

Project 02 : UIC Bank v2.0

(doc v1.1)

Assignment: C++ program to manage N bank accounts

Evaluation: Gradescope followed by manual review

Policy: Individual work only

Complete By:

Part 01 (50pts): Thursday Feb 4th @ 11:59pm CDT, late day Fri 2/5 @ 11:59pm

Part 02 (50pts): Saturday Feb 6th @ 11:59pm CDT, late day Sun 2/7 @ 11:59pm

Pre-requisites: Panopto videos 09 – 10, HW 05, Lab 03

Overview

The goal here in project 02 is to extend the banking program you created to support N bank accounts. In project 01 we only had 5 bank accounts --- not a very useful bank. Here in project 02 we're going to support any number of bank accounts, along with the ability to add and delete accounts. The primary change is the addition of dynamically-allocated array for storing and manipulating the bank data.

If you completed project 01, you can start with your solution as a base for project 02. If you did not complete project 01, a Panopto recording (#11) will be posted to Blackboard that will review our solution to project 01; feel free to use anything we demonstrate.

Project 02 will be submitted in two parts, hence the 2 separate deadlines. In part 01 you'll introduce arrays and update the interactive loop accordingly. In part 02, you'll focus on the dynamic aspect of the arrays and add support for adding, deleting, and merging accounts. To receive full credit you must finish each part by the specified deadline; you cannot earn full credit by skipping part 01 and submitting the complete project at the end. If you submit a part 24 hours before the deadline (or earlier) you'll earn 10% "early bonus points", which can be used to offset late penalties. If you submit late, it's a 10% penalty that can be erased by submitting another part early.

Part 01: Input, interact, output

In part 01 you're going to write the core of the application: input, interaction with the user, output. **UIC Bank 2.0** now supports $N > 0$ customers, each with an account number (positive integer) and a balance (a real number that can be negative, zero, or positive). In part 01 you're going to support the same set of interactive commands as project 01: +, -, ?, ^, *, and x. The only difference is that the * command will output a subset of accounts --- those within a specific range of account numbers. Here's a summary of the commands:

1. Deposit (+): **+ account balance**
2. Withdrawal (-): **- account balance**
3. Check balance (?): **? account**
4. Find the account with the largest balance (^): **^**
5. List by account range low..high, inclusive (*): *** low high**
6. Exit (x): **x**

Another difference is the banking file format: the first line of the file contains N, the # of bank accounts. For example, here are the contents of the “bankv2-1.txt” banking file that we’ll be using for demonstration purposes:

```
5
7 250.98
34 -10.00
123 5000000.25
467 24.08
9921 1250.75
```

Note three differences from the previous project: (1) the first line of the file contains the # of bank accounts (5 in this case, but this will change), (2) the data is now guaranteed to be in ascending order by account number, and (3) the balances are always in a fixed format with 2 digits of precision. You may assume the file format is valid, no error checking of the file contents is required.

The screenshot to the right shows a sample run based on the “bankv2-1.txt” text file. When the user enters “x” to exit the loop, the program writes the account data back to the same file and ends. Here’s the corresponding “bankv2-1.txt” file after execution of the above:

```
** Welcome to UIC Bank v2.0 **
Enter bank filename>
bankv2-1.txt
** Inputting account data...
** Checking arrays...
1. 7, $250.98
5. 9921, $1250.75
** Processing user commands...
Enter command (+, -, ?, ^, *, x):
* 7 9921
Account 7: balance $250.98
Account 34: balance $-10.00
Account 123: balance $5000000.25
Account 467: balance $24.08
Account 9921: balance $1250.75
Enter command (+, -, ?, ^, *, x):
^
Account 123: balance $5000000.25
Enter command (+, -, ?, ^, *, x):
oops
** Invalid command, try again...
Enter command (+, -, ?, ^, *, x):
+ 9921 500
Account 9921: balance $1750.75
Enter command (+, -, ?, ^, *, x):
- 34 10
Account 34: balance $-20.00
Enter command (+, -, ?, ^, *, x):
? 123
Account 123: balance $5000000.25
Enter command (+, -, ?, ^, *, x):
? 99
** Invalid account, transaction ignored
Enter command (+, -, ?, ^, *, x):
x
** Saving account data...
** Done **
```

```
5
7 250.98
34 -20.00
123 5000000.25
467 24.08
9921 1750.75
```

Notice the first line is N, and the remaining lines use a fixed format with 2 digits of precision. You’ll need to configure the output file much like we do the console output:

```
outfile << std::fixed;
outfile << std::setprecision(2);
```

This must be done after you open the output file, and before you start writing to the file; this assumes your ofstream file variable is named “outfile”. Recall we have to configure the console in a similar manner:

```
cout << std::fixed;
cout << std::setprecision(2);
```

Don't forget to `#include <iomanip>` at the top of "main.cpp".

