Homework 5

Assigned: 11/3

Due: 11/15, 11:59pm

Possible Points: 100

Submit on Canvas a zip or zipped tar file containing your solution code for each problem (just the cpp files, please don't submit the solution/project files), turn in one cpp file for each problem. If a problem has multiple parts, it should still be one file. Each problem specifies what you should title the file containing your solution so that the autograder can find it. Incorrectly named programs will not be graded and thus will get 0 points for the problem.

Since the assignment will be graded using an automated grading system you should have your program's output match the sample output shown. If your program fails to compile using Visual Studio 2015 it will recieve zero points. If your program compiles but fails to run properly (runtime errors or wrong output) points will be deducted based on how correct the program is.

If you have any questions or concerns about the assignment do not hesitate to post on the discussion board on canvas, send us mail on canvas, or see us after class or during office hours.

Students are expected to write their source code from scratch, those who copy code from each other or online (including from Canvas examples) will be reported for cheating. We will also look at your code so definitely don't just hard-code the answer and print it.

To receive full credit for the solution to the homework you must use good programming practices, including: not using global variables, reusing functions (do not replicate code), use meaningful variable names...

1. Matrix: 40 Points

File: `matrix.cpp`

In this problem you will implement arbitrary sized matrix that can hold elements of any type (defined by a template parameter T). You will create a class `Matrix` which has two constructors `Matrix()` (creates empty matrix) and `Matrix(int m, int n)` (creates m by n matrix). The destructor `~Matrix()` needs to deallocate the memory which holds matrix elements. Apply **rule of three**, so add copy constructor and assignment operator. Additionally, you need to implement following operators:

- `operator*` (multiplies two matrices and returns new matrix),

- `operator+` (adds two matrices and returns new matrix),
- `operator[]` (returns an element addressed with one index),
- `operator()` (returns an element addressed with two indices),
- `operator<<` (sends a matrix to ostream; do not use std::cout in the operator), and
- `operator>>` (reads a matrix).

The matrix element types tested in the main will only be `double` and `int` thus you need two instances of the templated Matrix. As usual, the input will contain the number of cases and for each case the operation to perform, the type of the operands, and the two matrices. The size of each matrix is specified by two numbers (m - number of rows and n - number of columns) and its elements. After performing the specified operation you need to print the resulting matrix. All inputs will be valid, so there is no need to check if the input are numbers or that the matrix sizes are compatible.

Example Input

```
3
multiply
int
2 3
1 2 3
5 6 7
3 5
-9 8 7 6 5
4 3 2 1 0
1 0 0 1 3
add
double
2 2
1 0
0 1
2 2
5 2
3 4
add
int
1 3
10 9 -98
1 3
10 0 99
```

Example Output

```
Case 0:
2 14 11 11 14
-14 58 47 43 46
Case 1:
6 2
3 5
Case 2:
20 9 1
```

2. Contact Manager: 60 Points

Files:

- `contact.cpp`, `contact.h`
- `contacts_main.cpp`

What makes associative containers like `std::map` powerful and useful is that they can associate keys with values and quickly find the value associated with a specific key. For example in a contact manager we want to be able to associate a person's name with some information about them (like phone number, e–mail, etc.) and quickly look up this information given a first and last name. In this problem you will create a contact management tool that will store some information about the user's contacts and allow them to list them, add or remove contacts and search for a contact by name.

Contact Information

For each contact you will store the following data:

- First name
- Last name
- Phone number. Note: while phone numbers are "numbers" they are not numeric (like an int), they are strings of characters in [0, 9]. **Do not** store the phone number as an int.
- E–mail address

The contact information should be stored in a class (maybe called `Contact`) declared in `contact.h` and defined in `contact.cpp`. Your contact class should also provide a default constructor (one that takes no parameters) as this is needed by the `std::map` to perform some operations.

User Interaction

The user will need to be able to interact with the contact manager by adding/removing contacts, searching by name and listing all contacts in the manager. They should also be able to exit the manager. To indicate to the user that the program is awaiting their input you should print a caret (>) followed by a space as a prompt. In the examples below the user input and expected output are shown interleaved, with the user input preceeded by the prompt.

After receiving and executing the user's command you should return to the initial input prompt so they can enter another command. Examples of a full interactive session are shown after the details of each command. To help you start this is what your main function should look like to loop over and handle a user's commands until they enter "exit". You can use this as starter code for your program. Note that there is no cases loop in this assignment, just the user interaction loop.

```
int main(){
    std::string cmd;
    do {
        if (cmd == "exit"){
            break;
        } else if (cmd == "list"){
            // List contacts in the manager
        } else if (cmd == "add"){
            // Add a contact to the manager
        } else if (cmd == "remove"){
            // Remove a contact from the manager
        } else if (cmd == "find"){
            // Find a contact in the manager
        }
        // Print a terminal prompt so the user knows
        // we're waiting for their input
        std::cout << "> ";
    } while (std::cin >> cmd);
    return 0;
}
```

Listing All Contacts

When the user enters the `list` command and you should print out the first and last names of all the contacts in the manager. If there are no contacts in the manager you should just print `no contacts`. The contacts should be printed in alphabetical order, sorted by last name then by first name. This sorting will be done automatically for you if you build the key in the `std::map` properly.

