Project Report - Simple ADIF Database

1 Description

In this project, we need to implement a simple database by C++. It can support importing or exporting ADIF or CSV files, searching records that satisfy the specific conditions and removing some records.

2 Idea

We represent the database by a class. Here we maintain 2 attributes, the set of all records stored in the database and the set of all field names. Since our executable file may be launched multiple times, we need data persistence, which means we need to store data onto the disk when we import new files or load data from the disk when we do searchs or modify records.

3 Working Process of the Program

We start running in `main.cpp` which is for reading the command line arguments and deciding the operation to do. Then we get into the correspounding member function of the class `ADIFDatabase` implemented by `defs.cpp`.

- Import ADIF/CSV files(in `ModifyADIF`/`ModifyCSV`)
  First we open the file, then put them in the memory of the program(in `ReadBinaryFile`). Then we load the data of the previous database from the disk(in `Load`). After that, we begin to handle the file through format given by the problem. Details are covered in the 4-th part, so here we skip it. After reading data from the file, we combine it with the database then store it back to the disk(in `Store`).
- Export ADIF/CSV files(in `ExportADIF`/`ExportCSV`)
  First we open the file that we will export data in, then we load the data of the previous database from the disk(in `Load`). Then we need to sort the set of all field names that they satisfy the requirement given by the problem that the fields are sorted by field name dictionary order in ascending order. Besides we also need to sort the set of all records by the primary keys. After that, we traverse and export all records.
  The only difference between ADIF and CSV files is their format. For ADIF every item is like `<fieldname:num>content` and there is a `<EOR>` in the end of a record. For CSV we need to export all field names first then each line is for one record. If there is some fields that the current record missing in the traversing, ADIF format will ignore such field while CSV format will export an empty string.
- Search for records that satisfy specific conditions(in `SearchCall`/`SearchTime`)
  First we load the data of the previous database from the disk(in `Load`). Then we traverse all records and pick records which meet the standard out. For the result set, we need to sort them and output them in the shell. At last we need to output a string, indicating how many records satisfy the condition.
  The only difference in `-s <call>` and `-l <start time> <end time>` is the condition we filters records.
- Delete records(also in `ModifyADIF`/`ModifyCSV`)
  Actullay, the process of deleting is almost the same as importing. The only difference is that when we deal with a record from the file, for importing we need to add it into the set of all records while for deleting we nned to remove it. Besides they share the same process as a result we implement them in the same functions.

4 Source Codes

4.1 `Load`

In the class of `ADIFDatabase`, we have two `vector`. `field` is for storing all field names and `record` is for storing all records in the database.

Since we use STL `vector` and `map` to store data and we need to maintain data persistence, we use a third-party library `cereal` to serialize STL data. The related files of `cereal` are in the directory `cereal`.

The database is stored in the file `out.cereal`. If we want to load data from it, just need to use a input stream and read data from it. In the code below some statements are deriving from the usage of the library `cereal`. Besides, as an external class, `cereal` needs to be friend element of the class `ADIFDatabase` so that it can visit `field` and `record`.

It's worth noting that in the beginning the file `out.cereal` will be nonexist or empty, so we need a special judgement.

```
void Load()
{
    std::ifstream in;
    {
        in.open("out.cereal",std::ios::in);
        if (!in.is_open()) return;
        if (in.eof()) return;
        cereal::BinaryInputArchive archive(in);
        archive(db);
    }
    in.close();
}
```

The code of `Store` is almost the same as `Load`, just replacing input stream with output stream. So we skip it.

