CS52 Lab 9: Exception Handling

Sources of Support

1. Pages 895 to 926 of Savitch 10th edition discuss Exceptions and Error handling.
2. Exception Handling PowerPoint available in canvas in this module.
3. An example is contained in this document
4. Savitch has videos describing Exception Handling, though in this assignment , you may do things a bit differently.

Exception handling is used in many different ways, but its use in three situations is well suited.

1. A member function of a class or a stand-alone function is required to return a value, but due to error condition it cannot do that. In that case a return by exception mechanism informs user of the error condition, and allows them to take corrective action by placing a catch block in the code.
2. A file or data source is opened either for reading or writing. The data source crashes or is not available due to hardware problems. In that case exception mechanism informs user of the problem and closes files safely, and releases resources that were in use.
3. Validation of data input.

There are other situations of course, for example out of bound array access, illegal casting operators etc., which can be handled by exception handling.

Exception Handling Use for Employee Class

UML diagram of Employee class is given below.

Notice that it has a field hoursWorked, which is number of hours worked by the employee in a week. A week contains (24*7 = 168), 168 hours. Obviously if user input for hours worked is greater than 168, then that is an error condition in the data input. We can also assume some other value of hoursWorked that is lesser than 168 as being in an error. In this situation we will just work with 168 hours.

The class also has a getInstance method which has the below header.

void getInstance (Employee & EMP);

This method takes user input for all employee data, including hoursWorked, from the keyboard. If user enters hours worked >168, then that is an exception/error condition that we need to correct. For that we need to write an exception class with name (related to the exception). We choose the class name to be: **HoursWorkedException**

It is important to name the exception handling classes, in a way that the name is closely related to the exception being handled.

Exception classes are similar to regular classes. They have data members, constructor(s) and generally only get methods. More often than not, the exception classes need not mutate, so they may not have set functions. We wish to design the HoursWorkedException class such that:

- It carries the invalid value of hoursWorked entered by the user.
- A message that can tell user the details of error.

Thus we add the following three data members to the **HoursWorkedException** class.

```
const static int NUM_HOURS_WEEK = 168;
int hoursWorked; // Number of hours worked inputted by the user
string Message; // Message to be displayed to the user when exception is handled
```

NUM_HOURS_WEEK stores the number of hours in a week. For class **HoursWorkedException**, we provide a dual role default and explicit constructor and getters for all three data members. The UML diagram for **HoursWorkedException class is shown below.**

The source code is shown in Table below.

Line #	Source code for class HoursWorkedException
1	class HoursWorkedException{
2	private:
3	const static int NUM_HOURS_WEEK = 168;
4	double hoursWorked;
5	string Message;
6	public:
7	HoursWorkedException(double hrs=0.0, string msg = "Message not set"):
8	hoursWorked(hrs),Message(msg){
9	//no code needed
10	}
11	double getHoursWorked() const{
12	return hoursWorked;
13	}
14	const string getMessage() const{
15	return Message;
16	}
17	static int getMaxHoursWeek(){
18	return NUM_HOURS_WEEK;
19	}
20	};

As mentioned earlier, the class has three data members, and three getters; one to return each of them. The default/explicit constructor initialize the hoursWorked and Message data members to default or user provided values.

Use of HoursWorkedException class inside the getInstance method of Employee class

If we wish to make the use of **HoursWorkedException** class, then an object of this class type need preparation and throwing at the appropriate location inside the getInstance method. To get the best use of this exception class, the code inside the getInstance has been rearranged. The code for modified getInstance is given in the table below.

