Assignment 2 Document

This document will explore the implementation of the program bank_system.cpp.

Pre-requisites:

- The **parent class** Account houses:

Protected variables: accountNumber, holderName, and balance.
Public functions:

- Account constructor with balance checker
- DuplicateAccount checker
- getAccountNumber for depositing and withdrawal
- displayDetails
- virtual function applyInterest.

- Subclass **savingsAccount** houses:

Private: interestRate.
Public:

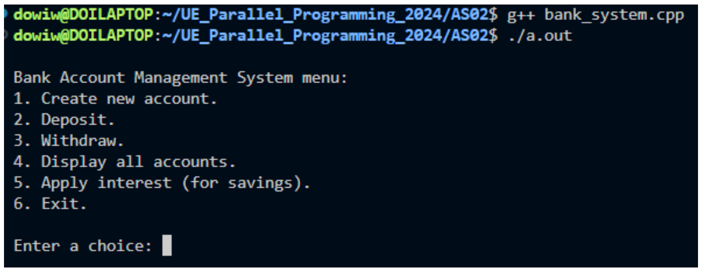
- Constructor that inherits from parent with interestRate
- Overridden applyInterest function
- Overridden displayDetails function to add interest rate

- Function **integerGetterAndChecker**
  This function will print out an input string passed by reference and check if the input is a valid integer and only an integer.

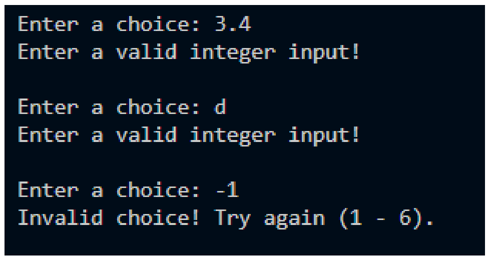
- Function **floatGetterAndChecker**
  This function does the same as the above but only for floats.

How to use the program?

When compiled and first run, this is what you will see:



```
dowiw@DOILAPTOP:~/UE_Parallel_Programming_2024/AS02$ g++ bank_system.cpp
dowiw@DOILAPTOP:~/UE_Parallel_Programming_2024/AS02$ ./a.out

Bank Account Management System menu:
1. Create new account.
2. Deposit.
3. Withdraw.
4. Display all accounts.
5. Apply interest (for savings).
6. Exit.

Enter a choice:
```

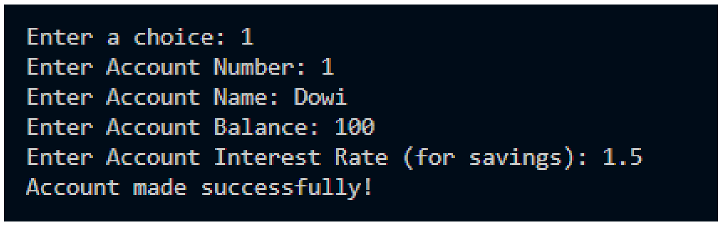


```
Enter a choice: 3.4
Enter a valid integer input!

Enter a choice: d
Enter a valid integer input!

Enter a choice: -1
Invalid choice! Try again (1 - 6).
```

You must enter **a valid choice**. This is checked if it is a string, negative input, float.

Assignment 2 Document

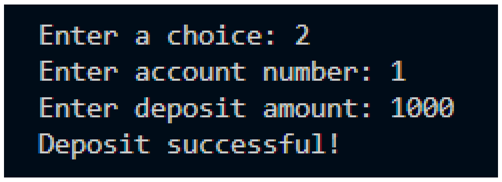
Choice 1:
You will be prompted to create a new account.



```
Enter a choice: 1
Enter Account Number: 1
Enter Account Name: Dowi
Enter Account Balance: 100
Enter Account Interest Rate (for savings): 1.5
Account made successfully!
```

- Account number is an int that is checked to be **only int and positive.**
- Account name is not checked for anything (string)
- Balance is checked to be **only positive and float.**
- Interest rates can be negative **but only float.**