4.2 `ModifyADIF`

First we use `ReadBinaryFile` to open the file and get data, then we use `Load` to load data. Then we search for `<EOH>` since the content before `<EOH>` should not be taken consideration into.
For each record we first search its end `<EOR>`, then we search its head structure `<fieldname:number>content`, extracting field name and field content. If we meet a new field name, we need to add it to `field`. After that, we compose a record with contect extracted and add it to `record` or remove it from `record`.
Here, `flag=false` indicates that we are importing ADIF files so we need to add while `flag=true` indicates that we are deleting items so we need to remove.
At last we use `Store` to store data back to the disk.

```
void ADIFDatabase::ModifyADIF(char *filename, bool flag)
{
    char *data, *tmp;
    int size = ReadBinaryFile(filename, &data);
    char *ed = data + size;
    if (size < 0) return;
    tmp = find(data, "<EOH>", 5);
    if (!tmp) return;
    tmp += 5;        // 跳过 <EOH>
    int num;
    Load();
    while (tmp && tmp + 5 < ed && *tmp != '\0') {
        char *st = tmp;         // 记录当前记录的开头
        tmp = find(tmp, "<EOR>", 5);        // 寻找当前记录的结尾
        map<string, string>a;
        for (char *p = st; p <= tmp; p++) {
            if (*p == '<') {
                char *pp = find(p, ":", 1);
                char *ppp = find(p, ">", 1);
                if (pp > tmp || ppp > tmp) break;
                string name = GetName(p+1, pp-1);
                num = GetNum(pp+1, ppp-1);
                if (!InField(field, name)) {      // 之前没有这个 field, 现在需要加
入
                    field.push_back(name);
                }
                string now = "";
                now.assign(ppp+1, num);
                if (name == "TIME_ON" && num == 4) now += "00";
                a[name] = now;
                p = ppp;
            }
        }
        ModifyRecord(a, flag);
        tmp += 5;
    }
    Store();
}
```

4.3 `ModifyCSV`

First we use `ReadBinaryFile` to open the file and get data, then we use `Load` to load data.

There is a bit difference between CSV and ADIF files. For CSV, we need first read all field names, storing them in a local vector `a`, updating `field`.

```
    vector<string>a;
    for (int i = 0; i < 7; i++) {
        if (i != 6)
            tmp = find(data, ",", 1);
        else {
            char *p = data;
            while (*p != '\r' && *p != '\n') p++;
            const char ch = *p;
            tmp = find(data, &ch, 1);
        }
        string name = GetName(data, tmp-1);
        if (!InField(field, name)) field.push_back(name);
        a.push_back(name);
        data = tmp + 1;
    }
```

Then we traverse all lines, for each line we read all fields of a record by commas which are used for separation. It's worth noting that there may be comma in the quotations, so we need first search the quotation, if not then we can search for the next comma.

```
while (data < ed) {
        while (*data == '\r' || *data == '\n') data++;
        if (data >= ed) break;
        char *p = data;
        map<string, string>t;
        while (*p != '\r' && *p != '\n' && p < ed) p++;
        for (int i = 0 ; i < 7; i++) {
            string name = "";
            if (*data == '\"') {
                tmp = find(data+1, "\"", 1);
                name.assign(data+1, tmp-data-1);
                data = tmp + 2;
            }
            else {
                const char ch = *p;
                if (i == 6) tmp = find(data, &ch, 1);       // 行末没有逗号
                else tmp = find(data, ",", 1);
                name.assign(data, tmp-data);
                data = tmp + 1;
            }
            if (a[i] == "TIME_ON" && name.length() == 4) name += "00";
            t[a[i]] = name;
        }
        data = p;
        ModifyRecord(t, flag);
    }
    Store();
```

Here, `flag=false` indicates that we are importing ADIF files so we need to add while `flag=true` indicates that we are deleting items so we need to remove.
At last we use `Store` to store data back to the disk.