Your program is required to check if the filename entered can be successfully opened for input. If not, your program should output an error message and immediately return 0; you can assume that if it was opened successfully for input, it will successfully open for output (though it never hurts to check). Example:

```
** Welcome to UIC Bank v2.0 **
Enter bank filename>
pizzabank.txt
** Inputting account data...
**Error: unable to open input file 'pizzabank.txt'
```

Project restrictions

How you solve the problem is just as important as developing a correct solution. In this assignment we are going to impose the following restrictions, which you must adhere to. Breaking a restriction typically leads to a score of 0.

1. No global variables. A global variable is a variable declared in "main.cpp" but outside of any functions. Use local variables and parameters instead of global variables.
2. The only data structure permitted is dynamically-allocated arrays, created using the **new** operator. No other data structures: no static arrays, no vectors, no linked-lists, no use of the built-in C++ containers. To be more specific, you'll need to have 2 dynamically-allocated arrays: one for the accounts (an int array), and one for the balances (a double array). The expectation is that the size of these arrays will dynamically adapt based on the input file, and any accounts that may be added or deleted (part 02). It is not acceptable to allocate larger arrays than required and assume those will be big enough; such an approach will yield a project score of 0.
3. To help enforce restriction #2, your main() function is required to call the following function:

```
void checkArrays(int accounts[], double balances[], int N);
```

This will be discussed in more detail in the next section ("Getting Started").

4. To help enforce good design, you are required to provide (and use) the following two functions:

```
int search(int accounts[], int N, int acct);
int maxBalance(double balances[], int N);
```

This will be discussed in more detail in the next section ("Getting Started").

5. This assignment is about arrays, not files. You are allowed to open the banking file exactly twice, once for input at program start and again for output at program end.
6. No use of the built-in algorithms for searching, sorting, min, max, etc. When necessary, you'll write your own functions.
7. Your program must represent a general solution, e.g. the ability to input & process any banking file not

just what you see here. Coding to the test cases on Gradescope will lead to a score of 0.

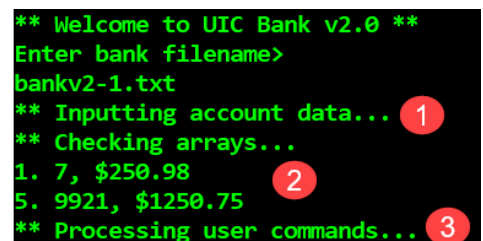
Getting Started

No C++ starter code is being provided, but are encouraged to build upon your solution to project 01. If you were unable to complete project 01, a Panopto recording (#11) will be posted to Blackboard that will go over our solution to project 01; feel free to use anything we demonstrate. If you prefer to work on Codio, a project “cs141-project02” has been created for you; alternatively you can continue to use the project associated with project 01.

Be sure to review the restrictions in the previous section before working out the details of your solution. In particular, note that you must use dynamically-allocated arrays, in particular two such arrays: one for the account numbers, and one for the balances. To help ensure this approach is being followed, we are requiring that you place the following function in your “main.cpp” program file, and call it from your main() function:

```
//  
// checkArrays  
//  
// Required function that currently outputs to console, but when submitted  
// to gradescope this function will be replaced by one that performs more  
// extensive checks to make sure the data was input correctly.  
//  
void checkArrays(int accounts[], double balances[], int N)  
{  
    cout << std::fixed;  
    cout << std::setprecision(2);  
  
    cout << "*** Checking arrays..." << endl;  
    cout << "1. " << accounts[0] << ", $" << balances[0] << endl;  
    cout << N << ". " << accounts[N-1] << ", $" << balances[N-1] << endl;  
}
```

