Курс лекций по олимпиадной информатике (C)

Лекция 5. STL

Михаил Густокашин, 2009

Конспекты лекций подготовлены для системы дистанционной подготовки, действующей на сайте **informatics.mccme.ru**.

При нахождении ошибок или опечаток просьба сообщать по адресу

gustokashin@gmail.com

Версия C от 21.11.2009

1 Введение

STL расшифровывается как **Standard Template Library**. По-русски это звучит как «Стандартная библиотека шаблонов».

Механизм шаблонов и библиотека **STL** относятся к языку **C++**, поэтому мы не будем сильно углубляться в синтаксис и часть моментов оставим без подробного описания. Программы, использующие **STL** можно компилировать только компилятором языка **C++** (например, **g++**)!

Вкратце шаблоны можно характеризовать как функции, для которых задается тип входных параметров. Например, **QuickSort** одинаково хорошо может сортировать и целые, и вещественные числа, а также другие типы данных. Однако без использования шаблонов нам пришлось бы создавать свою функцию для каждого из типов данных, при этом очень часто функции различались бы только в заголовке и типах локальных переменных. Использование шаблона значительно ускоряет разработку больших проектов и библиотек, но их непосредственное использование на олимпиаде неоправданно, т.к. значительно проще сделать copy-paste и внести необходимые изменения в новую функцию.

STL предоставляет собой набор шаблонных функций, которые реализуют функциональность многих структур данных и некоторых алгоритмов. В задачах, где требуется выполнить чисто технические действия (а такие действия почти всегда бывают в олимпиадных задачах) уместно использовать **STL**, т.к. это ускоряет процесс написания программы. Тем не менее, это не означает, что можно забыть методы сортировки и реализацию структур данных, т.к. часто решение задачи основывается на каком-либо алгоритме, но не может быть реализовано с помощью библиотечной функции (например, наложены какие-либо ограничения, которые невозможно или очень сложно отразить в библиотеке).

Таким образом, знание **STL** может избавить нас от рутинной работы, а это не только приятно, но и полезно.

Книги, посвященные **STL**, нередко содержат несколько сотен страниц, поэтому, естественно, в лекции будет охвачена далеко не вся функциональность библиотеки. В этой лекции мы будем рассматривать возможности **STL** только для уже изученных структур данных и алгоритмов, в дальнейшем будем указывать возможности **STL** по мере изучения новых структур данных и алгоритмов.

Для того чтобы **STL** работал, в начале программы (после всех `#include`) необходимо написать `using namespace std;` Эта команда указывает, что надо пользоваться пространством имен `std`. В нашем курсе суть команды не важна, главное — нам необходимо знать, что это накладывает на нас ограничения - запрещено использовать некоторые слова, которые становятся «ключевыми» (например: `stack`, `queue`, `vector`, `sort`, `count`, ...).

2 Пара (pair)

Пара, фактически является шаблонной структурой, которая содержит два поля (возможно, разных типов), называются они `first` и `second`.

Для того чтобы использовать `pair`, необходимо подключить библиотеку `<utility>` (обратите внимание, что `.h` писать не надо).

Пусть, например, мы хотим создать пару из целого и вещественного числа. Тогда ее создание будет выглядеть следующим образом:

```
pair <int, double> p;
```

Теперь мы можем обращаться к `p` точно так же, как к обычной структуре, например, так:

```
p.first = 5; p.second = 3.1415;
```

Пары одинакового типа можно присваивать друг другу. Пары используются в ассоциативных контейнерах (о них будет сказано позже). Поля пары могут быть не только элементарного, но и составного типа, например, опять же парой. Для примера, приведем реализацию структуры, хранящей дату с использованием пар (год, месяц, день):

```
pair <int, pair <int, int> > date1, date2;
```

Обратите внимание, что `>` разделены пробелом, если не разделять их, то эта запись будет интерпретироваться как оператор сдвига вправо `>>`, что приведет к ошибке при компиляции.

Обращение к полям будет выглядеть так:

```
int year = date1.first;
int month = date1.second.first;
int day = date1.second.second;
```

Не очень удобный формат, но можно придумать макросы, которые облегчат нам доступ к полям.