4.4 `ExportADIF`

Here we first load data from the disk. Then sort `field` and `record` so that the fields are sorted by field name dictionary order in ascending order and records are sorted by primary key dictionary order descending order between records.
. For primary key `QSO_DATE` and `TIME_ON` we put them into sorting together, but when we traverse all records we first export such 2 fields and them export other fields to achieve the demand of the problem.
Here `GetADIFString` will turn a field name and field into a ADIF item, which has the format `<field_name:number>content`. For each record we need to output `<EOR>` at last.

```
void ADIFDatabase::ExportADIF(char *filename)
{
    FILE *fp = fopen(filename, "wb");
    string s;
    Load();
    sort(field.begin(), field.end());// 升序排
    sort(record.begin(), record.end(), cmp);
    for (auto p : record) {
        s = GetADIFString("QSO_DATE", p["QSO_DATE"]);
        fwrite(s.c_str(), strlen(s.c_str()), 1, fp);
        s = GetADIFString("TIME_ON", p["TIME_ON"]);
        fwrite(s.c_str(), strlen(s.c_str()), 1, fp);
        for (auto tmp : field) {
            if (tmp == "QSO_DATE" || tmp == "TIME_ON") continue;
            if (p.count(tmp)) {
                s = GetADIFString(tmp, p[tmp]);
                fwrite(s.c_str(), strlen(s.c_str()), 1, fp);
            }
        }
        s = "<EOR>\n";
        fwrite(s.c_str(), strlen(s.c_str()), 1, fp);
    }
    fclose(fp);
}
```

4.5 `ExportCSV`

The loading and sorting is the same as `ExportADIF` so we skip it. Here we need output field name first. However there may be some field names that there have been no record has such this field due to deletions but we still keep it, so we need first find them and delete it from `field`. The detailed implementation is traversing all field names and checking whether there is a record related. If so we put it into `new_field`. Finishing traversing, we assign `new_field` to `field`.

After updating, we output all field names, and traverse all records, output the correspounding field content. If the record does not have this field, just output empty string.

```
void ADIFDatabase::ExportCSV(char *filename)
{
    FILE *fp = fopen(filename, "wb");
    string s;
    Load();
    sort(field.begin(), field.end());// 升序排
    sort(record.begin(), record.end(), cmp);
    s = "QSO_DATE,";
    fwrite(s.c_str(), strlen(s.c_str()), 1, fp);
    s = "TIME_ON";
    fwrite(s.c_str(), strlen(s.c_str()), 1, fp);
    vector<string>new_field;        // 可能有些属性 因为删除操作，表中不再有记录有这个
属性，那我们在这里将属性删掉。
    for (auto tmp : field) {
        bool bj = false;
        for (auto p : record)
            if (p.count(tmp)) {
                bj = true;
                break;
            }
        if (bj) new_field.push_back(tmp);
    }
    field = new_field;
    for (auto tmp : field) {
        if (tmp == "QSO_DATE" || tmp == "TIME_ON") continue;
        s = "," + tmp;
        fwrite(s.c_str(), strlen(s.c_str()), 1, fp);
    }
    s = "\n";
    fwrite(s.c_str(), strlen(s.c_str()), 1, fp);
    for (auto p : record) {
        s = p["QSO_DATE"] + ",";
        fwrite(s.c_str(), strlen(s.c_str()), 1, fp);
        s = p["TIME_ON"];
        fwrite(s.c_str(), strlen(s.c_str()), 1, fp);
        for (auto tmp : field) {
            if (tmp == "QSO_DATE" || tmp == "TIME_ON") continue;
            if (p.count(tmp)) {
                if (p[tmp].find(",") == string::npos)
                    s = "," + p[tmp];
                else s = ",\"" + p[tmp] + "\"";
            }
            else s = ",";
            fwrite(s.c_str(), strlen(s.c_str()), 1, fp);
        }
        s = "\n";
        fwrite(s.c_str(), strlen(s.c_str()), 1, fp);
    }
    fclose(fp);
}
```