Example: The manager contains three people: Grace Hopper, Dennis Ritchie and Leslie Lamport

```
> list
Hopper, Grace
Lamport, Leslie
Ritchie, Dennis
```

Example: The manager is empty.

```
> list
no contacts
```

Adding a Contact

When the user enters the `add` command you should prompt them to enter the first name, last name, phone number and e–mail address for the contact, one after the other. You should print a prompt each time so the user knows what the program is expecting them to input. First you should read the first name and print the prompt `first name:`, for the last name the prompt is `last name:`, for the phone number it's `phone number:` and for the e–mail address it's `e-mail:`. **If a contact with same first and last name is already in contact manager, then overwrite it with this new contact.**

Note that there is a trailing space after the colon printed for each prompt so the user's input doesn't run right against the prompt.

Example: The user wants to add Ada Lovelace as a contact.

```
> add
first name: Ada
last name: Lovelace
phone number: 555123456
e-mail: ada@lovelace.mathmail
```

Removing a Contact

When the user enters the `remove` command you should prompt them to enter the first and last name of the contact, with the same prompts used when adding a contact (`first name:` and `last name:`). If the contact is found they should be removed from the list and you should print a confirmation of the form `removed FIRST LAST` where FIRST and LAST are the first and last name of the contact that was removed. If no contact is found with the name entered you should print out `contact not found`.

Example: The user wants to remove Richard Stallman, who is in the manager.

```
> remove
first name: Richard
last name: Stallman
removed Richard Stallman
```

Example: The user wants to remove John McCarthy, who is not in the manager.

```
> remove
first name: John
last name: McCarthy
contact not found
```

Finding a Contact

When the user enters the `find` command you should prompt them to enter the first and last name to search for, using the same prompt style as before. If the contact is found you should print out the full information about the contact (the name, phone number and e–mail address). If the contact is not found you should print out `contact not found`.

If a contact is found the contact's information should be printed out as:

```
Name: FIRST LAST
Phone Number: PHONENUMBER
E-mail: EMAIL
```

Where FIRST, LAST, PHONENUMBER and EMAIL correspond to the contact's data.

Example: The user searches for Donald Knuth, who is in the manager.

```
> find
first name: Donald
last name: Knuth
Name: Donald Knuth
Phone Number: 555898123
E-mail: knuth@donald.texmail
```

Example: The user searches for Edsger Dijkstra, who is not in the manager.

```
> find
first name: Edsger
last name: Dijkstra
contact not found
```

Exiting the Manager

When the user enters the `exit` command your program should exit as done in the main function provided for you above. There is no example use case as the

only thing your program should do after receiving the "exit" command is exit.

Some Notes and Hints on `std::map`

You should use a `std::map` to store the contact information where the keys will be some combination of their first and last name (as people can have the same first or last name, but not both) and the value will be your `Contact` struct. The actual values returned by the iterator over your map will be a `std::pair<Key, Value>` where key would likely be `std::string` and Value would be a `Contact`. To access each element of the pair you can use `first` or `second` as shown below.

```
std::map<std::string, int> my_map{{"hello", 4}, {"bob", 2}};
for (const auto &e : my_map){
    // first accesses the key, second accesses the value
    std::cout << e.first << " mapped to " << e.second << "\n";
}
```

This code will print:

```
bob mapped to 2
hello mapped to 4
```

To insert into a map we can index it by the key with `operator[]`. If the key is not found a new entry is created using the default constructor and a reference returned (this is why we need a default constructor for Contact), if the key is found the existing entry will be returned.

```
std::map<std::string, int> my_map{{"hello", 4}, {"bob", 2}};
// Create key, value pair for joe and set it to 10
std::string joe = "joe";
my_map[joe] = 10;
// Update the existing entry for hello
my_map["hello"] = 5;
for (const auto &e : my_map){
    // first accesses the key, second accesses the value
    std::cout << e.first << " mapped to " << e.second << "\n";
}
```

This code will print:

```
bob mapped to 2
hello mapped to 5
joe mapped to 10
```

To find an entry by its key you can use `find`, see the interactive example on the documentation page for usage, and for how to see if the key was not found. Since the map will create the entry if it's not found when indexing with `operator[]` you must use `find` to see if a key exists in the map. `find` returns an iterator so

we can use `find` and `erase` to remove an element by its key since `erase` takes the iterator to the element that we want to remove.