Copy-paste this [function](#) exactly. Also, do not modify this function because it will be replaced when you submit to Gradescope. Your main() should call this function after successfully inputting the data but before the interactive loop; it represents step #2 ----->



```
*** Welcome to UIC Bank v2.0 **  
Enter bank filename>  
bankv2-1.txt  
*** Inputting account data...  
*** Checking arrays...  
1. 7, $250.98  
5. 9921, $1250.75  
*** Processing user commands...
```

To encourage good design, we are also requiring that you define two functions in your solution. First, ask yourself what the +, - and ? commands have in common? They all have to search through the accounts for the one that matches the account input from the keyboard. For this reason, you are required to implement (and use) the following search function in your +, - and ? command processing logic:

```
//  
// search  
//  
// Given an array of N accounts in ascending order, searches for
```

```
// the account that matches "acct". If found, the index of that
// account is returned; if not found -1 is returned.
//
int search(int accounts[], int N, int acct);
```

By returning the index, the appropriate account and balance can then be updated or output. When you submit to Gradescope we will call and test this function (this testing is on top of our normal testing of your entire program); submissions without this function will be rejected. Second, the ^ command performs a search for the largest balance. This search should be performed by the following function, which you are required to implement and use as part of your ^ command processing:

```
//
// maxBalance
//
// Given an array of N balances, searches for the highest balances
// and returns the index of this balance. If there's a tie, the
// first (smaller) index is returned. Assumes N > 0.
//
int maxBalance(double balances[], int N)
```

This function will also be called and tested when you submit to Gradescope; submissions without are rejected.

Have a question? Use Piazza, not email

As discussed in the syllabus, questions should be posted to our course Piazza [site](#) — questions via email are typically ignored. Remember the guidelines for using Piazza:

1. Look before you post — the main advantage of Piazza is that common questions are already answered, so search for an existing answer before you post a question.
2. Post publicly — only post privately when asked by the staff, or when it's absolutely necessary (e.g. the question is of a personal nature). Private posts defeat the purpose of piazza, which is answering questions to the benefit of everyone.
3. Ask pointed questions — do not post a big chunk of code and then ask "help, please fix this". Staff and other students are willing to help, but we aren't going to type in that chunk of code to find the error. You need to narrow down the problem, and ask a pointed question, e.g. "on the 3rd line I get this error, I don't understand what that means...".
4. Post a screenshot — sometimes a picture captures the essence of your question better than text. Piazza allows the posting of images, for "how-to" see <http://www.take-a-screenshot.org/>.

Don't post your entire answer / code — if you do, you just gave away the answer to the ENTIRE CLASS. When posting code, do so privately; there's an option to create a private post ("visible to staff only").

Part 01: local testing

To help with testing, we're going to setup a simple test script that mimics Gradescope. On Codio you'll see some "testp1" files that create and test the scenario shown on page 2. Before submitting to Gradescope, you can run a preliminary test and catch simple formatting errors without having to wait on Gradescope. To run, enter the following in a Codio terminal window:

```
./testp1
```

Your program's output will be compared to the correct output using the diff tool; no differences implies your program is correct. Feel free to modify these test files to do additional testing. If you are working in a different Codio project, or in another Linux environment, you can download a copy of the test files from [dropbox](#). See the "readme" file for details.

Part 01: submission, late policy, and grading

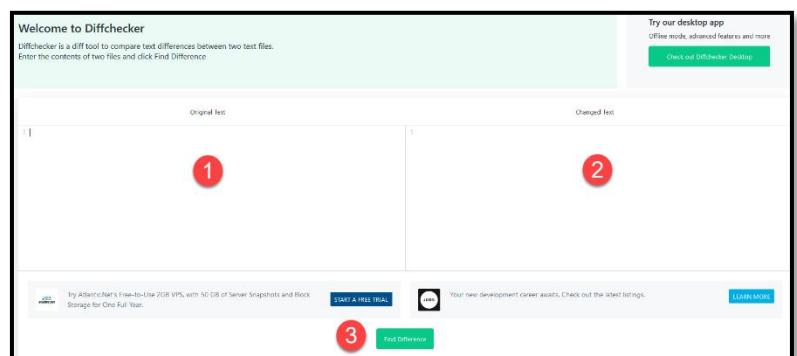
Submission: submit "main.cpp" to Gradescope under "Project 02: Part 01". You have an unlimited number of submissions, though it is expected you will test locally before submitting. If necessary, you may submit up to 24 hours late for a 10% penalty; if you submit 24+ hours before the deadline, you can earn a 10% "early bonus" to offset a future late penalty.