4.6 `SeachCall` / `SeachTime`

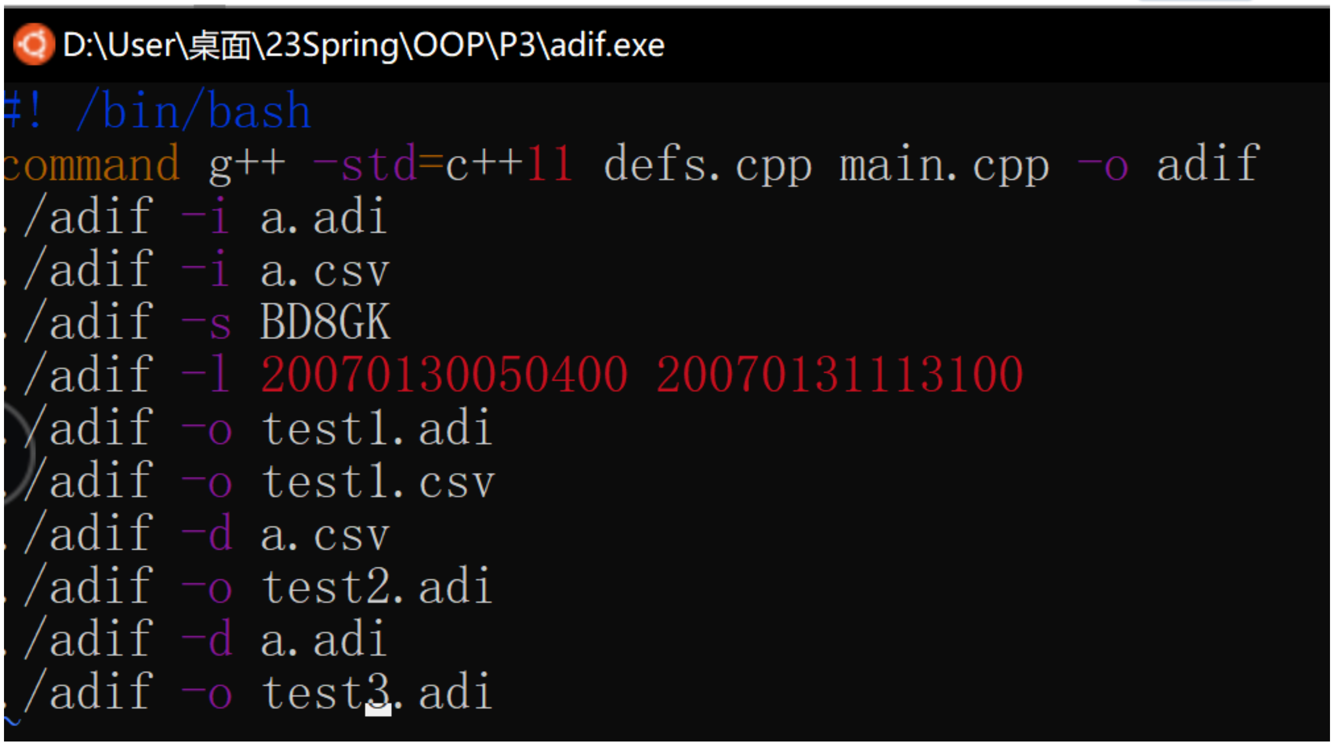
First we load data, then traverse all records. If the record satisfy the condition we put it into the vector `a`. After that we sort the vector so that records are sorted by primary key dictionary order descending order between records. Then we output the result records and their fields in the givin format. At last print `x record(s) found` where x is the size of the result set.

```
void ADIFDatabase::SearchCall(string s)
{
    Load();
    vector<map<string, string>>a;
    for (auto p : record) {
        if (!p.count("CALL")) continue;
        if (p["CALL"] != s) continue;
        a.push_back(p);
    }
    sort(a.begin(), a.end(), cmp);
    for (auto p : a) {
        cout << "QSO_DATE: " << p["QSO_DATE"] << ", TIME_ON: " << p["TIME_ON"];
        for (auto tmp : p) {
            if (tmp.first == "QSO_DATE" || tmp.first == "TIME_ON") continue;
            cout << ", "<< tmp.first << ": " << tmp.second;
        }
        cout << endl;
    }
    cout << a.size() << " record(s) found" << endl;
}
```

`SearchTime` is almost the same except the condition.