Крайне полезное свойство состоит в том, что пары можно сравнивать. При этом сравнение идет слева направо. Т.о. для нашей даты сначала сравнятся годы, при равных годах сравнятся месяцы и т.д. Это очень удобно использовать при сортировке. На этом закончим рассмотрение пар и перейдем к более осмысленным структурам данных.

3 Стек (stack)

Стек уже знакомая нам структура данных. Удобство использование стека **STL** состоит в том, что у нас нет необходимости заранее задавать размер стека, однако за это приходится расплачиваться накладными расходами. Фактически, **STL** реализует стек

на структуре, которая работает аналогично расширяемому массиву из второй лекции ($\log N$ выделений памяти и двукратное увеличение требуемой памяти).

Чтобы использовать стек, необходимо подключить библиотеку `<stack>`. Приведем пример программы, а затем разберемся со всеми использованными функциями:

```
#include <stack>
#include <stdio.h>

using namespace std;

int main()
{
  stack <int> S;
  S.push(8);
  S.push(7);
  int x = S.size(); //x==2
  while (!S.empty()) {
    printf("%d ", S.top());
    S.pop();
  }
  return 0;
}
```

Вывод этой программы будет 78 (как и положено стеку).

Вначале мы создаем стек для целых чисел `stack <int>`, при этом он пуст. Для стека определены следующие функции:

`void push(<type>)` — добавление элемента в стек.

`void pop()` — удаляет элемент с вершины стека.

`<type> top()` — возвращает элемент с вершины стека.

`unsigned int size()` — определяет размер стека (количество элементов).

`bool empty()` — возвращает истину, если стек пуст.

Эти функции являются методами класса. Для тех, кто не знает **C++** следует понимать их аналогично полям структуры (синтаксис обращения к методам класса такой же, как к полям структуры).

4 Очередь (queue)

Для работы с очередью необходимо подключить библиотеку `<queue>`. Очередь также называется `queue`. Чтобы создать очередь из вещественных чисел следует наисать `queue <double> q;`

У очереди имеются следующие методы:

`void push(<type>)` — добавление элемента в очередь.

`void pop()` — удаляет элемент из головы очереди.

`<type*> front()` — возвращает ссылку на элемент из головы очереди.

`<type*> back()` — возвращает ссылку на элемент из хвоста очереди.

`unsigned int size()` — определяет размер очереди (количество элементов).

`bool empty()` — возвращает истину, если очередь пуста.

5 Дек (deque)

Для работы с деком необходимо подключить библиотеку `<deque>`. Чтобы создать дек из символов следует наисать `deque <char> d;`

Очереди и стеки в **STL** реализуются над деком (т.к. дек позволяет реализовывать функциональность и того и другого).

Некоторые методы дека:

`void push_back(<type>)` — добавление элемента в конец дека.

`void push_front<type>)` — добавление элемента в начало дека.

`void pop_back()` — удаляет элемент с конца дека.

`void pop_front()` — удаляет элемент с начала дека.

`<type*> front()` — возвращает ссылку на элемент из головы дека.

`<type*> back()` — возвращает ссылку на элемент из хвоста дека.

`unsigned int size()` — определяет размер дека (количество элементов).

`bool empty()` — возвращает истину, если дек пуст.

6 Очередь с приоритетами (куча, priority_queue)

Для работы с кучей (очередью с приоритетами) необходимо подключить библиотеку `<queue>`. Куча называется `priority_queue`. Чтобы создать кучу из вещественных чисел следует наисать `priority_queue <double> h;`

Некоторые методы кучи:

`void push(<type>)` — добавление элемента в кучу.

`void pop()` — удаляет наибольший элемент из кучи.

`<type> top()` — возвращает наибольший элемент в куче.

`unsigned int size()` — определяет размер кучи (количество элементов).

`bool empty()` — возвращает истину, если куча пуста.

7 Динамически расширяемый массив (vector)

`vector` в **STL** ведет себя точно также, как динамически расширяемый массив из второй лекции. Для того чтобы `vector` работал необходимо подключить библиотеку `<vector>`.

Мы можем создавать пустой вектор так: `vector <int> v;`

Кроме того, мы можем создать вектор с заранее заданным начальным количеством элементов: `vector <int> v(10);`

Обратите внимание, что запись `vector <int> v[10];` создаст массив из 10 пустых векторов.