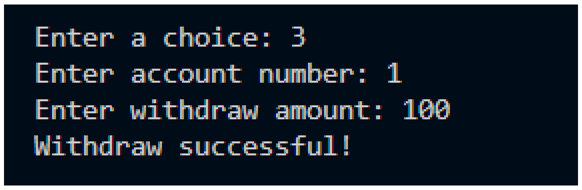
Choice 2:
You will be prompted to deposit an amount based on account number.



```
Enter a choice: 2
Enter account number: 1
Enter deposit amount: 1000
Deposit successful!
```

- Account number is checked for **only int and exists in runtime.**
- Deposit is checked for **only float.**

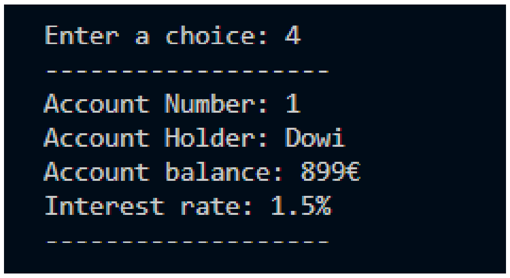
Choice 3:
You will be prompted to withdraw an amount.



```
Enter a choice: 3
Enter account number: 1
Enter withdraw amount: 100
Withdraw successful!
```

- Account number is checked for **only int and exists in runtime.**
- Withdraw is checked for **only float and balance is greater than amount. (not equal so that account has at least 1 euro in it)**

Choice 4:
The option will display every account made in runtime.

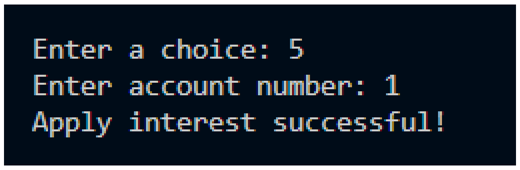


```
Enter a choice: 4
-------------------
Account Number: 1
Account Holder: Dowi
Account balance: 899€
Interest rate: 1.5%
-------------------
```

- Loops through the vector data structure using auto

Assignment 2 Document

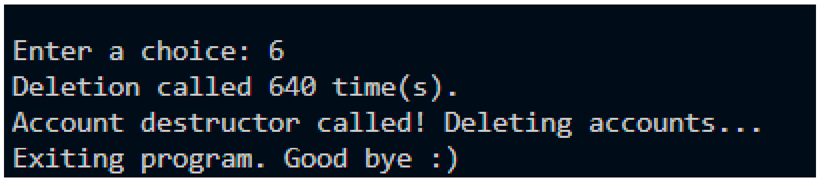
Choice 5:
You will be prompted for an account number to apply interest on.



```
Enter a choice: 5
Enter account number: 1
Apply interest successful!
```

- Account number is checked if **existing and int**.

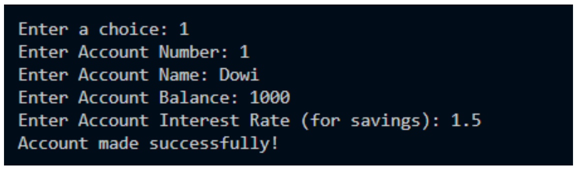
Choice 6:
Exit program. Displays the number of times the deletion is called.



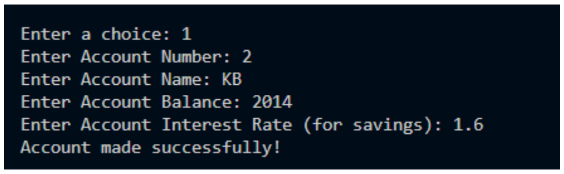
```
Enter a choice: 6
Deletion called 640 time(s).
Account destructor called! Deleting accounts...
Exiting program. Good bye :)
```

Sample tests:

- Create 2 savings accounts:

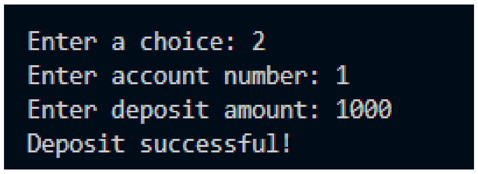


```
Enter a choice: 1
Enter Account Number: 1
Enter Account Name: Dowi
Enter Account Balance: 1000
Enter Account Interest Rate (for savings): 1.5
Account made successfully!
```