5 Test Results

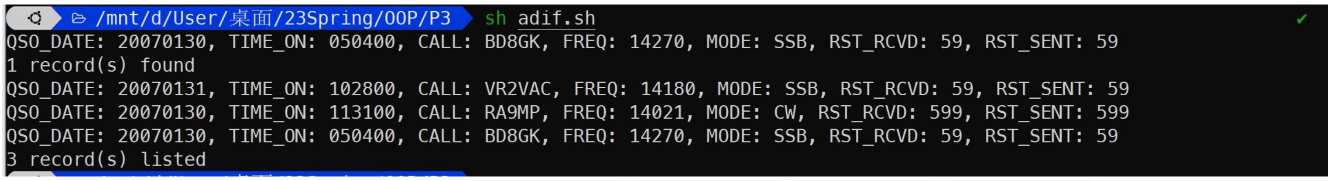
We use the shell scipt that compiles our program and do some testing. The content of the shell script is below.(`adif.sh`)



```
D:\User\桌面\23Spring\OOP\P3\adif.exe
#! /bin/bash
command g++ -std=c++11 defs.cpp main.cpp -o adif
./adif -i a.adi
./adif -i a.csv
./adif -s BD8GK
./adif -l 20070130050400 20070131113100
./adif -o test1.adi
./adif -o test1.csv
./adif -d a.csv
./adif -o test2.adi
./adif -d a.adi
./adif -o test3.adi
```

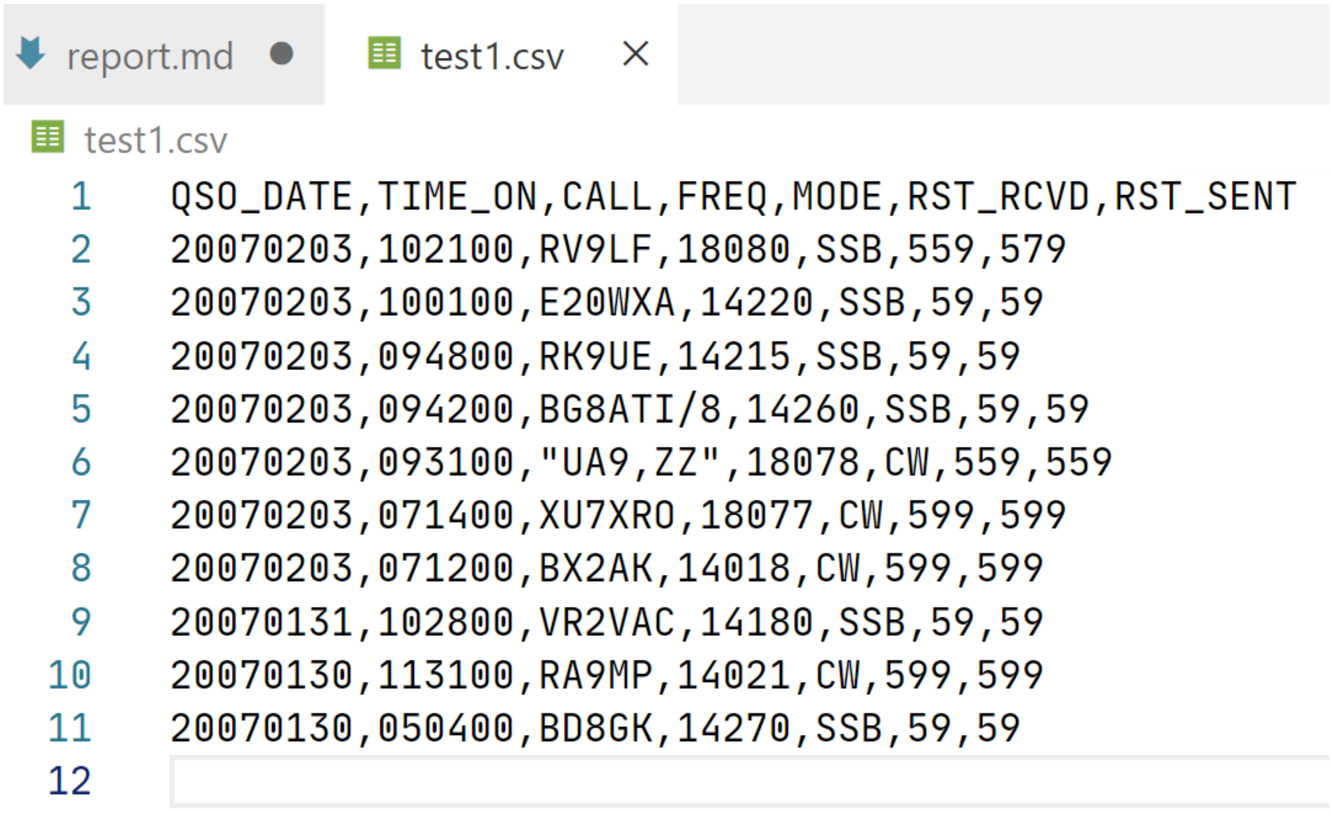
The input files(`a.adi` `a.csv`) are in the first zip file.