Для добавления элементов в вектор следует использовать уже знакомый нам `push_back`. Доступ к элементам вектора можно осуществлять так же, как и в обычном массиве, задавая индекс в квадратных скобках.

Кроме того, в векторе существуют и некоторые другие полезные методы:

`void push_back(<type>)` — добавление элемента в конец вектора. При необходимости (вектор заполнен) он расширяется вдвое.

`unsigned int size()` — определяет размер вектора (количество элементов).

`unsigned int capacity()` — возвращает максимальное количество элементов, которое может поместится в вектор без расширения.

`void resize()` — изменяет размер вектора. Эта операция аналогична `realloc`.

`void clear()` — очищает содержимое вектора (размер вектора становится равным нулю).

8 Вектор битов (bit_vector)

В библиотеке `<vector>` также существует специальный тип для `vector <bool>` (логические переменные, принимающие значения `true` и `false`) — `bit_vector`. Этот тип не просто создает вектор из булевских переменных (каждая из которых занимает 1 байт), а оптимизирует расположение так, что в один байт помещается 8 `bool`-переменных. Работа его аналогична битовому массиву, но мы лишаемся возможности оперировать целыми наборами битов в рамках одной переменной (что часто бывает удобно и является необходимым этапом решения задачи). Таким образом, область применения `bit_vector` весьма ограничена. Для него существуют операции `resize`, обращение по индексу и некоторые другие. Проверьте, что ваш компилятор поддерживает `bit_vector` (этот тип доступен далеко не во всех компиляторах).

9 Строка (string)

Класс `string` является расширением класса `vector <char>` и содержит некоторые специфичные для строк методы. `string` также умеет динамически расширяться, однако следует помнить, что это требует некоторых накладных расходов (как и в динамически расширяемом массиве).

В принципе, значительная часть функциональности работы со строками реализована в библиотеке `<string.h>` (функции `strcmp`, `strcat`, `strcpy`, `strlen` и др.), но использование `string`-объектов часто бывает удобнее.

Для использования `string` необходимо подключить библиотеку `<string>`. Сам объект string создается так: `string s1, s2;`

Для `string`-объектов определена операция сложения, т.е. можно написать `s1 + s2` и результатом этой операции будет конкатенация строк (строка `s2` будет дописана в конец строки `s1`). Также определена операция присваивания, которая создает копию строки (а не присваивает указатель, как в случае `char[]`). Кроме того, для `string` переопределены операции `<<` и `>>`, которые позволяют выводить и вводить `string` с помощью потоков ввода-вывода. Строки можно сравнивать между собой, пользуясь обычными операциями сравнения (`<`, `>` и т.д.)

`string` можно присваивать обычным C-массивам типа `char`, массивы могут участвовать в операциях сложения, сравнения и т.п. (для них неявно строится `string`-обертка)

Объект `string` поддерживает всю функциональность объекта `vector` (т.е. методы, применимые к `vector` могут быть также применены к `string` с тем же синтаксисом). Рассмотрим некоторые специфичные для `string` методы:

`char* c_str()` — возвращает указатель на массив `char`, который содержит C-строку. Этим методом удобно пользоваться, если используется вывод библиотеки `<stdio.h>` и в некоторых других ситуациях.

`unsigned int length()` — возвращает длину строки. Предпочтительнее использовать этот метод, а не `size()`, т.к. реализация строки может быть не только на векторе, и в таком случае функция `length` будет работать быстрее.

`unsigned int find(string substr)` — ищет подстроку `substr` в строке. Возвращает индекс, с которого начинается первое вхождение подстроки в строку.

`unsigned int find(string substr, unsigned int from)` — ищет подстроку `substr` в строке начиная с позиции `from`. Это удобно использовать для поиска всех данных подстрок.

`unsigned int rfind(string substr)` — ищет подстроку `substr` в строке. Возвращает индекс, с которого начинается последнее вхождение подстроки в строку. Фактически, то же самое что и `find`, но мы начинаем искать вхождения с конца.

`unsigned int rfind(string substr, unsigned int from)` — ищет последнюю подстроку `substr` в строке, индекс начала которой меньше `from`.

