**Требования к программам**

1. В программе должны быть реализованы следующие структуры данных:

   - Enum class, задающий условия для полей:

```
# ifndef condition_H
# define condition_H

enum class condition
{
  none,  // not specified
  eq,    // equal
  ne,    // not equal
  lt,    // less than
  gt,    // less than
  le,    // less equal
  ge,    // great equal
  like,  // strings only: match pattern
  nlike, // strings only: not match pattern
};
# endif
```

   - Enum class, задающий условия для вывода полей:

```
# ifndef ordering_H
# define ordering_H

enum class ordering
{
  none,  // not specified
  name,  // print name
  phone, // print phone
  group, // print group
};

# endif
```

   - Контейнер данных объектов типа `record`:

```
# ifndef record_H
# define record_H
# include <memory>
# include <stdio.h>
# include "condition.h"
# include "ordering.h"

enum class read_status
{
  success = 0,
  eof = -1,
  format = -2,
  memory = -3,
};
```

```
class record
{
  private:
    std::unique_ptr<char []> name = nullptr;
    int phone = 0;
    int group = 0;
  public:
    record () = default;
   ~record () = default;
    char  * get_name () const { return name.get (); }
    int get_phone () const { return phone; }
    int get_group () const { return group; }
    int init (const char *n, int p, int g);
    // Allow as return value for functions
    record (record &&x) = default; // move constructor
    // Assignement move operator
    record& operator= (record&& x) = default;
    // Prohoibit pass by value
    // (it is default when move constructor is declared)
    record (const record &x) = delete;
    // Prohoibit assignement operator
    // (it is default when move constructor is declared)
    record& operator= (const record&) = delete;
    // Check condition 'x' for field 'name' for 'this' and 'y'
    bool compare_name (condition x, const record& y) const;
    // Check condition 'x' for field 'phone' for 'this' and 'y'
    bool compare_phone (condition x, const record& y) const;
    // Check condition 'x' for field 'group' for 'this' and 'y'
    bool compare_group (condition x, const record& y) const;
    void print (const ordering order[] = nullptr, FILE * fp = stdout);
    read_status read (FILE *fp = stdin);
};
# endif
```

Функции сравнения в этом классе сравнивают одно из полей класса с соответствующим полем класса `y` согласно условию, заданному аргументом `x`.

- Enum class, задающий логические операции для полей:

```
# ifndef operation_H
# define operation_H

enum class operation
{
  none,  // not specified
  land,  // logical and
  lor,   // logical or
};
# endif
```

- Класс, задающий условие для проверки:

```
# ifndef command_H
```

```
# define command_H

# include <stdio.h>
# include "record.h"

class command : public record
{
  private:
    condition c_name  = condition::none;
    condition c_phone = condition::none;
    condition c_group = condition::none;
    operation op = operation::none;
  public:
    command () = default;
   ~command () = default;
    // Convert string command to data structure
    // Example: "phone = 1234567 and name like St% and group <> 208"
    // parsed to
    // command::name  = "St%",   command::c_name  = condition::like
    // command::phone = 1234567, command::c_phone = condition::eq
    // command::group = 208,     command::c_group = condition::ne
    // command::op = operation::land
    // other fields are unspecified
    bool parse (const char * string);
    // Print parsed structure
    void print (FILE *fp = stdout) const;
    // Apply command, return comparision result for record 'x'
    bool apply (const record& x) const;
};

# endif
```