The output of the shell is below.



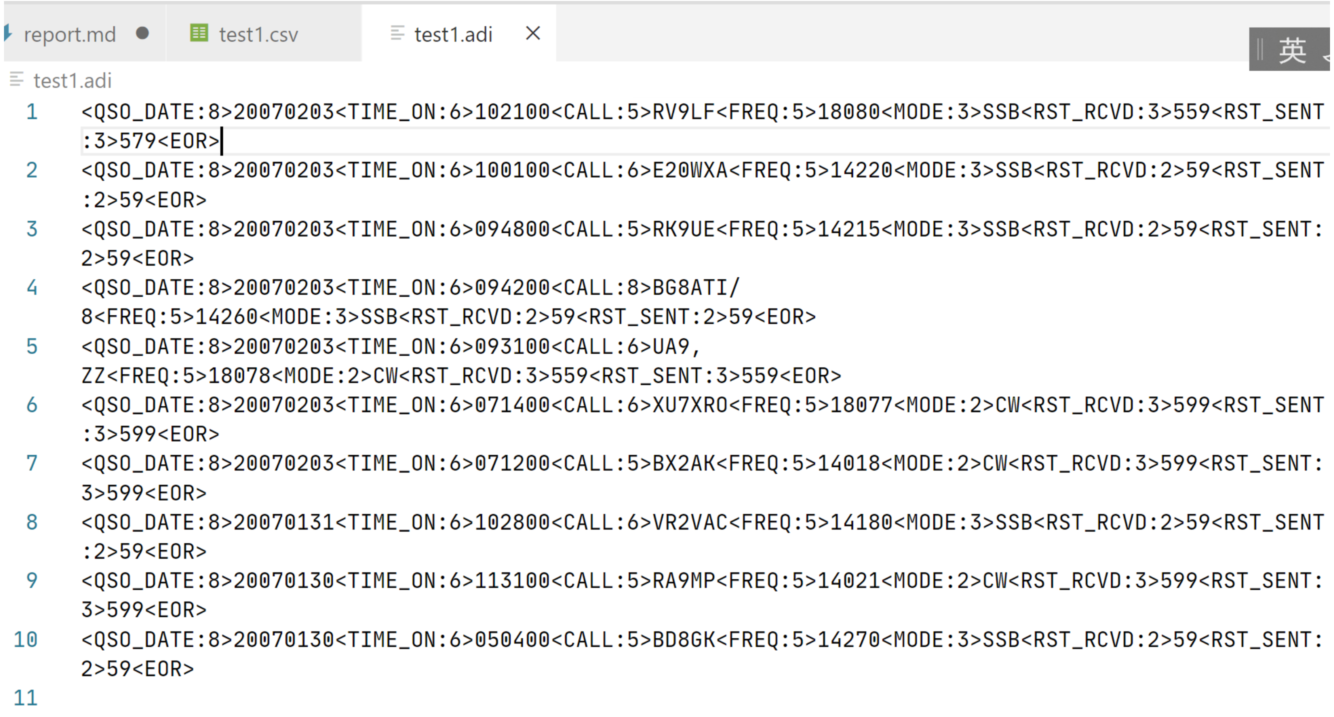
```
/mnt/d/User/桌面/23Spring/OOP/P3  sh adif.sh
QSO_DATE: 20070130, TIME_ON: 050400, CALL: BD8GK, FREQ: 14270, MODE: SSB, RST_RCVD: 59, RST_SENT: 59
1 record(s) found
QSO_DATE: 20070131, TIME_ON: 102800, CALL: VR2VAC, FREQ: 14180, MODE: SSB, RST_RCVD: 59, RST_SENT: 59
QSO_DATE: 20070130, TIME_ON: 113100, CALL: RA9MP, FREQ: 14021, MODE: CW, RST_RCVD: 599, RST_SENT: 599
QSO_DATE: 20070130, TIME_ON: 050400, CALL: BD8GK, FREQ: 14270, MODE: SSB, RST_RCVD: 59, RST_SENT: 59
3 record(s) listed
```