`unsigned int find_first_of(string substr)` — ищет первое вхождение какого-либо символа из `substr` в строке. Существует также аналог с параметром `from`, функция `find_last_of`, которая делает то же самое, но с конца строки, а также функция `find_first_not_of`, она ищет первый символ в строке, который не входит в `substr` (также существует `find_last_not_of`). Использование этих функций позволяет, например, разбить строку на токены (слова), передавая в качестве `substr` строку, содержащую знаки препинания и пробелы, например “ ,.\t\n)”. Это часто бывает полезно, когда необходио разбить текст на отдельные слова (встроенных средств для этого, к сожалению, нет, и приходится реализовывать это вручную).

`string substr(unsigned int n)` — создает новую строку, содержащую первые n символов строки.

`string substr(unsigned int k, unsigned int n)` — создает новую строку, содержащую n символов строки, начиная с k-го.

Также существует много других методов и их вариаций, подробнее о которых можно прочитать в соответствующем разделе помощи. Отметим методы `replace` (удаление подстрок) и `insert` (вставка подстрок), которые часто бывают полезны, но обилие их вариаций не позволяет описать их в рамках одной лекции.

10 Итераторы

Итератор — универсальный способ доступа к элементам контейнера. Итератор можно представить себе как указатель, для которого определено несколько операций. В частности, полезными для нас будут операции разыменовывания `(*it)` — доступ к данным по этому «указателю» (можно также использовать `->` для обращения к полям), инкремента (`++`) — переход к следующему объекту в контейнере и сравнение итераторов (`==` и `!=`). На самом деле, операций для итераторов несколько больше, но мы не будем их использовать.

Итераторы делятся на 3 типа: произвольного доступа (мы не будем их использовать, т.к. они либо не поддерживаются контейнером, либо могут быть заменены на обращение по индексу), последовательного и обратного доступа. Итератор последовательного доступа удобен для последовательной обработки элементов контейнера, а итератор обратного доступа осуществляет доступ к элементам контейнера в обратном порядке.

Для любого контейнера определены два итератора: начала и конца. Допустим, мы

хотим вывести все числа из вектора (`vector <int> v`) последовательно. Соответствующий код будет выглядеть так:

```
vector <int>::iterator it;
for (it = v.begin(); it != v.end(); it++)
  printf("%d ", *it);
```

Чтобы создать итератор по какому-либо контейнеру, необходимо сначала указать тип контейнера (`vector <int>` в нашем случае), затем написать `::iterator` и указать имя итератора.

Чтобы итератор указывал на первый элемент контейнера, необходимо присваивание `it = v.begin()`. Метод `begin()` возвращает указатель на первый элемент любого контейнера. Мы будем идти до последнего элемента. Это реализуется с помощью `it != v.end()`. `end()` указывает на следующий после последнего элемент контейнера. Переход к следующему элементу осуществляется с помощью операции инкремента `it++`. При выводе мы просто разыменовываем итератор и получаем его содержимое.

Теперь выведем наш вектор в обратном направлении. Это делается очень похоже:

```
vector <int>::reverse_iterator it;
for (it = v.rbegin(); it != v.rend(); it++)
  printf("%d ", *it);
```

Обратите внимание, что при создании итератора у нас появилось `reverse_`, а при указании начала и конца вектора в начале появилась буква `r`.

Остальные свойства итераторов будем изучать при рассмотрении новых контейнеров и алгоритмов.

11 Хеш-таблица (hash_set)

Хеш-таблица — уже знакомая нам структура, которая позволяет добавлять, удалять и проверять наличие элемента во множестве за $O(1)$. В этом разделе мы будем рассматривать хеш-таблицы только от элементарных типов (об использовании `hash_set` для других типов можно прочитать в файле помощи по языку).

Для того чтобы `hash_set` работал, необходимо подключить библиотеку `<hash_set>`. В некоторых компиляторах `hash_set` находится не в пространстве имен `std`, а в пространстве `stdext`. Если ваш компилятор относится к такому типу, то следует вначале писать `using namespace stdext;`

Допустим, мы хотим создать хеш-таблицу для целых чисел. Это запишется так `hash_set <int> h;`

Для нас будут полезны следующие методы:

`void insert(<type>)` — добавление элемента в хеш-таблицу.

`hash_set <type>::iterator find(<type>)` — возвращает итератор, указывающий на элемент, который передается в качестве параметра. Если такого элемента не существует, то результат будет равен значению метода `end()` для соответствующей хеш-таблицы. Изменять значение по этому итератор нельзя!