2. Пример реализации некоторых функций из класса `record`:

```
# include <string.h>
# include <stdio.h>
# include "record.h"

# define LEN 1234
using namespace std;

int record::init (const char *n, int p, int g)
{
  phone = p;
  group = g;
  if (n)
    {
      name = std::make_unique<char []> (strlen (n) + 1);
      if (!name) return -1;
      strcpy (name.get(), n);
    }
  else
```

```
    name = nullptr;
  return 0;
}

read_status record::read (FILE *fp)
{
  char buf[LEN];
  name = nullptr;
  if (fscanf (fp, "%s%d%d", buf, &phone, &group) != 3)
    {
      if (feof(fp)) return read_status::eof;
      return read_status::format;
    }
  if (init (buf, phone, group))
    return read_status::memory;
  return read_status::success;
}

void record::print (const ordering order[], FILE *fp)
{
  const int max_items = 3;
  const ordering default_ordering[max_items]
    = {ordering::name, ordering::phone, ordering::group};
  const ordering * p = (order ? order : default_ordering);

  for (int i = 0; i < max_items; i++)
    switch (p[i])
      {
        case ordering::name:
          printf (" %s", name.get()); break;
        case ordering::phone:
          printf (" %d", phone); break;
        case ordering::group:
          printf (" %d", group); break;
        case ordering::none:
          continue;
      }
  fprintf (fp, "\n");
}

// Check condition 'x' for field 'phone' for 'this' and 'y'
bool record::compare_phone (condition x, const record& y) const
{
  switch (x)
  {
    case condition::none: // not specified
      return true; // unspecified opeation is true
    case condition::eq:   // equal
      return phone == y.phone;
    case condition::ne:   // not equal
      return phone != y.phone;
```

```
    case condition::lt:   // less than
      return phone < y.phone;
    case condition::gt:   // less than
      return phone > y.phone;
    case condition::le:   // less equal
      return phone <= y.phone;
    case condition::ge:   // great equal
      return phone >= y.phone;
    case condition::like: // strings only: match pattern
      return false; // cannot be used for phone
  }
  return false;
}
```

3. **Задача программы**:
   - Построить **двунаправленный список** объектов типа `record` и считать его из указанного файла (аргумент командной строки)
   - Считывать команды поиска в этом списке **по одной со стандартного ввода (stdin)**, до тех пор пока команды не закончатся
   - Применять команду поиска к списку и выводить **только найденные (т.е. удовлетворяющие условию)** элементы в **стандартный вывод (stdout)**
4. Все команды поиска имеют следующий вид:
   - <условия на выводимые поля> where <условие поиска>
5. <условия на выводимые поля> имеют вид:
   - <список полей> – выводить указанные поля в указанном порядке, список состоит из разделенных запятыми имен полей без повторений; например, group, name – выводить только поле поле group и поле name (в этом порядке);
   - * – выводить все поля, эквивалентен name, phone, group
6. <условие поиска> имеют вид:
   - <условие поиска на одно поле> – задает **одно условие на одно поле** записи `record`.
   - <условие поиска на одно поле 1> and <условие поиска на одно поле 2>
   - <условие поиска на одно поле 1> or <условие поиска на одно поле 2>
   - <условие поиска на одно поле 1> and <условие поиска на одно поле 2> and <условие поиска на одно поле 3>
   - <условие поиска на одно поле 1> or <условие поиска на одно поле 2> or <условие поиска на одно поле 3>

   Если в <условие поиска> участвует более одного условия на поля записи `record`, то они **задают условия на разные поля записи** `record`.
7. <условие поиска на одно поле> записи `record` имеет вид:
   - <поле> <оператор> <выражение>, где
     - <поле> – имя поля (name, phone, value)

- <оператор> – логический оператор отношения: = – равно, <> – не равно, <, >, <=, >= – соответствуют языку C
- <выражение> – константное выражение соответствующего типа

- <поле> like <образец> где
  - <поле> – имя поля символьного типа (т.е. только name)
  - <образец> – образец поиска. Может включать в себя специальные символы:
    * '_' – соответствует 1 любому символу, а символы "_" и "\\" соответствуют литеральным символам "_" и "\"
    * '%' – соответствует 0 или более любым символам, а символы "\%" и "\\" соответствуют литеральным символам "%" и "\"
    * '[$n$–$m$]' ($n$, $m$ – символы) – соответствует 1 любому символу, имеющему код в диапазоне $n \ldots m$, а символы "\[", "\]" и "\\" соответствуют литеральным символам "[", "]" и "\"
    * '[^$n$–$m$]' ($n$, $m$ – символы) соответствует любому символу, имеющему код, не содержащийся в диапазоне $n \ldots m$, а символы "\[", "\]", "\^" и "\\" соответствуют литеральным символам "[", "]", "^" и "\"

  Условие выполнено, если <поле> соответствует образцу поиска.