Line #	Source code for modified getInstance in Employee class
1	void Employee::getInstance(Employee & EMP){
2	//why do we need below if structure code?
3	if(!cin){
4	return;
5	}
6	//Get input for data that can throw exception first so that user does not have to
7	//enter other data until data subject to exception is finished first
8	cout<<"Please enter employee hours worked [xx.yy] :";
9	double hrs;
10	cin>>hrs;
11	hrs = abs(hrs);
12	

```
13      if (hrs>HoursWorkedException::getMaxHoursWeek()) {
14         string msg= "Number of hours worked " + to_string(hrs)+
15         " exceeds total hours in a week: " +to_string(HoursWorkedException::getMaxHoursWeek())+
16         ".\n";
17         throw HoursWorkedException(hrs,msg);
18      }
19
20      cout<<"Please enter employee social security number [digits only]: ";
21      string ssn;
22      cin>>ssn;
23      cout<<"Please enter employee first name: ";
24      string first;
25      cin>>first;
26      cout<<"Please enter employee last name: ";
27      string last;
28      cin>>last;
29         cout<<"Please enter employee hourly pay rate [xx.yy] :";
30      double rate;
31      cin>>rate;
32      rate = abs(rate);
33      EMP = Employee(ssn,first,last,rate,hrs);
34   }
```

In above table, the user inputted hours worked in a week completes on line 11. Line 12 then codes a conditional if to test whether hrs(hours worked), exceeds the value HoursWorkedException::getMaxHoursWeek(), latter being 168. [See code for function getMaxHoursWeek in the class HoursWorkedException in previous table]. Exception condition exists if the condition

hrs>HoursWorkedException::getMaxHoursWeek())

evaluates to true. In that case the body of if structure prepares a string msg which contains the description of exception condition [Line 14 to 16]. Then the object of type HoursWorkedException is prepared and thrown. The throw clause will terminate the execution of the getInstance and return the object constructed on line 17 to the calling block.

Handling the Exception in main function

The table below shows the main function where exception handler HoursWorkedException is used to validate the user input for hours worked. The code that could throw an exception is placed inside a try block and catch block then gets the exception object, where it is used further.

Line #	Code in the main function which uses the Exception Handler [All includes have been done]
1	int main( ){
2	Employee EMP;
3	bool done = false;
4	
5	while (!done) {
6	try {
7	Employee::getInstance(EMP);
8	done = true;
9	} catch (HoursWorkedException ex) {

10	cout<<ex.getMessage()<<endl;
11	done = false;
12	}
13	}
14	cout<<"Data of employee you just entered:\n";
15	cout<<EMP.toString();
16	return 0;
17	}
	Typical output from the main function
18	Please enter employee hours worked [xx.yy] :200
19	Number of hours worked 200.000000 exceeds total hours in a week: 168.
20	
21	Please enter employee hours worked [xx.yy] :300
22	Number of hours worked 300.000000 exceeds total hours in a week: 168.
23	
24	Please enter employee hours worked [xx.yy] :55.0
25	Please enter employee social security number [digits only]: 11223344
26	Please enter employee first name: Sam
27	Please enter employee last name: Jones
28	Please enter employee hourly pay rate [xx.yy] :10.02
29	Data of employee you just entered:
30	Employee Social Security Number: 11223344
31	Employee Name: Sam Jones
32	Employee Hours Worked: 55.00 hours.
33	Employee pay rate: $10.02 per hour.
34	Employee salary: $626.25

In this code a flag controlled loop in addition to the try catch block are used for data validation for the hours worked. A flag controlled loop is started on line 5. Since getInstance can throw exception, the call to it is surrounded by a try block. If hours worked entered by the user exceeds 168, then line 17 of getInstance throws the exception object and rest of the try block is skipped. But since there is a catch block whose argument has the exception object that is thrown, we can make use of it to handle the invalid data input. When employee hour worked = 200 are entered (see output line 18 in above table), then exception object is thrown by the getInstance function. The line 8 inside the try block is skipped. the control then transfers to the line 10 in catch block. Inside this block we call the getMessage function of exception object ex. The output from this function is shown on line 19 of output part in above table. It clearly informs user that entering 200 hours is an error condition. Then line 11 sets done to false, and control returns to the top of the loop. The call to getInstance will keep repeating until the hours worked less than 168 are entered. That is the case when user inputs the hours worked = 55 as shown by the line 24 of output. In that case exception object is not thrown because in the getInstance function the condition

hrs>HoursWorkedException::getMaxHoursWeek()

evaluates to false . [See code of getInstance function]. The program is then able to complete the data input for rest of the employee and inputted employee is printed to console. [See code line 15, and output lines 30 to 34].

