ";
   for ( L_Iter = L.begin( ) ; L_Iter != L.end( ); L_Iter++)    cout << *L_Iter << " ";
   cout << ")." << endl;
                        // Используем шаблон функции для вставки элемента
   front_insert_iterator< list < int> > Iter (L);
   *Iter = 100;
                        // Как альтернативу можно использовать функцию-член front_inserter
   front_inserter ( L ) = 200;
   cout << "After the front insertions,\n the list L is:\n ( ";
   for ( L_Iter = L.begin( ) ; L_Iter != L.end( ); L_Iter++)
      cout << *L_Iter << " ";
   cout << ")." << endl;
   char s; cin >>s;
}
```

```
The list L is:
 ( -2 -1 0 1 2 3 ).
After the front insertions,
 the list L is:
 ( 200 100 -2 -1 0 1 2 3 ).
```

Простой объект функции

```
#include <iostream>
#include <vector>
#include <algorithm>
using namespace std;
class PrintInt {
public:
  void operator() (int elem) const {    cout << elem << " ";  }
};

void main()
{    vector<int> col;
    // Вставка элементов со значениями от 1 до 9
    for (int i = 1; i <= 9; ++i) {     col.push_back(i);   }
    // Вывод всех элементов
    for_each (col.begin(), col.end(), PrintInt()) ;
    cout << endl;
}
```



Простой объект функции – зачем это нужно? (1)

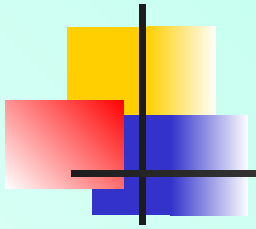
1) *Объект функции - это «умная функция».* Объекты, которые ведут себя как указатели, называются «умными указателями». По аналогии объекты, которые ведут себя как функции, можно назвать «умными функциями», поскольку их возможности не ограничиваются вызовом оператора ().

Объекты функций могут представлять другие функции и иметь другие атрибуты, а это означает, что объекты функций обладают **состоянием**. Одна и та же функция, представленная объектом функции, в разные моменты времени может находиться в разных состояниях. У обычных функций такая возможность отсутствует. Из этого следует одно из достоинств объектов функций – возможность их инициализации на стадии выполнения перед вызовом / использованием.

Простой объект функции – зачем это нужно? (2)

2) *Каждому объекту функции соответствует свой тип.* Обычные функции обладают разными типами только при различиях в их сигнатурах. С другой стороны, объекты функций могут иметь разные типы даже при полном совпадении сигнатур. Это открывает новые возможности для унифицированного программирования с применением шаблонов, потому что функциональное поведение может передаваться как параметр шаблона.

Например, это позволяет контейнерам разных типов использовать единственный объект функции в качестве критерия сортировки. Тем самым предотвращается возможное присваивание, слияние и сравнение коллекций, использующих разные критерии сортировки. Вы даже можете спроектировать иерархию объектов функций, например, для определения нескольких специализированных версий одного общего критерия.



Простой объект функции – зачем это нужно? (3)

3) *Объекты функций обычно работают **быстрее** функций*. Шаблоны обычно лучше оптимизируются, поскольку на стадии компиляции доступно больше информации. Следовательно, передача объекта функции вместо обычной функции часто повышает быстродействие программы.

Простой объект функции – зачем это нужно? (4)

Если приращение известно на стадии компиляции…

Предположим, что нам требуется провести увеличение всех элементов контейнера на 10. Вариант решения:

```
void add10 (int& elem )
{   elem += 10;    }

void func()
{
    vector<int> col;
    …
    for_each(col.begin(), col.end(),     // Интервал
                add10) ;                 // Операция
}
```

Простой объект функции – зачем это нужно? (5)

Если приращений несколько известно на стадии компиляции…

Можно оформить шаблон функции. Вариант решения:

```
template <int theValue>
void add (int& elem)
{    elem += theValue;  }

void func()
{
    vector<int> col;
    …
    for_each(col.begin(), col.end(),     // Интервал
                add<10>) ;                // Операция
}
```

Простой объект функции – зачем это нужно? (6)

Если приращение определяется только во время выполнения программы …

Используем объект функции. Вариант решения:

```
// Объект функции прибавляет к значению элемента приращение.
// заданное при его инициализации
class AddValue {
  private:
      int theValue; // Приращение
  public:           // Конструктор инициализирует приращение
      AddValue(int v) : theValue(v) {  }
  // Суммирование выполняется "вызовом функции" для элемента
  void operator() (int& elem) const
  {    elem += theValue;   }
} ;
```

```
for_each (col.begin(), col.end(),  AddValue(10)) ;   // использование
      AddValue(*col. begin()) // увеличить на первый элемент (а не
10…)
```

Стандартные объекты функций

Чтобы использовать стандартные объекты функций, необходимо включить в программу заголовочный файл <functional> (ранее – function.h)

Выражение	*Описание*
negate<type>()	-param
plus<type>()	param1+param2
minus<type>()	param1-param2
multiplies<type>()	param1*param2
divides<type>()	param1 / param2
modulus<type>()	param1 % param2
equal_to<type>()	param1 == param2
not_equal_to<type>O	param1 != param2
less<type>()	param1 < param2
greater<type>()	param1 > param2
less_equal<type>()	param1 <= param2
greater_equal<type>()	param1 >= param2
logical_not<type>()	! param
logical_and<type>()	param1 && param2
logical_or<type>()	param1 || param2

Функциональные адаптеры

Функциональным адаптером называется объект, который позволяет комбинировать объекты функций друг с другом, с определенными значениями или со специальными функциями. Функциональные адаптеры тоже объявляются в заголовочном файле <functional>. Например, в следующей команде выражение bind2nd(greater<int>(),42) создает комбинированный объект функции для проверки условия «целое число больше 42»:

```
find_if (col.begin(), col.end(),            // Интервал
         bind2nd(greater<int>() ,42)) // Критерий
```

Фактически bind2nd() преобразует бинарный объект функции (например, greater<>) в унарный объект функции. Второй параметр всегда используется в качестве второго аргумента бинарной функции, передаваемой в первом параметре. Следовательно, в приведенном примере greater<> всегда вызывается со вторым аргументом, равным 42.

Классы стандартных функциональных адаптеров:

Выражение	Описание
bind1st(op,value)	op(value,param)
bind2nd(op,value)	op(value,param)
not1(op)	!op(param)
not2(op)	!op(param1, param2)

Функциональные адаптеры

Функциональные адаптеры сами по себе являются объектами функций, поэтому их можно объединять с другими адаптерами и объектами функций для построения более мощных (и более сложных) выражений. Например, следующая команда находит первый четный элемент коллекции:

```
pos = find_if (col.begin(), col.end(),          // Интервал
      not1( bind2nd( modulus<int>(), 2)));   // Критерий
```

в этой команде выражение bind2nd(modulus<int>(), 2) возвращает 1 для нечетных значений. Следовательно, оно может использоваться в качестве критерия для нахождения первого нечетного элемента, потому что значение 1 эквивалентно true. Адаптер not1() производит логическую инверсию результата, поэтому вся команда ищет первый элемент с четным значением.

Объединяя объекты функций при помощи функциональных адаптеров, можно строить достаточно сложные выражения. Подобный стиль программирования называется **функциональной композицией**.

Функциональные адаптеры

Функциональные адаптеры для функций классов позволяют вызвать некоторую функцию класса для каждого элемента коллекции

Выражение	*Описание*
mem_fun_ref(op)	Вызов op() как функции объекта
mem_fun (op)	Вызов op() как функции указателя на объект

```
class Person {
…
void print () const { … }
void printWithPrefix (std::string prefix) const  { … }
…
};
```

