C++ Foundation, Assignment 2

Transaction lists and claims

Write a C++ program that processes a list of resource transactions/ claims stored in a text file and answers queries selected by a user.

A transaction / claim states the amount of memory the process wants or can release Transactions are listed in chronological order *(the oldest entry is at the top).* Each transaction in a file is a line that contains a single-word identifier and an integer number separated by a whitespace character. An example input file with 11 transactions is shown below.

```
mail 100
mail 45
browser 80
editor 35
mail 20
browser -60
editor 40
mail -30
browser 85
editor 0
mail 15
```

A positive number represents a transaction where a resource is acquired, for instance `mail 45` can be interpreted as an acquisition of 45 units of memory by a mail program. A negative number represents a resource release, `browser -60` means that a browser program released 60 units of memory. Zero has a special meaning, `editor 0` indicates that an editor program released all the resources it claimed before, such a transaction is called *claim termination.* Resource claims accumulate. Each acquisition has a positive effect on the total resource claim and each release decreases it. The table below shows the evolution of the total claims for the example transactions. The 10th transaction, `editor 0` releases all the resources in possession by `editor` at that moment. There are two such prior claims, `editor 35` and `editor 40` that sum up to 75. That's why after `editor 0` 75 units of memory are subtracted from the total claims.

#	identifier	value	total claims
1	mail	100	100
2	mail	45	145
3	browser	80	225
4	editor	35	260
5	mail	20	280
6	browser	-60	220
7	editor	40	260
8	mail	-30	230
9	browser	85	315
10	editor	0	240
11	mail	15	255

Similarly, resource claims for a particular item can be summed up:

#	identifier	value	mail total	browser total	editor total
1	mail	100	100	0	0
2	mail	45	145	0	0
3	browser	80	145	80	0
4	editor	35	145	80	35
5	mail	20	165	80	35
6	browser	-60	165	20	35
7	editor	40	165	20	75
8	mail	-30	135	20	75
9	browser	85	135	105	75
10	editor	0	135	105	0
11	mail	15	150	105	0

The resource on which the claims are made is not infinite. There's some initial, positive number that's available for claiming. If this initial number is 500, the free pool (available resources) will always be non-negative and will change as follows:

#	description	value	total claims	free pool
0	—	—	—	500
1	mail	100	100	400
2	mail	45	145	355
3	browser	80	225	275
4	editor	35	260	240
5	mail	20	280	220
6	browser	-60	220	280
7	editor	40	260	240
8	mail	-30	230	270
9	browser	85	315	185
10	editor	0	240	260
11	mail	15	255	245

However, if the initial pool is only 300, the 9th claim `browser 85` will lead to a situation where the sum of claims exceeds the resource size. This is known as *resource collapse* and should never happen.

Sometimes, there is an error in a list of resource transactions that leads to a negative total claim for some item. In the list below, the last claim `browser -90` brings to *browser total* to -10.

#	description	value	mail total	browser total	editor total
1	mail	100	100	0	0
2	mail	45	145	0	0
3	browser	80	145	80	0
4	editor	35	145	80	35
5	mail	20	165	80	35
6	browser	-90	165	-10	35

If such a situation happens the transaction list is invalid and it should be signalled to a user.

Transaction as a C++ structure

A single transaction/ claim can be modelled as a `struct`:

```
struct Transaction {
    std::string id;
    int value;
};
```

A list of transactions is another `struct`:

```
struct TransactionList {
    static const int MAX_TRANSACTIONS{100};
    Transaction transactions[MAX_TRANSACTIONS];
    int size;   // the actual count of transactions stored in the array

    // member functions that operate on transactions:
    // ...
};
```

Task

Write a program that processes lists of transactions stored in text files. A list never contains more than 100 transactions. When run the program should do the following:

1. Ask a user for a name of a text file, e.g. *filename.txt*.
2. Read the text file with the name provided by the user (or print an error message if the file doesn't exist).
3. Store the data read from the file in the `TransactionList` structure.
4. Present a list of choices to the user with operations he can perform. Most of those operations must call *member functions* of the `TransactionList struct` that must be added to it and implemented.

The following operations should be available after loading a transaction list:

1. Count of transactions - prints the number of transactions.
2. Minimum resource size - prints the minimum resource size (free pool) that won't lead to *resource collapse* for this list of transactions.
3. All transactions - prints all the transactions, together with the total running claim.
4. Transactions of id - prints all the transactions for an identifier entered by the user, together with the total running claim for this identifier.
5. Sum of claims until n - prints the sum of all claims up to the n-th transaction.
6. List identifiers (*)- prints the unique identifiers in the transaction list.
7. Validate (**)- validates the transaction list by verifying that no total claim for any item becomes negative at any point.
8. Maxima (***) - prints the identifier(s) of an item(s) that:
    - has the biggest resource claim at the end of the list (end-of-list maximum).
    - has the biggest resource claim at any point in the transaction list.
9. Check pool size (***) - given a pool size entered by the user, checks if it's enough to execute the transactions without a *resource collapse* event. If yes, prints the free pool

size at the end and the free pool size under maximum load. If no, lists the transactions that must be removed for the total claims to fit within the given size. You cannot remove the first transaction of some process. Print the free pool sizes after removing those claims.