You can assume that you will receive valid input and commands and for simplicity that case does matter, for example Dennis Ritchie is a different person than dennis ritchie.

Example Usage

Interactive Session

Since the program is interactive just reading the output can be kind of confusing. Instead you can copy-paste the sample input (from `contacts_stdin.txt`) to the console to see what the actual usage would look like. You should see the following if you paste the sample input into your console. There is no case loop for this problem, just the user interaction loop.

```
> list
no contacts
> add
first name: Ada
last name: Lovelace
phone number: 555123456
e-mail: ada@lovelace.mathmail
> list
Lovelace, Ada
> add
first name: Grace
last name: Hopper
phone number: 555823457
e-mail: hopper@debugging
> add
first name: Dennis
last name: Ritchie
phone number: 555898123
e-mail: dennis@email
> add
first name: Leslie
last name: Lamport
phone number: 000912931
e-mail: leslie.lamport
> list
Hopper, Grace
Lamport, Leslie
Lovelace, Ada
Ritchie, Dennis
```

```
> remove
first name: Dennis
last name: Ritchie
removed Dennis Ritchie
> remove
first name: Dennis
last name: Ritchie
contact not found
> list
Hopper, Grace
Lamport, Leslie
Lovelace, Ada
> find
first name: Dennis
last name: Ritchie
contact not found
> find
first name: Leslie
last name: Lamport
Name: Leslie Lamport
Phone Number: 000912931
E-mail: leslie.lamport
> add
first name: Dennis
last name: Ritchie
phone number: 555898123
e-mail: dennis@email2
> list
Hopper, Grace
Lamport, Leslie
Lovelace, Ada
Ritchie, Dennis
> find
first name: Dennis
last name: Ritchie
Name: Dennis Ritchie
Phone Number: 555898123
E-mail: dennis@email2
> add
first name: Dennis
last name: Ritchie
phone number: 555898123
e-mail: dennis@email
> add
first name: Bob
last name: Ritchie
```

```
phone number: 555898123
e-mail: dennis@email
> add
first name: Bob
last name: Jackson
phone number: 555898123
e-mail: dennis@email
> add
first name: Jack
last name: Jackson
phone number: 555898123
e-mail: dennis@email
> add
first name: Jack
last name: Jackson
phone number: 555898123
e-mail: dennis@email
> add
first name: Grace
last name: Hopper
phone number: 5551020001238123
e-mail: grace@email
> add
first name: Brian
last name: Kernighan
phone number: 555
e-mail: brian.kernighan@email
> list
Hopper, Grace
Jackson, Bob
Jackson, Jack
Kernighan, Brian
Lamport, Leslie
Lovelace, Ada
Ritchie, Bob
Ritchie, Dennis
> exit
```

Input/Output Redirected Session

See the `stdin_stdout.zip` file for the input/output files for this example since it's a bit hard to read the output here due to long lines.

Input:

```
list
```

```
add
Ada
Lovelace
555123456
ada@lovelace.mathmail
list
add
Grace
Hopper
555823457
hopper@debugging
add
Dennis
Ritchie
555898123
dennis@email
add
Leslie
Lamport
000912931
leslie.lamport
list
remove
Dennis
Ritchie
remove
Dennis
Ritchie
list
find
Dennis
Ritchie
find
Leslie
Lamport
add
Dennis
Ritchie
555898123
dennis@email2
list
find
Dennis
Ritchie
add
Dennis
```

```
Ritchie
555898123
dennis@email
add
Bob
Ritchie
555898123
dennis@email
add
Bob
Jackson
555898123
dennis@email
add
Jack
Jackson
555898123
dennis@email
add
Jack
Jackson
555898123
dennis@email
add
Grace
Hopper
5551020001238123
grace@email
add
Brian
Kernighan
555
brian.kernighan@email
list
exit
```

Expected Output

```
> no contacts
> first name: last name: phone number: e-mail: > Lovelace, Ada
> first name: last name: phone number: e-mail: > first name: last name: phone number: e-mail
Lamport, Leslie
Lovelace, Ada
Ritchie, Dennis
> first name: last name: removed Dennis Ritchie
> first name: last name: contact not found
> Hopper, Grace
```

```
Lamport, Leslie
Lovelace, Ada
> first name: last name: contact not found
> first name: last name: Name: Leslie Lamport
Phone Number: 000912931
E-mail: leslie.lamport
> first name: last name: phone number: e-mail: > Hopper, Grace
Lamport, Leslie
Lovelace, Ada
Ritchie, Dennis
> first name: last name: Name: Dennis Ritchie
Phone Number: 555898123
E-mail: dennis@email2
> first name: last name: phone number: e-mail: > first name: last name: phone number: e-mail
Jackson, Bob
Jackson, Jack
Kernighan, Brian
Lamport, Leslie
Lovelace, Ada
Ritchie, Bob
Ritchie, Dennis
>
```