```
                // Вызов функции print() без аргументов для каждого элемента
for_each (col.begin(), col.end(),
      mem_fun_ref(&Person::print));
                // Вызов функции printWithPrefix() для каждого элемента
                // строка "person: " передается при вызове
for_each (col.begin(), col.end(),
           bind2nd(mem_fun_ref(&Person::printWithPrefix), "person: "));
```

Алгоритмы STL

Чтобы использовать алгоритмы библиотеки, необходимо включить в программу заголовочный файл <**algorithm**>.

Некоторые алгоритмы STL, предназначенные для обработки числовых данных, определяются в заголовочном файле <**numeric**>.

Любой алгоритм STL работает с одним или несколькими интервалами, заданными при помощи итераторов. Для первого интервала обычно задаются обе границы (начало и конец), а для остальных интервалов часто достаточно одного начала, потому что конец определяется количеством элементов в первом интервале. Перед вызовом необходимо убедиться в том, что заданные интервалы действительны, то есть начало интервала предшествует концу или совпадает с ним, а оба итератора относятся к одному контейнеру.

Алгоритмы работают в режиме замены, а не в режиме вставки, поэтому перед вызовом алгоритма необходимо убедиться в том, что приемный интервал содержит достаточное количество элементов. Специальные итераторы вставки переводят алгоритм в режим вставки.

Для повышения мощи и гибкости некоторые алгоритмы позволяют передавать пользовательские операции, которые вызываются при внутренней работе алгоритма. Такие операции оформляются в виде функций или объектов функций.

Предикаты применяются . . .

Функция, возвращающая логическое значение, называется *предикатом*.

Предикаты применяются в следующих ситуациях:

- Функция (или объект функции), определяющая унарный предикат, может передаваться алгоритму поиска в качестве критерия поиска. Унарный предикат проверяет, соответствует ли элемент заданному критерию. Например, это позволяет найти первый элемент со значением, меньшим 50.
- Функция (или объект функции), определяющая бинарный предикат, может передаваться алгоритму сортировки в качестве критерия сортировки. Бинарный предикат сравнивает два элемента. Например, с помощью бинарного предиката можно отсортировать объекты, представляю щи е людей, по фамилиям и по именам.
- Унарный предикат может использоваться как критерий, определяющий, к каким элементам должна применяться операция. Например, из коллекции можно удалить только элементы с нечетными значениями.
- Предикаты также используются для модификации операций в численных алгоритмах. Например, алгоритм **accumulate()**, обычно вычисляющий сумму элементов, также позволяет вычислять произведения всех элементов.

Классификация алгоритмов (по основным областям применения)

По названию алгоритма можно получить первое представление о его назначении. Проектировщики STL ввели два специальных суффикса.

Суффикс **_if.** *Суффикс **_if** используется при наличии двух похожих форм* алгоритма с одинаковым количеством параметров; первой форме передается значение, а второй - функция или объект функции. В этом случае версия без суффикса **_if** используется при передаче значения, а версия с суффиксом **_if** - при передаче функции или объекта функции. Например, алгоритм **find()** ищет элемент с заданным значением, а алгоритм **find_if()** элемент, удовлетворяющий критерию, определенному в виде функции или объекта функции.