- <поле> not like <образец> где
  - <поле> – имя поля символьного типа (т.е. только name)
  - <образец> – образец поиска.

  Условие выполнено, если <поле> не соответствует образцу поиска.

8. Примеры команд поиска:

   - group, name where phone = 1234567 and name = Student – вывести поля group и name для всех элементов списка, у которых поле phone равно 1234567 и поле name равно "Student".
   - * where phone >= 1234567 and name like St% – вывести все поля для всех элементов списка, у которых поле phone больше или равно 1234567 и поле name соответствует образцу поиска "St%".
   - name, phone where group = 208 and phone <> 1234567 – вывести поля name и phone для всех элементов списка, у которых поле group равно 208 и поле phone не равно 1234567.
   - * where name = Student or phone = 1234567 – вывести все поля для всех элементов списка, у которых поле name равно "Student" или поле phone равно 1234567.
   - name where name not like St% and phone = 1234567 and group = 208 – вывести поле name для всех элементов списка, у которых поле name не соответствует образцу поиска "St% поле phone равно 1234567 и поле group равно 208.

9. Программа должна получать все параметры в качестве аргументов командной строки и стандартного ввода. Аргументы командной строки:

   1) `filename` – имя файла, откуда надо прочитать список.

   Например, запуск

   ```
   cat commands.txt | ./a.out a.txt > result.txt
   ```

   означает, что файл `commands.txt` подается на стандартный ввод, список надо прочитать из файла `a.txt`, а результаты будут перенаправлены со стандартного вывода в файл `result.txt`.

10. Класс "список" должен содержать функцию ввода списка из указанного файла.

11. Ввод списка из файла. В указанном файле находится дерево в формате:

| | | |
|---|---|---|
| Слово-1 | Целое-число-1 | Целое-число-2 |
| Слово-2 | Целое-число-3 | Целое-число-4 |
| ... | ... | |
| Слово-n | Целое-число-2n-1 | Целое-число-2n |

    где слово – последовательность алфавитно-цифровых символов без пробелов. Длина слова неизвестна, память под него выделяется динамически. Все записи в файле различны (т.е. нет двух, у которых совпадают все 3 поля). Концом ввода считается конец файла. Программа должна выводить сообщение об ошибке, если указанный файл не может быть прочитан или содержит данные неверного формата.

12. Вывод результата работы функции в функции `main` должен производиться по формату:

```
printf ("%s : Result = %d Elapsed = %.2f\n",
        argv[0], res, t);
```

    где

    - `argv[0]` – первый аргумент командной строки (имя образа программы),
    - `res` – общее количество найденных элементов списка,
    - `t` – время работы на все команды.

    **Вывод должен производиться в точности в таком формате**, чтобы можно было автоматизировать обработку запуска многих тестов.

### Задачи

Требуется написать программу, которая

- строит **двунаправленный список** объектов типа `record` и считывает его из указанного файла (аргумент командной строки)
- считывает команды поиска в этом списке **по одной со стандартного ввода (stdin)**, до тех пор пока команды не закончатся
- для каждой прочитанной команды применяет эту команду поиска к списку и выводит **только найденные (т.е. удовлетворяющие условию)** элементы в **стандартный вывод (stdout)**.

**Задание оценивается по качеству реализованной системы классов и успешности реализации следующих элементов:**

1. Поддержать условие вида <поле> not like <образец> со всеми специальными символами
2. Поддержать задание <условия на выводимые поля>
3. Поддержать задание условия <and>
4. Поддержать задание условия <or>