Statement of Work You Need to do

Write a BankAccount class, whose details are given below:

Detailed Description

Class **BankAccount** models a customer account in a bank. It has data members and their types as follows:

ActNum string type - Account number

LastName string type - Last name of account holder

FirstName string type - First name of account holder

balance double type - amount of money in the account

Class Invariant:

Class has an invariant that ***balance*** class data member cannot be negative. What that means is that invariant is established as soon as constructor finishes constructing object (meaning that call to constructor has ended). Also invariant is established before a mutator or destructor is called and remains established after the mutator call is over.

Member Data Documentation: Class must have following data members. Their access levels are noted inside square brackets.

string ActNum `[private]` Account number for this account.
double balance `[private]` Money in dollars in the account.
string FirstName `[private]` First name of the account holder
string LastName `[private]` Last name of the account holder

All constructors and member function are public.

Constructor Descriptions (Table below gives constructor descriptions)

BankAccount () Default constructor sets balance to zero, all string data members to "not set value". For example the data member ActNum will be set to "Account Number not set". Data member LastName will be set to "Last Name not set". Follow the same pattern for setting the first name.

BankAccount (const string & *actN*, const string & *fname*, const string & *lname*, double *bal*)

Explicit constructor sets all class data members to user defined explicit values. Obeys invariant and if balance is negative then informs user and sets the balance to zero. Member initialization list must be used.

Parameters:

actN is the account number *fname* is the customer first name *lname* is the customer last name *bal* is the customer balance in the account

Member Function Description (They are written alphabetically. However, you must code in the following order and do tests as you go along).

toString then mutators, then accessors. See details at the end about main function.

void deposit (double *money*)

Deposits the money into the account. Ascertains that value passed to this mutator is not negative. Updates the balance if money being deposited is non-zero and positive. No balance update is done otherwise. User is informed accordingly.

Parameters: *money* is the amount that customer is seeking to deposit.

string getAccountNumber () const

Accessor member function getAccountNumber returns the account number.

Returns:

the account number.

double getBalance () const

Accessor member function getBalance returns the current balance in the account.

Returns: the current balance.

const string getFirstName () const

Accessor member function getFirstName returns the first name of account holder.

Returns: the first name of the account holder.

const string getFullName () const

Accessor member function getFullName returns the full name of account holder.

Returns: the full name of the account holder.

const string getLastName () const

Accessor member function getLastName returns the last name of account holder.

Returns: the last name of the account holder.

void setAccountNumber (const string & *actN*)

Sets the account number to a new value.

Parameters: *actN* is the new account number.

void setLastName (const string & *lname*)

Sets the customer last name to a new value.

Parameters: *lname* is the new last name for the account holder.

const string toString() const

Returns the string version of the BankAccount object such that apart from printing full name, account number, the toString function also contains current balance shown to 2 decimal places only and having the dollar sign before the balance. The comma separator for digits for currency is not required, though C++11 has a mechanism to show that.

[See relevant video on my channel on youtube.]

void withdraw (double *money*)

Performs the withdrawal of the money from account so that invariant is obeyed. Therefore by the time action of member function withdrawal is complete the balance data member obeys the invariant. Gives proper message if the program attempts to violate the invariant.

Parameters: *money* is the amount that customer is seeking to withdraw.

void print(ostream & out=cout) const

Prints all details of this one bank account to console or output file as the case may be.
Caution: Default argument is not part of the header in the implementation file.