Впрочем, не все алгоритмы, получающие функции и объекты функций, имеют суффикс **_if**. Если такая версия вызывается с дополнительными аргументами, отличающими ее от других версий, за ней сохраняется прежнее имя. Например, версия алгоритма **min_element()** с двумя аргументами находит в интервале минимальный элемент, при этом элементы сравниваются оператором <. В версии **min_element()** с тремя аргументами третий аргумент определяет критерий сравнения.

Суффикс **_copy**. Суффикс **_copy** означает, что алгоритм не только обрабатывает элементы, но и копирует их в приемный интервал. Например, алгоритм **reverse()** переставляет элементы интервала в обратном порядке, а **reverse_copy()** копирует элементы в другой интервал в обратном порядке.

Классификация алгоритмов (по группам)

- немодифицирующие алгоритмы;
- модифицирующие алгоритмы;
- алгоритмы удаления;
- перестановочные алгоритмы;
- алгоритмы сортировки;
- алгоритмы упорядоченных интервалов;
- численные алгоритмы.

Если алгоритм принадлежит сразу нескольким категориям, он описывается только в одной из них.

Немодифицирующие алгоритмы

Немодифицирующие алгоритмы сохраняют как порядок следования обрабатываемых элементов, так и их значения. Они работают с итераторами ввода и прямыми итераторами и поэтому могут вызываться для всех стандартных контейнеров.

Название	*Описание*
for_each()	Выполняет операцию с каждым элементом (* - возможна и модификация!)
count()	Возвращает количество элементов
count_if()	Возвращает количество элементов, удовлетворяющих заданному критерию
min_element()	Возвращает элемент с минимальным значением
max_element()	Возвращает элемент с максимальным значением
find()	Ищет первый элемент с заданным значением
find_if()	Ищет первый элемент, удовлетворяющий заданному критерию
search_n()	Ищет первые n последовательных элементов с заданными свойствами
search()	Ищет первое вхождение подинтервала
find_end()	Ищет последнее вхождение подинтервала
find_first_of()	Ищет первый из нескольких возможных элементов
adjacent_find()	Ищет два смежных элемента, равных по заданному критерию
equal()	Проверяет, равны ли два интервала
mismatch()	Возвращает первый различающийся элемент в двух интервалах
lexicographical_compare()	Проверяет, что один интервал меньше другого по лексикографическому критерию

Модифицирующие алгоритмы

Модифицирующие алгоритмы изменяют значения элементов. Модификация производится непосредственно внутри интервала или в процессе копирования в другой интервал. Если элементы копируются в приемный интервал, исходный интервал остается без изменений.

Название	*Описание*
for_each()	Выполняет операцию с каждым элементом
copy()	Копирует интервал, начиная с первого элемента
copy_backwards()	Копирует интервал, начиная с последнего элемента
transform()	Модифицирует (и копирует) элементы; объединяет элементы двух интервалов
merge()	Производит слияние двух интервалов
swap_ranges()	Меняет местами элементы двух интервалов
fill()	Заменяет каждый элемент заданным значением
fill_n()	Заменяет n элементов заданным значением
generate()	Заменяет каждый элемент результатом операции
generate_n()	Заменяет n элементов результатом операций
replace()	Заменяет элементы с заданным значением другим значением
replace_if()	Заменяет элементы, соответствующие критерию, заданным значением
replace_copy()	Заменяет элементы с заданным значением при копировании интервала
replace_copy_if()	Заменяет элементы, соответствующие критерию, при копировании интервала

Модифицирующие алгоритмы

В группе модифицирующих алгоритмов центральное место занимают алгоритмы **for_each()** (снова) и **transform()**. Оба алгоритма используются для модификации элементов последовательности, однако работают они по-разному.

Алгоритму **for_each()** передается операция, которая модифицирует его аргумент. Это означает, что аргумент должен передаваться по ссылке. Пример:

```
void square (int& elem) // Передача по ссылке
{  elem=elem*elem ;  }    // Прямое присваивание вычисленного
значения
...
for_each (col.begin(), col.end(),  // Интервал
           square) ;            // Операция
```

Алгоритм **transform()** использует операцию, которая возвращает модифицированный аргумент. Результат присваивается исходному элементу. Пример:

```
void square (int elem)  // Передача по значению
{  return elem*elem;  }
...
for_each (col.begin(), col.end(),  // Интервал
          col.begin(),                 // Приемник
          square) ;          // Операция
```

Алгоритм **transform()** работает чуть медленнее. С другой стороны, этот подход более гибок, поскольку он также может использоваться для модификации элементов в процессе копирования в другой интервал. Кроме того, у алгоритма **transform()** есть еще одна версия, которая позволяет обрабатывать и комбинировать элементы из двух разных интервалов.

Алгоритмы удаления

Алгоритмы удаления составляют отдельную подгруппу модифицирующих алгоритмов. Они предназначены для удаления элементов либо в отдельном интервале, либо в процессе копирования в другой интервал. Как и в случае с модифицирующими алгоритмами, их приемником не может быть ассоциативный контейнер, поскольку элементы ассоциативного контейнера считаются const.

Название	*Описание*
remove()	Удаляет элементы с заданным значением
remove_if()	Удаляет элементы по заданному критерию
remove_copy()	Копирует элементы, значение которых отлично от заданного
remove_copy_if()	Копирует элементы, не соответствующие заданному критерию
unique()	Удаляет смежные дубликаты (элементы, равные своему предшественнику)
unique_copy()	Копирует элементы с удалением смежных дубликатов

Алгоритмы удаления ограничиваются только «логическим» удалением элементов, т.е. их перезаписью следующими элементами, которые не были удалены. Т.о., количество элементов в интервалах, с которыми работают алгоритмы, остается прежним, а алгоритм возвращает позицию нового «логического конца» интервала.Программист может использовать ее по своему усмотрению (например, физически уничтожить удаленные элементы).

Перестановочные алгоритмы

Перестановочными алгоритмами называются алгоритмы, изменяющие порядок следования элементов (но не их значения) посредством присваивания и перестановки их значений. Их приемником не может быть ассоциативный контейнер, поскольку элементы ассоциативного контейнера считаются const.

Название	*Описание*
reverse()	Переставляет элементы в обратном порядке
reverse_copy()	Копирует элементы, переставленные в обратном порядке
rotate()	Производит циклический сдвиг элементов
rotate_copy()	Копирует элементы с циклическим сдвигом
next_permutation()	Переставляет элементы
prev_permutation()	Переставляет элементы
random_shuffle()	Переставляет элементы в случайном порядке
partition()	Изменяет порядок следования элементов так, что элементы, соответствующие критерию, оказываются спереди
stable_partition()	То же, что и partition(), но с сохранением относительного расположения элементов, соответствующих и не соответствующих критерию

Алгоритмы сортировки

Алгоритмы сортировки являются частным случаем перестановочных алгоритмов, поскольку они тоже изменяют порядок следования элементов. Тем не менее сортировка является более сложной операцией и обычно занимает больше времени, чем простые перестановки. На практике эти алгоритмы требуют поддержки итераторов произвольного доступа (для приемника).

Название	*Описание*
sort()	Сортирует все элементы
stable_sort()	Сортирует с сохранением порядка следования равных элементов
partial sort()	Сортирует до тех пор, пока первые **n** элементов не будут упорядочены правильно
nth_element()	Сортирует элементы слева и справа от элемента в позиции **n**
partition()	Изменяет порядок следования элементов так, что элементы, соответствующие критерию, оказываются спереди
stable_partition()	То же, что и partition(), но с сохранением относительного расположения элементов, соответствующих и не соответствующих критерию
make_heap()	Преобразует интервал в кучу
push_heap()	Добавляет элемент в кучу
pop_heap()	Удаляет элемент из кучи
sort_heap	Сортирует кучу (которая после вызова перестает быть кучей)

Алгоритмы сортировки (sort)

Алгоритмы сортировки часто критичны по времени, поэтому стандартная библиотека C++ содержит несколько алгоритмов, различающихся по способу сортировки или составу сортируемых элементов (полная/частичная сортировка).

sort(). Исторически этот алгоритм основан на механизме быстрой сортировки, гарантирующем хорошую среднестатистическую сложность (n*log(n)), но очень плохую (квадратичную) сложность в худшем случае:

```
/* Сортировка всех элементов
* - средняя сложность n*log(n)
* - квадратичная сложность n*n в худшем случае
*/
sort (col.begin(), col.end());
```

Если в вашей ситуации очень важно избежать худших случаев, воспользуйтесь другим алгоритмом, например **partial_sort()** или **stable_sort()**.

Алгоритмы сортировки (partial_sort)

Исторически этот алгоритм основан на механизме сортировки в куче (**heapsort**), гарантирующем сложность n*log(n) в любом случае. Тем не менее обычно сортировка в куче выполняется в 2-5 раз медленнее быстрой сортировки. И так, с учетом реализаций **sort()** и **partial_sort()**, алгоритм **partial_sort()** лучше по сложности, но по скорости работы **sort()** обычно превосходит его. Преимущество алгоритма **partial_sort()** заключается в том, что сложность n*log(n) гарантирована и никогда не ухудшается до квадратичной сложности.

Алгоритм **partial_sort()** также обладает особым свойством: он прекращает сортировку, когда требуется отсортировать только ***n*** *первых элементов.* Чтобы отсортировать все элементы, передайте конец последовательности во втором и в последнем аргументах;

```
/* Сортировка всех элементов
* - всегда сложность n*log(n)
* - но обычно работает вдвое медленнее sort()
*/
partial_sort (col.begin(), col.end(), col.end());
```

Алгоритмы сортировки (stable_sort)

Исторически этот алгоритм основан на механизме сортировки со слиянием.

Для достижения сложности n*log(n) необходима дополнительная память, в противном случае алгоритм выполняется со сложностью n*log(n)*log(n). Достоинством stable_sort() является сохранение порядка следования равных элементов.