`void erase(hash_set <type>::iterator)` — удаляет значение, на которое указывает итератор, из хеш-таблицы.

`void clear()` — очищает содержимое хеш-таблицы.

Интересующиеся могут самостоятельно изучить класс `hash_multiset`, в котором может быть несколько одинаковых элементов.

12 Хеш-словарь (hash_map)

Хеш-словарь очень похож на хеш-таблицу, но принимает в качестве значения пару, при этом хеширование идет только по первому элементу пары. Например, мы можем хранить пару ICQ UIN и имя пользователя. При этом первый элемент в паре должен быть уникальным.

Для того чтобы `hash_map` работал, необходимо подключить библиотеку `<hash_map>`. `hash_map` также может находиться в пространстве имен `stdext`.

Для нашего примера мы можем создать такой хеш-словарь: `hash_map <int, string> hm;`

Методы у `hash_map` очень похожи на методы `hash_set`. Поэтому просто приведем прмер:

```
hash_map <int, string> hm;
hash_map <int, string>::iterator it;
hm.insert(pair <int, string> (204883678, "gu"));
hm.insert(pair <int, string> (666666666, "unknown"));
it = hm.find(666666666);
printf("%s\n", it->second.c_str());
```

У `hash_map` есть приятная и удобная дополнительная функциональность, а именно оператор `[]` (оператор доступа по индексу). Его использование может выглядеть следующим образом:

```
int i = 204883678;
printf("%s", hm[i].c_str());
```

Добавлять элементы в хеш-словарь также можно с помощью оператора `[]`. Т.е. например, мы можем написать следующую конструкцию:

`hm[123456789] = "somebody";` и этот элемент будет добавлен в словарь.

13 Алгоритмы в STL

В библиотеке `<algorithm>` представлен достаточно большой (и для нас зачастую бесполезный) набор алгоритмов. Все алгоритмы работают с итераторами на контейнерах, для корректной работы необходимо использование пространства имен `std`. Нам, в первую очередь, интересны алгоритмы над контейнером `vector` (в этом разделе будем пользоваться `vector <int> v`). На самом деле, мы можем пользоваться обычным массивом, например `int v[100]`. Тогда вместо `v.begin()` следует подставлять `v`, а вместо `v.end()` — `v+n`, где `n` — количество элементов в массиве (не размер массива, а количество осмысленных элементов). Это возможно благодаря тому, что обычный указатель может интерпретироваться как итератор.

При использовании алгоритмов **STL** на векторе следует, по возможности, избегать лишних накладных расходов, связанных с выделениями памяти. В олимпиадных задачах почти всегда либо известно максимально возможное количество элементов (тогда мы можем конструировать вектор константного размера в начале программы), либо количество элементов задается во входных данных (тогда мы можем конструировать вектор после прочтения переменной, задающей количество элементов). В таком случае у нас не будет использоваться лишняя память, а накладные расходы будут очень малы (сравнимы с выделением памяти с помощью `malloc`).

Мы будем рассматривать не все алгоритмы и далеко не все способы работы с ними. В конце этой части будет приведен список других более или менее полезных алгоритмов. Все многообразие алгоритмов и способов работы с ними описано в разделах помощи среды разработки или в `man`. Таким образом, нам достаточно лишь понимать смысл и помнить (хотя бы примерно) название алгоритма.

Начнем с алгоритмов сортировки. В **STL** представлено два алгоритма сортировки: `sort` — осуществляющий быструю сортировку и `stable_sort`, который сохраняет относительный порядок следования одинаковых элементов.

Синтаксис вызова функции сортировки будет такой: `sort(v.begin(), v.end());`

Если данные необходимо отсортировать в обратном порядке, то следует использовать обратный итератор: `sort(v.rbegin(), v.rend());`

Можно сортировать не только массивы целиком, но и их непрерывные части, задавая нужные итераторы (обычно при этом используются итераторы произвольного доступа).

Следующий интересный алгоритм создает кучу максимумов из массива. Его вызов выглядит так: `make_heap(v.begin(), v.end());` Фактически, это то же самое, что первая часть HeapSort (построение кучи). Сложность построения кучи $O(N \log N)$.