Your score on part 01 is based solely on correctness, as determined by Gradescope; correctness will be worth 50 points out of the total 100 project points. Note that the TAs may manually review to ensure adherence to project restrictions, and your score could change as a result.

Suggestion: when you fail a test on Gradescope, we show your output, the correct output, and the difference between the two (as computed by Linux's **diff** utility). Please study the output carefully to see where your output differs. If there are lots of differences, or you can't see the difference, here's a good tip:

1. Browse to <https://www.diffchecker.com/>
2. Copy your output as given by Gradescope and paste into the left window
3. Copy the correct output as given by Gradescope and paste into the right window
4. Click the "Find Difference" button

You'll get a visual representation of the differences. Modify your program to produce the required output, and resubmit.



In terms of grading, note that we expect all submissions to compile, run, and pass at least some of the test cases; do not expect partial credit with regards to correctness unless your program compiles and runs. By default, we grade your **last** submission. Gradescope keeps a complete submission history, so you can **activate** an earlier submission if you want us to grade a different one; this must be done before the due date. We assume **every** submission on your Gradescope account is your own work; do not submit someone else's

work for any reason, otherwise it is considered academic misconduct (how else did you end up with someone else's program?).

Part 02: More commands, dynamically adding & deleting accounts

In part 02 you're going to add support for 4 additional commands:

7. List of accounts with a negative balance: **<**
8. Total of all positive balances in the bank: **\$**
9. Add a new account: **add**
10. Delete an existing account: **del acct**

The last two commands are the most interesting, as they change the size of the arrays. When adding a new account, the new account number is 1 + the largest existing account number. The balance on this new account is \$0. However, the array will need to be enlarged by 1 in order to store this new account number and balance. Here are the appropriate steps:

1. Compute the new account number = 1 + max existing account number
2. Allocate 2 new arrays of size N+1 (using different pointer variables)
3. Copy the existing accounts and balances into the new arrays
4. Store the new account number in the last location of the new accounts array
5. Store the new balance in the last location of the new balances array
6. Delete the memory associated with the existing arrays
7. Update the array pointers to the new, larger arrays
8. Update N += 1
9. Output new account information

Please note project restriction #2. You may be tempted to create extra-large arrays at the beginning of the program so you can avoid having to enlarge the arrays here. But this type of approach is (1) not general, and (2) wasteful in other ways. When adding a new account, you are required to follow the steps above as it reinforces the goals of the assignment: dynamically-allocated arrays.

To make things more interesting, you do have flexibility in how you implement the delete command. When deleting an account, option #1 is to follow a similar approach as above and create new arrays that are one smaller. Option #2 is to shift the elements to the left, overwriting the deleted account; reduce N by 1 and the element is now "deleted". Option #3 is to "mark" the account deleted somehow and simply ignore it; this is the least attractive choice as it will require changes to how the other commands work

```
** Welcome to UIC Bank v2.0 **
Enter bank filename>
bankv2-1.txt
** Inputting account data...
** Checking arrays...
1. 7, $250.98
5. 9921, $1250.75
** Processing user commands...
Enter command (+, -, ?, ^, *, <, $, add, del, x):
* 1 10000
Account 7: balance $250.98
Account 34: balance $-10.00
Account 123: balance $5000000.25
Account 467: balance $24.08
Account 9921: balance $1250.75
Enter command (+, -, ?, ^, *, <, $, add, del, x):
- 467 100
Account 467: balance $-75.92
Enter command (+, -, ?, ^, *, <, $, add, del, x):
<
Account 34: balance $-10.00
Account 467: balance $-75.92
Enter command (+, -, ?, ^, *, <, $, add, del, x):
$
Total deposits: $5001501.98
```

```
Enter command (+, -, ?, ^, *, <, $, add, del, x):
add
Added account 9922: balance $0.00
Enter command (+, -, ?, ^, *, <, $, add, del, x):
del 34
Deleted account 34
Enter command (+, -, ?, ^, *, <, $, add, del, x):
del 123
Deleted account 123
Enter command (+, -, ?, ^, *, <, $, add, del, x):
del 99
** Invalid account, transaction ignored
Enter command (+, -, ?, ^, *, <, $, add, del, x):
<
Account 467: balance $-75.92
Enter command (+, -, ?, ^, *, <, $, add, del, x):
$
Total deposits: $1501.73
Enter command (+, -, ?, ^, *, <, $, add, del, x):
* 1 10000
Account 7: balance $250.98
Account 467: balance $-75.92
Account 9921: balance $1250.75
Account 9922: balance $0.00
Enter command (+, -, ?, ^, *, <, $, add, del, x):
X
** Saving account data...
** Done **
```