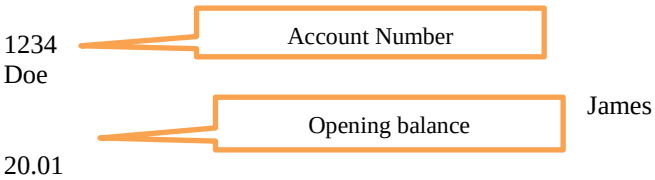
static void getInstance(BankAccount & BA)

The static function getInstance above gets an unfilled BankAccount object BA passed by reference and is filled by this function from keyboard data entry by the user. For model of this function, look up the similar function in Employee class code given to you as a paper copy and also available in content area of D2L.

static void getInstance (Bank ccount & BA, ifstream & in);

The static function getInstance above gets an unfilled BankAccount object BA passed by reference and is filled by this function from the input file bonded to ifstream object in. The input file format is below:

Account Number
Last Name
First Name
Starting Balance
Typical data for James Doe in the input file is shown below.



For model of this function, look up the similar function in Employee class code given to you as a paper copy and also available in content area of D2L.

Main Function Description

First write a main function that makes calls to all constructors, mutators, and accessors so that all lines of code are executed. Verify from output that all member functions are working correctly. Then rename this main function as main1. This is because at one time you can only have one function with name main in a C++ program. Now write the other main function, where you use a loop to present a menu to do followings (this will work similar to an ATM machine):

1. Create an account from keyboard data entry
2. Create an account from input file data entry
3. Print account balance to console. [Must have $ sign and 2 digits in output].
4. Withdraw money
5. Deposit money
6. Print account details to console
7. Print account details to an output file.
8. Print full name of account holder to console only.
9. Print account number only of the account holder to console.
10. Exit

Each time menu is presented to the user, user can choose any of the menu items and after completion of that item the menu is presented again. If user tries to use menu items other than 1 and 2 before using 1 and 2, the program should tell them that an instance of BankAccount is yet to be created. This can be done using some Boolean flags.

For BankAccount class one special situation that also deserves exception handling is in the getInstance method which has the following version.

```
static void getInstance(BankAccount & BA){

}
```

Understand that in this version at the end of the function, the constructor construct an object of BankAccount type which must be a legal object with balance > 0.0. It is reading data from the keyboard. What if user enters an account balance that is zero or negative? Obviously, the bank cannot afford to start an account with zero or negative balance. Thus you need to write an exception class to handle that situation. The class UML diagram of this class is given below.

The class has two data members. The double initBalance will store the initial balance entered by the user to construct a BankAccount object when getInstance function is called. Message stores the message to be displayed to the user when catch block executes. The class has two getters, one for each data member. The class also has the constructor, which folds both default and explicit constructors into one. That means that default values are provided for both constructor arguments.

Proceed as Follows

1. Create a C++ project.
2. For sake of simplicity you could create a single file. But if you like to create multiple files then its ok. In that case all your files must be zipped into one folder before uploading to Canvas.
3. Create BankAccount Class and main function. Confirm that main function and the class works.
4. Code the full definition of class ZeroOrNegativeInitialBalanceException. See the model of Exception class I did for the Employee class in the table in previous pages.
5. Modify the getInstance (BankAccount & BA) in your implementation of this function that function throws the object of type ZeroOrNegativeInitialBalanceException when user enters the 0 or a negative value for the initial balance. You can look at the mode of getInstance I did for Employee in the table in previous pages. Rest of the function getInstance remains unchanged.
6. Write the main function now so that inside the flag controlled loop and inside the try block the BankAccount getInstance that you modified is called. You can look for the model of main function in the table in previous pages where I applied it to Employee class.
7. Add the appropriate catch block.
8. The typical output from my program is given below. Your output should be qualitatively similar.

Enter the balance or initial deposit: -2.0

Initial balance of negative dollars is not allowed.

Enter the balance or initial deposit: 0.0

Initial balance of zero dollars is insufficient

Enter the balance or initial deposit: 200.0

Enter the new (unique) account number: 1234

Enter the first name on account: Sam

Enter the last name on account: Jones

Bank Account object created:

Account Number: 1234 Name: Sam Jones Balance: $200.00