

**CC-211L**

**Object Oriented Programming**

**Laboratory 14**

**Random-Access Files & Exception Handling**

**Version: 1.0.0**

**Release Date: 04-05-2023**

**Department of Information Technology**

**University of the Punjab**

**Lahore, Pakistan**

**Contents:**

- Learning Objectives
- Required Resources
- General Instructions
- Background and Overview
  - Files and Streams
  - Random-Access Files
  - Exception
  - Exception Handling
- Activities
  - Pre-Lab Activity
    - File Position Pointers
    - Task 01
  - In-Lab Activity
    - Creating a Random-Access File
    - Writing data to a Random File
    - Reading data from a Random File
    - Exception Handling
      - C++ try
      - C++ catch
      - C++ throw
    - Standard Exceptions
    - User Defined Exceptions
    - Rethrowing an Exception
    - Constructor Destructor and Exception Handling
    - Task 01
    - Task 02
    - Task 03
  - Post-Lab Activity
    - Task 01
- Submissions
- Evaluations Metric
- References and Additional Material
- Lab Time and Activity Simulation Log

## Learning Objectives:

- Random-Access File
- Create Random-Access File
- Write to a Random-Access File
- Read from a Random-Access File
- Defining an Exception Class
- Rethrowing an Exception
- Constructor, Destructor and Exception handling

## Resources Required:

- Desktop Computer or Laptop
- Microsoft ® Visual Studio 2022

## General Instructions:

- In this Lab, you are **NOT** allowed to discuss your solution with your colleagues, even not allowed to ask how is s/he doing, this may result in negative marking. You can **ONLY** discuss with your Teaching Assistants (TAs) or Lab Instructor.
- Your TAs will be available in the Lab for your help. Alternatively, you can send your queries via email to one of the followings.

Teachers:		
Course Instructor	Prof. Dr. Syed Waqar ul Qounain	<a href="mailto:swjaffry@pucit.edu.pk">swjaffry@pucit.edu.pk</a>
Lab Instructor	Azka Saddiqa	<a href="mailto:azka.saddiqa@pucit.edu.pk">azka.saddiqa@pucit.edu.pk</a>
Teacher Assistants	Saad Rahman	<a href="mailto:bsef19m021@pucit.edu.pk">bsef19m021@pucit.edu.pk</a>
	Zain Ali Shan	<a href="mailto:bcsf19a022@pucit.edu.pk">bcsf19a022@pucit.edu.pk</a>

## **Background and Overview:**

### **Files and Streams:**

Files are used to store data in a storage device permanently. File handling provides a mechanism to store the output of a program in a file and to perform various operations on it.

A stream is an abstraction that represents a device on which operations of input and output are performed. A stream can be represented as a source or destination of characters of indefinite length depending on its usage.

### **Random-Access File:**

Random file access enables us to read or write any data in our disk file without having to read or write every piece of data before it while in Sequential files to reach data in the middle of the file you must go through all the data that precedes it. We can quickly search for data, modify data, delete data in a random-access file.

### **Exception:**

An exception is a problem that arises during the execution of a program. It is a response to an exceptional circumstance that arises while a program is running, such as an attempt to divide by zero.

### **Exception Handling:**

Exception handling is the process of handling errors and exceptions in such a way that they do not hinder normal execution of the system. For example, User divides a number by zero, this will compile successfully but an exception or run time error will occur due to which our applications will be crashed.

## Activities:

### Pre-Lab Activities:

#### File Position Pointers:

<istream> and <ostream> classes provide member functions for repositioning the file pointer (the byte number of the next byte in the file to be read or to be written). These member functions are:

- seekg (seek get) for istream class
- seekp (seek put) for ostream class

Following are some examples to move a file pointer:

- **inClientFile.seekg(0)** - repositions the file get pointer to the beginning of the file
- **inClientFile.seekg(n, ios::beg)** - repositions the file get pointer to the n-th byte of the file
- **inClientFile.seekg(m, ios::end)** - repositions the file get pointer to the m-th byte from the end of file
- **inClientFile.seekg(0, ios::end)** - repositions the file get pointer to the end of the file

The same operations can be performed with <ostream> function member seekp.

#### Member functions tellg() and tellp():

Member functions tellg and tellp are provided to return the current locations of the get and put pointers, respectively.