После того, как у нас появился heap (он будет находиться в том же самом векторе `v`), мы можем проделать с ним некоторые действия. Например, получить их этой кучи отсортированный массив. Для этого надо вызвать функцию `sort_heap(v.begin(), v.end());` Вектор `v` будет отсортирован. Это то же самое, что второй этап HeapSort.

Кроме того, мы можем добавлять элементы к вектору, являющемуся кучей (вообще говоря, для куч лучше использовать `priority_queue`). Для этого нужно добавить элемент `x` с помощью `v.push_back(x)`, а затем вызвать `push_heap(v.begin(), v.end());`

Чтобы получить значение максимального элемента, нам достаточно разыменовать итератор `v.begin()`. Для удаления максимального элемента из кучи используется функция `pop_heap(v.begin(), v.end());`

Почти все алгоритмы вызываются аналогично — указанием начального и конечного итераторов. Так, например, полезными могут оказаться следующие алгоритмы: `reverse` — записывает последовательность «задом наперед», `random_shuffle` — переставляет элементы в случайном порядке, `is_sorted` и `is_heap` проверяющие, соответственно, является ли массив отсортированным или кучей.

Для поиска порядковой статистики можно воспользоваться функцией `nth_element(v.begin(), v`
где `k` — номер искомой порядковой статистики. В среднем поиск имеет линейную сложность (он полностью аналогичен рассмотренному нами в лекции по поиску методу).

Также полезной может оказаться функция бинарного поиска в массиве. При этом массив должен быть упорядочен по невозрастанию. Она вызывается так: `binary_search(v.begin(),`
где `i` — искомый элемент.

14 Использование собственных структур

До сих пор мы использовали **STL** для элементарных типов или других сравнимых типов. Как же использовать мощь **STL** для своих типов данных (структур, массивов и т.п.)?

Программирующие на **C++** могут сделать это очень просто — создать класс, в котором определить `operator<`. Например, мы можем использовать класс `myint` для кучи минимумов:

```
class myint
{
public:
  int num;
  bool operator<(myint &other)
  {
    if (num > other.num) return true;
    return false;
  }
};
```

Будьте аккуратны, ведь теперь во всех случаях числа типа `myint` будут сравниваться «неправильно» (т.е. «меньше» будет означать, что на самом деле «не меньше»)!

Незнакомые с **C++** могут сделать это не менее просто. Для этого надо описать функцию, принимающую на вход две структуры и возвращающую `true`, если первая «меньше» второй и `false` в противном случае. После этого во всех вызовах **STL** функций следует указывать имя этой функции (без скобок и параметров).

Приведем пример программы, в которой создается набор структур, и они сортируются только по первому полю (довольно частая задача):

```
#include <algorithm>

using namespace std;

typedef struct
{
  int f, s;
} tfs;

bool ls(tfs a, tfs b)
{
  if (a.f < b.f) return true;
  else return false;
}

int main() {
  tfs b[10];
  for (int i=9; i>=0; i--) {
    b[i].f = 10-i;
```

```
    b[i].s = i*i;
  }
  sort(b, b+10, ls);
  return 0;
}
```

Точно так же, переопределяя функцию сравнения, можно добиться построения кучи минимумов, сортировки масива в обратном порядке и т.п.

В конце лекции приведем ссылку на хорошую документацию по **STL**:

`http://www.sgi.com/tech/stl/table_of_contents.html`

15 Пример решения задачи с использованием STL

Рассмотрим задачу с Московской олимпиады 2006 года.

Тупики

На вокзале есть K тупиков, куда прибывают электрички. Этот вокзал является их конечной станцией, поэтому электрички, прибыв, некоторое время стоят на вокзале, а потом отправляются в новый рейс (в ту сторону, откуда прибыли).

Дано расписание движения электричек, в котором для каждой электрички указано время ее прибытия, а также время отправления в следующий рейс. Электрички в расписании упорядочены по времени прибытия. Поскольку вокзал — конечная станция, то электричка может стоять на нем довольно долго, в частности, электричка, которая прибывает раньше другой, отправляться обратно может значительно позднее.