```
/* Сортировка всех элементов
* - сложность n*log(n) или n*log(n)*log(n)
*/
stable_sort (col.begin(), col.end());
```

Алгоритм сортирует все элементы переданного интервала.

Помимо перечисленных существуют и другие алгоритмы сортировки элементов. Например, алгоритмы сортировки в куче вызывают функции, непосредственно работающие с кучей (то есть с бинарным деревом, используемым в реализации этих алгоритмов). Алгоритмы сортировки в куче заложены в основу эффективной реализации приоритетных очередей.

Алгоритмы сортировки (nth_element)

Алгоритм nth_element() прекращает работу, когда n-й элемент последовательности занимает правильное место в соответствии с критерием сортировки.

Для остальных элементов он гарантирует только то, что предшествующие элементы имеют меньшее либо равное, а последующие элементы – большее либо равное значение. Таким образом алгоритм nth_element() разделяет элементы на два подмножества в соответствии с критерием сортировки.

Алгоритму nth_element() передается требуемое количество элементов в первой части (а следовательно, и во второй). Пример:

```
// Перемещение четырех наименьших элементов в начало
nth_element (col.begin(),  // Начало интервала
col.begin()+3,             // Позиция, отделяющая первую часть от второй
col.end());                // Конец интервала
```

Но после вызова невозможно сказать, по какому критерию первая часть отличается от второй. Более того, в обеих частях могут присутствовать элементы, совпадающие по значению с n-м элементом.

```
3,7,8,3,2,1,4,5,6,3,5
1,2,3,3,3,4,5,5,6,7,8
```

Алгоритмы сортировки (partition и stable_partition)

Алгоритму **partition()** передается конкретный критерий сортировки, определяющий различия между первой и второй частями:

```
// Перемещение всех элементов, меньших 3, в начало
vector<int>::iterator  pos;
pos = partition (col.begin(), col.end(),
bind2nd(less<int>(),7));
```

На этот раз после вызова невозможно сказать, сколько элементов входит в первую и вторую части. Возвращаемый итератор **pos** указывает на первый элемент второй части, которая содержит все элементы, не соответствующие критерию.

Алгоритм **stable_partition()** в целом аналогичен **partition()**, но он дополнительно гарантирует сохранение порядка следования элементов в обеих частях по отношению к другим элементам, входящим в ту же часть.

Любому алгоритму сортировки в необязательном аргументе всегда можно передавать критерий сортировки. По умолчанию критерием сортировки является объект функции **less< >**, а элементы сортируются по возрастанию значений. Приемником алгоритмов сортировки не может быть *ассоциативный контейнер*. Алгоритмы сортировки не могут вызываться для *списков*.

Алгоритмы упорядоченных интервалов

Алгоритмы упорядоченных интервалов требуют, чтобы интервалы, с которыми они работают, были изначально упорядочены по соответствующему критерию. Достоинством этих алгоритмов является невысокая сложность.

Название	*Описание*
binary_search()	Проверяет, содержит ли интервал заданный элемент
includes()	Проверяет, что каждый элемент интервала также является элементом другого интервала
lower_bound()	Находит первый элемент со значением, большим либо равным заданному
upper_bound()	Находит первый элемент со значением, большим заданного
equal_range()	Возвращает количество элементов в интервале, равных заданному значению
merge()	Выполняет слияние элементов двух интервалов
set_union()	Вычисляет упорядоченное объединение двух интервалов
set_intersection()	Вычисляет упорядоченное пересечение двух интервалов
set_difference()	Вычисляет упорядоченный интервал, который содержит все элементы интервала, не входящие в другой интервал
set_symmetric_ difference()	Вычисляет упорядоченный интервал, содержащий все элементы, входящие только в один из двух интервалов
inplace_merge()	Выполняет слияние двух последовательных упорядоченных интервалов

Тест с отчетом 5

1) Какие методы компилятор может автоматически сгенерировать для данного класса?

class A { };

(**1**) конструктор по умолчанию,
конструктор копирования,
оператор operator=(const A&),
деструктор

(**2**) конструктор копирования
оператор operator=(const A&)
деструктор

(**3**) конструктор по умолчанию
конструктор копирования
деструктор

(**4**) конструктор по умолчанию
деструктор

2) Чему будет равно значение переменной X после выполнения фрагмента кода:

```
int arr[] = { 1 , 2 };
 int X = (arr[1] - arr[0]) [arr];
```

Варианты: (1) 0
(2) ошибка на этапе компиляции
(3) 2
(4) 1