And we can see the result files that we export(`test1.adi` `test1.csv` `test2.adi` `test3.adi`). You can see them in the directory of `test_output` of the first zip file.



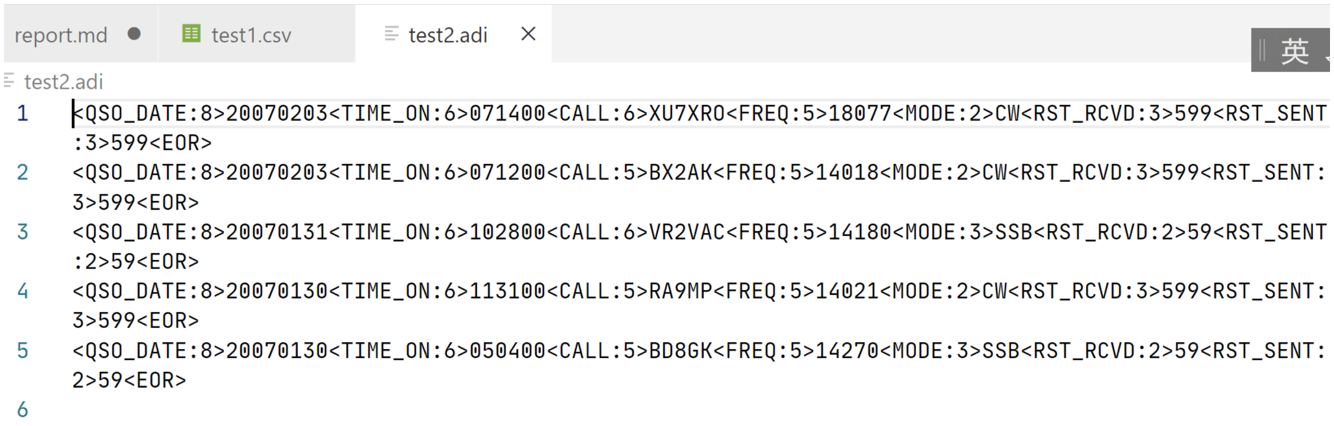
```
QSO_DATE,TIME_ON,CALL,FREQ,MODE,RST_RCVD,RST_SENT
20070203,102100,RV9LF,18080,SSB,559,579
20070203,100100,E20WXA,14220,SSB,59,59
20070203,094800,RK9UE,14215,SSB,59,59
20070203,094200,BG8ATI/8,14260,SSB,59,59
20070203,093100,"UA9,ZZ",18078,CW,559,559
20070203,071400,XU7XRO,18077,CW,599,599
20070203,071200,BX2AK,14018,CW,599,599
20070131,102800,VR2VAC,14180,SSB,59,59
20070130,113100,RA9MP,14021,CW,599,599
20070130,050400,BD8GK,14270,SSB,59,59