Тупики пронумерованы числами от 1 до K. Когда электричка прибывает, ее ставят в свободный тупик с минимальным номером. При этом если электричка из какого-то тупика отправилась в момент времени X, то электричку, которая прибывает в момент времени X, в этот тупик ставить нельзя, а электричку, прибывающую в момент $X+1$ — можно.

Напишите программу, которая по данному расписанию для каждой электрички определит номер тупика, куда прибудет эта электричка.

Формат входных данных

Сначала вводятся число K — количество тупиков и число N — количество электропоездов ($1 \leqslant K \leqslant 500000, 1 \leqslant N \leqslant 500000$). Далее следуют N строк, в каждой из которых записано по 2 числа: время прибытия и время отправления электрички. Время задается натуральным числом, не превышающим 10^9. Никакие две электрички не прибывают в одно и то же время, но при этом несколько электричек могут отправляться в одно и то же время. Также возможно, что какая-нибудь электричка (или даже несколько) отправляются в момент прибытия какой-нибудь другой электрички. Время отправления каждой электрички строго больше времени ее прибытия.

Все электрички упорядочены по времени прибытия. Считается, что в нулевой момент времени все тупики на вокзале свободны.

Формат выходных данных

Выведите N чисел — по одному для каждой электрички: номер тупика, куда прибудет соответствующая электричка. Если тупиков не достаточно для того, чтобы организовать движение электричек согласно расписанию, выведите два числа: первое должно равняться 0 (нулю), а второе содержать номер первой из электричек, которая не сможет прибыть на вокзал.

Примеры

Входные данные	Выходные данные
1 1 2 5	1
1 2 2 5 5 6	0 2
2 3 1 3 2 6 4 5	1 2 1

Приведем разбор задачи, который облегчит понимание решения:

Нам необходимо упорядочить события (приезды и отъезды электричек). Сортировать необходимо по времени события (ключ, по которому строится куча); номеру электрички, с которой произошло событие; и признаку события (приезд или отъезд). В случае, если приезд и отъезд случаются одновременно, сначала должны стоять отъезжающие электрички (в этой задаче удобно прибавить ко времени отъезда единицу).

Кроме того, заведем кучу (минимумов) свободных тупиков. Сначала добавим туда все тупики.

Затем пойдем по событиям. Если событие — приезд электрички, то необходимо взять из кучи свободных тупиков минимальный тупик и сохранить для этой электрички номер тупика, в который она встанет (это удобно делать в массиве, где индекс равен номеру электрички). Если же куча свободных тупиков пуста, то сразу выводим, что расставить электрички нельзя.

Если событие — отъезд электрички, то смотрим, какой тупик она занимала и добавляем этот тупик в кучу свободных тупиков.

Для вывода ответа необходимо просто вывести содержимое массива, в котором сохранялись номера тупиков при приезде электрички.

Решение этой задачи с использованием STL может выглядеть так:

```
#include <stdio.h>
#include <queue>
#include <algorithm>

typedef struct {
  int num, time, etype;
} rail_event;

using namespace std;

#define MAXN 500010

bool ls (rail_event a, rail_event b)
{
  if (a.time < b.time)
    return true;
  else if ((a.time == b.time) && (a.etype < b.etype))
```

```
    return true;
  else return false;
}

int answer[MAXN];
priority_queue <int, vector <int>, greater <int> > bays;
vector <rail_event> events;

int main()
{
  int n, k, i, j, arrival, departure;
  rail_event revent;
  scanf("%d%d", &k, &n);
  for (i=1; i<=k; i++)
    bays.push(i); // все тупики свободны
  for (i=1; i<=n; i++) {
    scanf("%d%d", &arrival, &departure);
    revent.num = i;
    revent.time = arrival;
    revent.etype = 1;
    events.push_back(revent);
    revent.num = i;
    revent.time = departure + 1;
    revent.etype = 0;
    events.push_back(revent);
  }
  sort(events.begin(), events.end(), ls);
  for (i=0; i<events.size(); i++) {
    revent = events[i];
    if (revent.etype == 1) { // прибытие
      if (bays.empty()) {
        printf("0 %d", revent.num);
        return 0;
      } else {
        answer[revent.num] = bays.top();
        bays.pop();
      }
    } else  // отбытие
      bays.push(answer[revent.num]);
  }
  for (i=1; i<=n; i++)
    printf("%d\n", answer[i]);
  return 0;
}
```