**long location = inClientFile.tellg();**

To move the pointer relative to the current location use ios::cur

**inClientFile.seekg(n, ios::cur)** - moves the file get pointer n bytes forward.

#### Example:

In this example, we open a file named “example.txt” for writing using std::ofstream. We then write some data to the file using the << operator. We use the tellp() function to get the current position of the file pointer and use seekp() to move the file pointer back to the beginning of the file. We then write some more data to the file and use seekp() again to move the file pointer to the end of the file. Finally, we write some more data to the file and close it.

```

1  #include <iostream>
2  #include <fstream>
3
4  int main() {
5      std::ofstream file("example.txt");
6      // Write some data to the file
7      file << "Hello, world!" << std::endl;
8      // Get the current position of the file pointer
9      std::ostream::pos_type pos = file.tellp();
10     // Move the file pointer back to the beginning of the file
11     file.seekp(0);
12     // Write some more data to the file
13     file << "This is a test" << std::endl;
14     // Move the file pointer to the end of the file
15     file.seekp(pos);
16     // Write some more data to the file
17     file << "The end" << std::endl;
18     // Clean up
19     file.close();
20     return 0;
21 }
```

Fig. 01 (File Position Pointer)

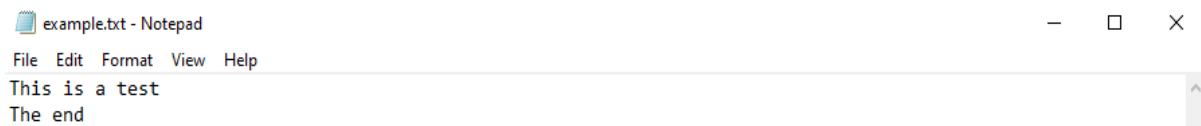
**File Content:**

Fig. 02 (File Position Pointer)

**Example:**

In this example, we open a file named "example.txt" and use the `tellg()` function to get the current position of the file pointer. We then use the `seekg()` function to move the file pointer to the end of the file and use `tellg()` again to get the size of the file. We then move the file pointer back to the beginning of the file using `seekg()` and read the data from the file using the `read()` function. Finally, we output the data to the console and close the file.

```
1 #include <iostream>
2 #include <fstream>
3 #include <string>
4 int main() {
5     std::ifstream file("example.txt");
6     // Get the current position of the file pointer
7     std::istream::pos_type pos = file.tellg();
8     // Move the file pointer to the end of the file
9     file.seekg(0, std::ios::end);
10    // Get the size of the file
11    std::istream::pos_type size = file.tellg();
12    // Move the file pointer back to the beginning of the file
13    file.seekg(pos);
14    // Read data from the file
15    std::string buffer;
16    while (file) {
17        getline(file, buffer);
18        // Output the data to the console
19        std::cout << buffer << std::endl;
20    }
21    // Clean up
22    file.close();
23    return 0;
24 }
```

Fig. 03 (File Position Pointer)

**Output:**

Fig. 04 (File Position Pointer)

**Task 01: Read Records****[Estimated time 30 minutes / 20 marks]**

Write a program that:

- Reads in a text file called “**data.txt**” containing records(objects) of the class “**Record**” with data members:
  - Id (int)
  - Name (string)
  - Balance (double)
- The program should then allow the user to search for a record by ID and update the balance for that record
- Use seekg() and seekp() functions to move the read and write positions to the correct locations in the file

## In-Lab Activities:

### Creating a Random-Access File:

Instant access is possible with random access files. Individual records of a random-access file can be accessed directly without searching many other records.

#### Example:

```

1  #include <iostream>
2  #include <fstream>
3  using namespace std;
4  struct clientData {
5      int accountNumber;
6      char lastName[15];
7      char firstName[10];
8      float balance;
9  };
10 int main()
11 {
12     ofstream outCredit("credit1.dat", ios::out);
13     if (!outCredit) {
14         cerr << "File could not be opened." << endl;
15         exit(1);
16     }
17     clientData blankClient = { 0, "", "", 0.0 };
18     for (int i = 0; i < 100; i++)
19         outCredit.write
20             (reinterpret_cast<const char*>(&blankClient),
21              sizeof(clientData));
22     return 0;
23 }
```

Fig. 05 (Creating a Random-Access File)

The `<ostream>` member function `write` outputs a fixed number of bytes beginning at a specific location in memory to the specific stream. When the stream is associated with a file, the data is written beginning at the location in the file specified by the “put” file pointer. The `write` function expects a first argument of type “`const char *`”, hence we used the `reinterpret_cast` to convert the address of the `blankClient` to a `const char *`. The second argument of `write` is an integer of type “`size_t`” specifying the number of bytes to written. Thus, the `sizeof (clientData)`.