(e.g. the ^ command must now ignore deleted accounts). A good design rule is to avoid changes to working code whenever possible. But it's your choice, pick from options 1, 2, or 3. You can assume the bank will always have at least one customer, the test code won't delete all the customers.

Part 02: submission, late policy, and grading

Submission: submit "main.cpp" to Gradescope under "Project 02: Part 02". You have an unlimited number of submissions, though it is expected you will test locally before submitting. If necessary, you may submit up to 24 hours late for a 10% penalty; if you submit 24+ hours before the deadline, you can earn a 10% "early bonus" to offset a future late penalty.

Your score on part 02 is based on two factors: (1) correctness as determined by Gradescope (40 points), and (2) manual review of "main.cpp" for commenting, readability (consistent indentation and whitespace), appropriate variable names, and adherence to project restrictions. In terms of commenting, you must have a header comment at the top of "main.cpp" with your name, UIC, the date, and an overview of the project.

Example: a good header comment would explain that the program interacts with the user to perform banking transactions, and supports the following commands:

```
//
// Deposit:          + account amount
// Withdrawal:       - account amount
// Check balance:    ? account
// Find the account with the largest balance: ^
// List accounts and balances in range: * low high
// List of accounts with a negative balance: <
// Total of all positive balances in the bank: $
// Add a new account: add
// Delete an existing account: del acct
// Exit: x
//
```

Every function should have a header comment above it, with function name, description, explanation of parameters and return value(s). Follow the examples we have shown for **search()** and **maxBalance()**.

Major steps of the program should have a comment above, especially in the main() function. Example:

```
//
// (1) Input bank filename, confirm file can be opened,
// and the input N from the first line to set array size:
//
```

Line-by-line comments are only needed to explain code that isn't obvious to another programmer. Comments like the following are meaningless and a waste of time:

```
int count; // declare a count variable
count = count + 1; // add one to count
```

Here's an example of a useful line comment:


```
int result = -1; // assume search will fail and in that case return -1
```

In terms of grading, note that we expect all submissions to compile, run, and pass at least some of the test cases; do not expect partial credit with regards to correctness unless your program compiles and runs. By default, we grade your **last** submission. Gradescope keeps a complete submission history, so you can **activate** an earlier submission if you want us to grade a different one; this must be done before the due date. We assume **every** submission on your Gradescope account is your own work; do not submit someone else's work for any reason, otherwise it is considered academic misconduct (how else did you end up with someone else's program?).

Academic Honesty

In this assignment, all work submitted for grading **must** be done individually. While we encourage you to talk to your peers and learn from them, this interaction must be superficial with regards to all work submitted for grading. This means you **cannot** work in teams, you cannot work side-by-side, you cannot submit someone else's work (partial or complete) as your own. The University's policy is available here:

<https://dos.uic.edu/conductforstudents.shtml>

In particular, note that you are guilty of academic dishonesty if you **extend or receive any kind of unauthorized assistance**. Absolutely no transfer of program code between students is permitted (paper or electronic), and you may not solicit code from family, friends, or online forums. Other examples of academic dishonesty include emailing your program to another student, screen sharing, copying-pasting code from the internet, working in a group on a homework assignment, and allowing a tutor, TA, or another individual to write an answer for you. Academic dishonesty is unacceptable, and penalties range from a letter grade drop to expulsion from the university; cases are handled via the official student conduct process described at <https://dos.uic.edu/conductforstudents.shtml>.