# MCIS-4135

# Homework Assignment 5

## Purpose:

This assignment gives you an opportunity to learn about files, arrays, and pointers.

## What to hand in:

Hand in listings of the program which should be captured in 3 files: a header file (`cost.h`), a test program (`hw5.cpp`), and an implementation file (`cost.cpp`). Also, hand in the output of a sample run (or runs) of the program. All of these must be formatted in a reasonable style. Choose sufficient test cases to ensure your functions work properly.

This program is written much like a recipe. If you follow the instructions carefully, it will all work. Additional time is provided with this assignment so that you can reflect on how each step works, and how the features of C++ provided this interwoven solution.

## Problems:

### Accounting for files:

A cost involves a description, and an amount (for simplicity). A description may be handled by a string, and a cost may be handled by a double, for this exercise. A class discussion topic will be what a more realistic set of records might look like, and the advantages of different formats and representations.

Write a program that opens a file, reads records into an container of data structures, and prints out the sum in order. Use the following steps to accomplish this goal:

1. Create a data structure to represent the record (struct cost in cost.h)
2. Write a function called `parse_account` that parses one (string) record from a file, and populates a data structure with that data
3. Write a function called `sum_accounts` that, when passed a container of structures, returns a double representing the sum of the amounts in those structures.
4. Create a `main` program
   a) Create an appropriate container of data structures
   b) Open the accounts file (`costfile.txt`)
   c) While you can read a line from the file without error
      - Call `parse_account,` which returns a data structure.
      - Add the returned data structure to the container (using, say, `push_back`)
   d) Call `sum_accounts` to determine the amount of money represented
   e) Print a message and the result.

Each of these will be discussed in a section below in the notes.

## Notes:

For step 1, the data structure should have two components, a description `string`, and an amount `double`. The string type is found in `<string>`, and is declared in package `std`, like all other standard library components.. This data structure, if called “cost”, belongs in `cost.h`.

The cost structure might look like the following:

```
struct cost {
     string description;
     double amount;
```

```
};
```

Since the description goes in a header (.h) file, your IDE may have a "header file" category within your project to create this file within.

We also create the "derived" type used for a collection of these data structures in the header file, called `costRecords`, by including `<vector>`, and adding the line:

```
typedef vector<cost> costRecords;
```

This statement says that the type "costRecords" is another name for a vector of cost structures.

For step 2, we are creating a function in the `cost.cpp` file. We use another type from *iostream*, called an *stringstream.* To use an `istringstream`, you must include `<sstream>`, and also the statement `using std::istringstream` or `using namespace std`. Note the use of the char array to represent input, as is common in C++ with console input. The parse_account function may look as follows:

```
cost parse_account (char record[]) {
   cost thisCost;
   istringstream istr(record);
   istr >> thisCost.description;
   istr >> thisCost.amount;
   return thisCost;
}
```

Note how the use of the *istringstream* allows the input stream to keep working, even though the user's input might not adhere to the rules we want. Note the use of structure notation (using the dot). This function belongs in `cost.cpp`. The prototype, which is

```
cost parse_account (char record[]);
```

goes in `cost.h`.

For step 3, we get to once again use structure notation to loop through all the records. We will rely on the main program to create the container, and pass it in as a parameter to this function. In this sample code we will use array notation—but one could also use iterator notation (as you might see in the sample solution).

```
double sum_accounts(costRecords accounts) {
   double result = 0.0;
   for (int i=0; i<accounts.size(); i++)
       result += accounts[i].amount;
   return result;
}
```

This function goes in `cost.cpp`, and the prototype goes in `cost.h`.

For step 4, and all of its sub-steps, the code goes in a file such as `hw5.cpp`.

Note that this approach gives us 3 files altogether for the solution. The file cost.h provides the description of the types and functions we will use; the file cost.cpp contains the implementation of the functions described in cost.h; and the file hw5.cpp contains the main program. This paradigm is used throughout C++ development to keep files small, focused on a single problem, and also makes them able to be used on other projects!

Note that this approach gives us 3 files altogether for the solution. The file cost.h provides the description of the types and functions we will use; the file cost.cpp contains the implementation of the functions described in cost.h; and the file hw5.cpp contains the main program. This paradigm is used throughout C++ development to keep files small, focused on a single problem, and also makes them able to be used on other projects!

For step 4a, you will need to declare the container.

```
costRecords cost_items;
```

As in the above, this declaration appears in two places. In the cost.h file, it appears as

```
extern costRecords cost_items;
```

and in the cost.cpp file, it appears as

```
costRecords cost_items;
```

This matches the idea of a declaration (telling the compiler it exists) versus a definition (telling the compiler to create it.

For step 4b, to open the file, you will need the `ifstream` which is in the `<fstream>` library. Similar to the description above, you must include this header and follow with one of the statements `using std::ifstream` or `using namespace std`. The file can be opened by the statement

```
ifstream input("costfile.txt"); // Or, with a path, ifstream input("c:/mcis4135/costfile.txt");
```

See below for some sample contents of this file.

You may create the file with any text editor—just be sure to save it where you are putting the program executable—or you may create it in Visual Studio: File/New:Text File (costfile.txt).

For step 4c, the while loop for the file looks something like this (from input loops in C++ monograph). Note that here we use a character buffer for input:

```
char buffer[100];
while ( input.getline(buffer, sizeof buffer) ) {
   cost c = parse_account(buffer);
   cost_items.push_back(c);
}
```

The above loop will read the input from the file, up to 100 characters per line, parse the line into a structure, and then store a copy of the structure in the container.

For step 4d, to get the final result, we just call sum_accounts:

```
double sum = sum_accounts(cost_items);
```

For the final step, 4e, print the result as requested, using the `<<` and the `cout` stream.

Try this with some sample data, such as the following lines in `costfile.dat` (Note that I allowed no spaces in the description portion of the record). You may add or create your own data file to test the program with.:

```
Pass_Go 200.0
Reading_RR -50.0
Connecticut -120.0
Chance 25.0
```

## Evaluation Criteria:

| | |
|---|---|
| 30 | Interface file: proper definitions and content, date type, compliant set of operations |
| 35 | Implementation file: proper use of C++ for implementation |
| 35 | Test program: full test of interfaces, including boundary conditions. Inclusion of results. ***Note: If your program doesn't compile, hand in the error listing as the results!*** |