10. Quit - quits the program.

Items marked with the asterisk symbols (*) are difficult. The more asterisks, the more difficult an item is to implement. Items with 3 * are not mandatory. It's also not mandatory that your program correctly processes lists that contain *claim termination* transactions (those are transaction with a value of 0).

An interaction with this program for the list show at the beginning of this assignment might look like this:

```
> ./process_transactions

Please enter a file name. Empty file name quits the program.
File name: trans01.txt

Opening "trans01.txt"
Select option:
1.  Count of transactions
2.  Minimum resource size
3.  All transactions
4.  Transactions of id
5.  Sum of claims until n
6.  List identifiers
7.  Validate
8.  Maxima
9. Check pool size
10. Quit

> 1

There are 11 transactions.

> 2

The minimum resource size is 315.

> 3

1   mail    100 100
2   mail    45  145
3   browser 80  225
4   editor  35  260
5   mail    20  280
6   browser -60 220
7   editor  40  260
8   mail    -30 230
9   browser 85  315
```

```
10  editor  0   240
11  mail    15  255

> 4
ID: browser

3   browser 80  80
6   browser -60 20
9   browser 85  105

> 5
Until when: 7

The sum of claims until 7 transaction is 260.

> 6

mail
browser
editor

> 7
The list is valid.

> 8

The maximum total claim at the end is 150 for `mail`.
The maximum total claim in the list is 165 for `mail`.

> 9
Enter pool size: 350

OK
Free pool at the end: 95
Free pool under the maximum load: 35

> 9
Enter pool size: 250
Not OK.
Remove:
2   mail    45
9   browser 85
Free pool at the end: 80
Free pool under the maximum load: 15

> 9
Enter pool size: 150
No solution.
```

```
> 10

Bye!
```

Requirements

1. Separate your program into multiple files - your main program file (*main.cpp*) should ideally contain only the `main` function.
2. Use header and implementation files.
3. Check input and output operations for errors by inspecting the states of streams.
4. Use C++ functions, not C.
5. Use type inference (`auto`) whenever possible.
6. Use one naming convention.
7. No global state variables.

Tips and hints

1. Most of the operations should call a member function of `TransactionList`, like this one:

```
struct TransactionList {
    // ...
    int size;

    // ...
    int count() const;
};

int TransactionList::count() const {
    return size;
}

// ...


void print_count() {
    std::cout << "There are " << transactions.count() <<  " transactions \n";
}

void menu(){
    // ...
    switch(choice){
        case 1:
            print_count();
            break;
        case 2:
            // ...
    }
}
```

2. The functions `print_count` and `menu` above should (ideally) also belong to some structure:

```
struct Program {
    TransactionList transactions;

    void menu() const;
    void print_count() const;
};

struct TransactionList {
    // ...
    bool read_from_file(std::string file_name);
};

int main(){

    auto file_name = ask_file_name();

    TransactionList transactions;

    if (!transactions.read_from_file(file_name)){
        std::cout << "Error opening file\n";
    }

    Program program{ transactions };

    program.menu();
}
```

Check pool size

It's fairly easy to calculate whether a pool of a given size is sufficient for processing a transaction list - we can just piggy-back on the option 2 *Minimum resource size*. For the example list the minimum pool size is 315, so any smaller than this must have transactions removed.

Consider a pool of size 250.

#	description	value	total claims	free pool
0	—	—	—	250
1	mail	100	100	150
2	mail	45	145	105
3	browser	80	225	25
4	editor	35	260	-10
5	mail	20	280	-30
6	browser	-60	220	30
7	editor	40	260	-10
8	mail	-30	230	20
9	browser	85	315	-65

#	description	value	total claims	free pool
10	editor	0	240	10
11	mail	15	255	-5

The first transaction at which *resource collapse* occurs is transaction 4. It cannot be removed because it's the first transaction of the *editor* process. The first (and only) transaction that can be removed to re-establish order without violating the rules is transaction 2 `mail 45`. However, this has its own consequence - removing an acquisition of 45 units of memory by *mail* must be followed by adjustment of transaction 8, where 30 units are released. In fact, instead of releasing memory, *mail* will acquire 15 units (the difference 45 - 30) here. This makes the table look as follows:

#	description	value	total claims	free pool
0	—	—	—	250
1	mail	100	100	150
2	-removed-	—	100	150
3	browser	80	180	70
4	editor	35	215	35
5	mail	20	235	15
6	browser	-60	175	75
7	editor	40	215	35
8	mail	15	230	20
9	browser	85	315	-65
10	editor	0	240	10
11	mail	15	255	-5

Now, the offending claim is transaction 9 `browser 85`. Luckily it can be simply removed, producing the following list:

#	description	value	total claims	free pool
0	—	—	—	250
1	mail	100	100	150
2	-removed-	—	100	150
3	browser	80	180	70
4	editor	35	215	35
5	mail	20	235	15
6	browser	-60	175	75
7	editor	40	215	35
8	mail	15	230	20
9	-removed-	—	230	20
10	editor	0	155	95
11	mail	15	170	80

In total, two transactions were removed. After the removal, the free pool size under the maximum load is 15 and at the end of the list it's 80.

For the initial free pool size of 150 it's impossible to come up with a removal scheme that satisfies the constraints. Removing transaction 2 `mail 45` still results in *resource collapse* at transaction 3 `browser 80`. This one cannot be removed because it's the first transaction of this process.