### Writing Data to a Random File:

#### Example:

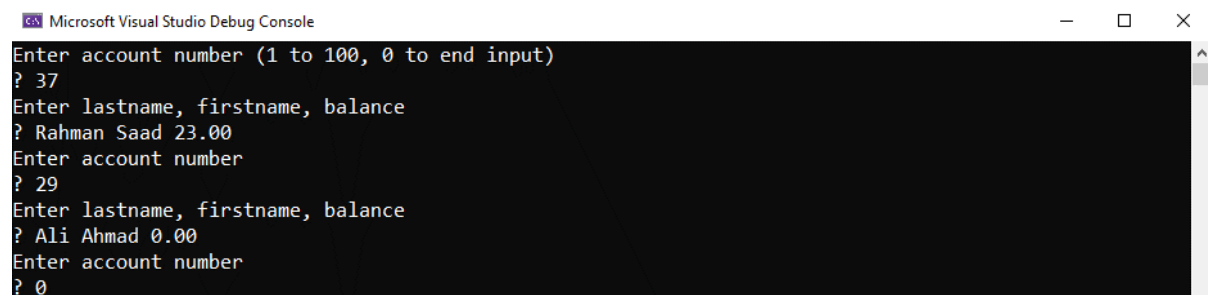


```

1  #include <iostream>
2  #include <fstream>
3  using namespace std;
4  struct clientData {
5      int accountNumber;
6      char lastName[15];
7      char firstName[10];
8      float balance;
9  };
10 int main(){
11     ofstream outCredit("credit.dat", ios::ate);
12     if (!outCredit) {
13         cerr << "File could not be opened." << endl;
14         exit(1);
15     }
16     cout << "Enter account number (1 to 100, 0 to end input)\n? ";
17     clientData client;
18     cin >> client.accountNumber;
19     while (client.accountNumber > 0 &&
20           client.accountNumber <= 100) {
21         cout << "Enter lastname, firstname, balance\n? ";
22         cin >> client.lastName >> client.firstName >> client.balance;
23         outCredit.seekp((client.accountNumber - 1) * sizeof(clientData));
24         outCredit.write(reinterpret_cast<const char*>(&client), sizeof(clientData));
25         cout << "Enter account number\n? ";
26         cin >> client.accountNumber;
27     }
28     return 0;
29 }

```

Fig. 06 (Writing Data to a Random-Access File)

**Output:**


```

Microsoft Visual Studio Debug Console
Enter account number (1 to 100, 0 to end input)
? 37
Enter lastname, firstname, balance
? Rahman Saad 23.00
Enter account number
? 29
Enter lastname, firstname, balance
? Ali Ahmad 0.00
Enter account number
? 0

```

Fig. 07 (Writing Data to a Random-Access File)

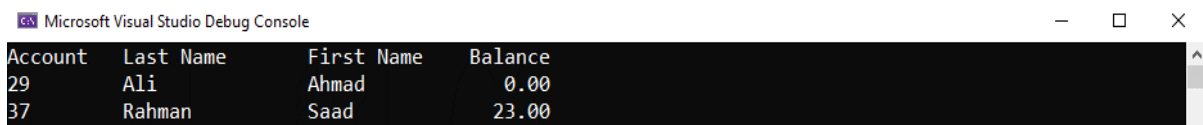
**Reading Data from a Random File:****Example:**

```

1  #include <iostream>
2  #include <iomanip>
3  #include <fstream>
4  using namespace std;
5  void outputLine(ostream&, const clientData&);
6  int main(){
7      ifstream inCredit("credit.dat", ios::in);
8      if (!inCredit) {
9          cerr << "File could not be opened." << endl;
10         exit(1);
11     }
12     cout << setiosflags(ios::left) << setw(10) << "Account" << setw(16) << "Last Name"
13     << setw(11) << "First Name" << resetiosflags(ios::left) << setw(10) << "Balance" << endl;
14     clientData client;
15     inCredit.read(reinterpret_cast<char*>(&client), sizeof(clientData));
16     while (inCredit && !inCredit.eof()) {
17         if (client.accountNumber != 0)
18             outputLine(cout, client);
19         inCredit.read(reinterpret_cast<char*>(&client), sizeof(clientData));
20     }
21     return 0;
22 }
23 void outputLine(ostream& output, const clientData& c){
24     output << setiosflags(ios::left) << setw(10) << c.accountNumber << setw(16) << c.lastName
25     << setw(11) << c.firstName << setw(10) << setprecision(2) << resetiosflags(ios::left)
26     << setiosflags(ios::fixed | ios::showpoint) << c.balance << '\n';
27 }