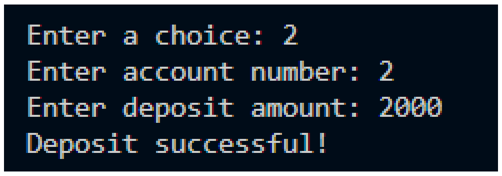


```
Enter a choice: 1
Enter Account Number: 2
Enter Account Name: KB
Enter Account Balance: 2014
Enter Account Interest Rate (for savings): 1.6
Account made successfully!
```

- Deposit 1000 and 2000 euro:

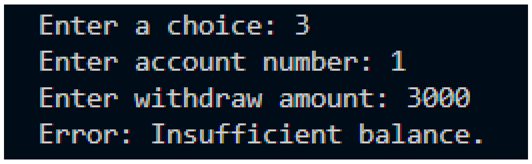


```
Enter a choice: 2
Enter account number: 1
Enter deposit amount: 1000
Deposit successful!
```

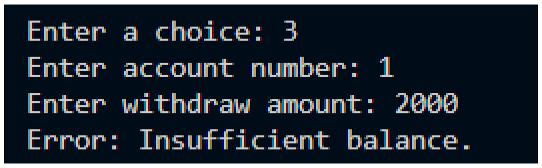


```
Enter a choice: 2
Enter account number: 2
Enter deposit amount: 2000
Deposit successful!
```

- Withdraw greater than balance (account one should have only 2000)

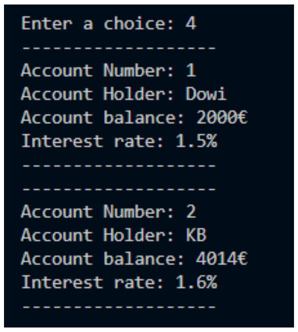


```
Enter a choice: 3
Enter account number: 1
Enter withdraw amount: 3000
Error: Insufficient balance.
```



```
Enter a choice: 3
Enter account number: 1
Enter withdraw amount: 2000
Error: Insufficient balance.
```

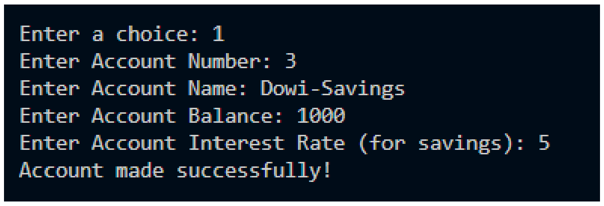
- Display all accounts:



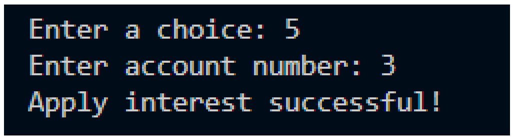
```
Enter a choice: 4
-------------------
Account Number: 1
Account Holder: Dowi
Account balance: 2000€
Interest rate: 1.5%
-------------------
-------------------
Account Number: 2
Account Holder: KB
Account balance: 4014€
Interest rate: 1.6%
-------------------
```

Assignment 2 Document

- Apply 5% interest

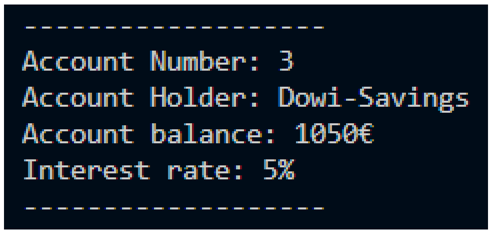


```
Enter a choice: 1
Enter Account Number: 3
Enter Account Name: Dowi-Savings
Enter Account Balance: 1000
Enter Account Interest Rate (for savings): 5
Account made successfully!
```



```
Enter a choice: 5
Enter account number: 3
Apply interest successful!
```

- Display updated value



```
--------------------
Account Number: 3
Account Holder: Dowi-Savings
Account balance: 1050€
Interest rate: 5%
--------------------
```

From 1000 to 1050.

Reflection:
Object oriented programming improved the structure of the program by allowing functions to be used within them. One very important thing I noticed is how it encapsulated data. While implementing the deposit and withdraw functions, I was not able to access accountNumber without making a getter. Other than that, it is a whole upgrade on function structures and how you can use them.

Exception handling made it easy to exit out of a function or a wrong switch case. This allowed for more thorough checking and made it easier to move across the program when the user inputs a wrong type or wrong specific input.