```

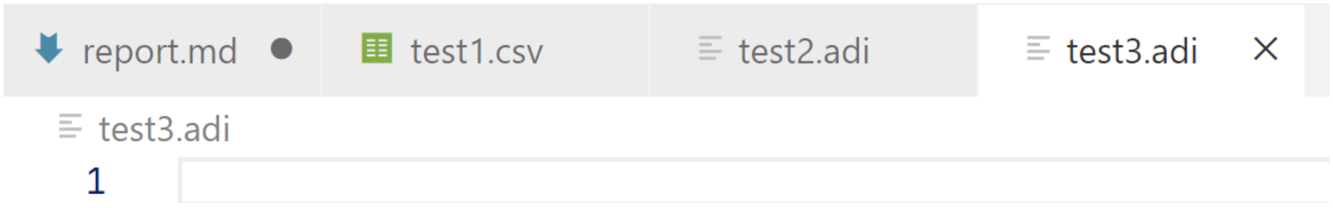


```
<QSO_DATE:8>20070203<TIME_ON:6>102100<CALL:5>RV9LF<FREQ:5>18080<MODE:3>SSB<RST_RCVD:3>559<RST_SENT:3>579<EOR>
<QSO_DATE:8>20070203<TIME_ON:6>100100<CALL:6>E20WXA<FREQ:5>14220<MODE:3>SSB<RST_RCVD:2>59<RST_SENT:2>59<EOR>
<QSO_DATE:8>20070203<TIME_ON:6>094800<CALL:5>RK9UE<FREQ:5>14215<MODE:3>SSB<RST_RCVD:2>59<RST_SENT:2>59<EOR>
<QSO_DATE:8>20070203<TIME_ON:6>094200<CALL:8>BG8ATI/8<FREQ:5>14260<MODE:3>SSB<RST_RCVD:2>59<RST_SENT:2>59<EOR>
<QSO_DATE:8>20070203<TIME_ON:6>093100<CALL:6>UA9,ZZ<FREQ:5>18078<MODE:2>CW<RST_RCVD:3>559<RST_SENT:3>559<EOR>
<QSO_DATE:8>20070203<TIME_ON:6>071400<CALL:6>XU7XRO<FREQ:5>18077<MODE:2>CW<RST_RCVD:3>599<RST_SENT:3>599<EOR>
<QSO_DATE:8>20070203<TIME_ON:6>071200<CALL:5>BX2AK<FREQ:5>14018<MODE:2>CW<RST_RCVD:3>599<RST_SENT:3>599<EOR>
<QSO_DATE:8>20070131<TIME_ON:6>102800<CALL:6>VR2VAC<FREQ:5>14180<MODE:3>SSB<RST_RCVD:2>59<RST_SENT:2>59<EOR>
<QSO_DATE:8>20070130<TIME_ON:6>113100<CALL:5>RA9MP<FREQ:5>14021<MODE:2>CW<RST_RCVD:3>599<RST_SENT:3>599<EOR>
<QSO_DATE:8>20070130<TIME_ON:6>050400<CALL:5>BD8GK<FREQ:5>14270<MODE:3>SSB<RST_RCVD:2>59<RST_SENT:2>59<EOR>

```



```
<QSO_DATE:8>20070203<TIME_ON:6>071400<CALL:6>XU7XRO<FREQ:5>18077<MODE:2>CW<RST_RCVD:3>599<RST_SENT:3>599<EOR>
<QSO_DATE:8>20070203<TIME_ON:6>071200<CALL:5>BX2AK<FREQ:5>14018<MODE:2>CW<RST_RCVD:3>599<RST_SENT:3>599<EOR>
<QSO_DATE:8>20070131<TIME_ON:6>102800<CALL:6>VR2VAC<FREQ:5>14180<MODE:3>SSB<RST_RCVD:2>59<RST_SENT:2>59<EOR>
<QSO_DATE:8>20070130<TIME_ON:6>113100<CALL:5>RA9MP<FREQ:5>14021<MODE:2>CW<RST_RCVD:3>599<RST_SENT:3>599<EOR>
<QSO_DATE:8>20070130<TIME_ON:6>050400<CALL:5>BD8GK<FREQ:5>14270<MODE:3>SSB<RST_RCVD:2>59<RST_SENT:2>59<EOR>
```



6 Discussions

I think this project has indeed improves my ability of C++, and the harvest is fruitful.
However, the requirements of the project can be clearer and I think more tests and standard judge program can be given so that we have a better experience. Besides, the workload of the project is too heavy while its score does not match the workload, which I think is the main reason of my pain.

My implementations are so brute-force. If more time given, I think I can do some optimization, which can take advantage of knowledge in the Database course, e.g. B+ Tree.

All in all, hope we can have a better project experience.