```

Fig. 08 (Reading Data from a Random-Access File)

**Output:**


Account	Last Name	First Name	Balance
29	Ali	Ahmad	0.00
37	Rahman	Saad	23.00

Fig. 09 (Reading Data from a Random-Access File)

**Exception Handling:**

Exception handling provides you with a way of handling unexpected circumstances like runtime errors. So, whenever an unexpected circumstance occurs, the program control is transferred to special functions known as handlers.

To catch the exceptions, you place some section of code under exception inspection. The exception handling is mainly performed using three keywords namely - try, catch, and throw using following syntax:

```

try
{
    // code
    throw exception;
}
catch(exception e)
{
    // code for handling exception
}

```

**C++ try:**

The try block is used to keep the code that is expected to throw some exception. Whenever our code leads to any exception or error, the exception or error gets caught in the catch block. In simple terms,

we can say that the try block is used to define the block of code that needs to be tested for errors while it is being executed.

E.g., Suppose we are dealing with databases, we should put the code that is handling the database connection inside a try block as the database connection may raise some exceptions or errors.

### C++ catch:

The catch block is used to catch and handle the error(s) thrown from the try block. If there are multiple exceptions thrown from the try block, then we can use multiple catch blocks after the try blocks for each exception. In this way, we can perform different actions for the various occurring exceptions. In simple terms, we can say that the catch block is used to define a block of code to be executed if an error occurs in the try block.

E.g., Let us take the same above example that we are dealing with the database. Now, if during the connection, there is an exception raised inside the try block, then there should be a catch block present to catch or accept the exception and handle the exception. The catch block ensures that the normal flow of the code is not halted.

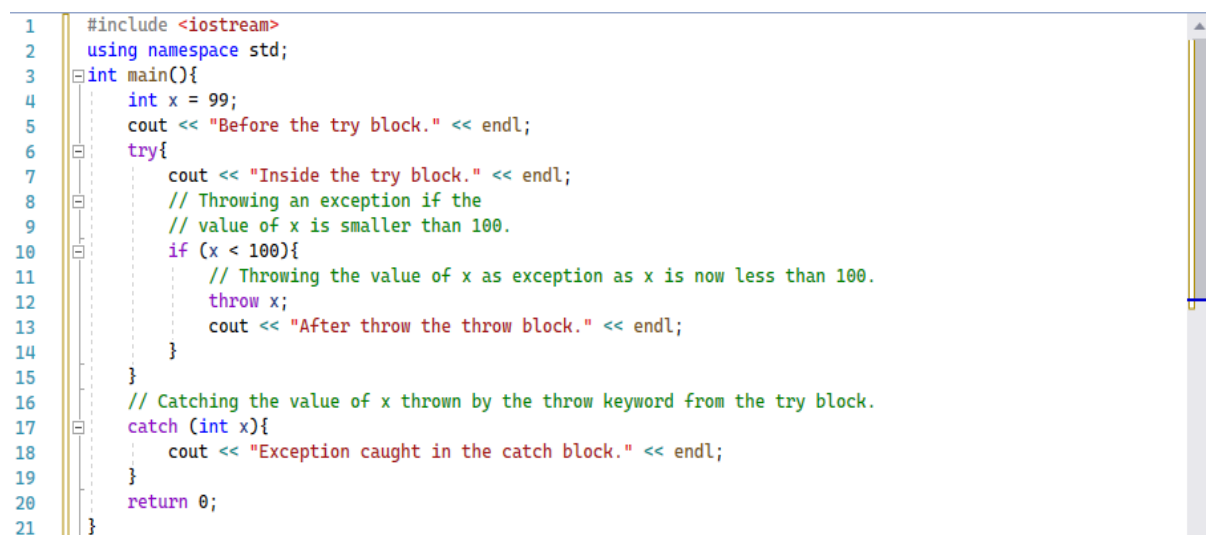
### C++ throw:

The throw block is used to throw exceptions to the exception handler which further communicates the error. The type of exception thrown should be same in the catch block. The throw keyword accepts one parameter which is passed to the exception handler. We can throw both pre-defined as well as custom exception(s) as per the requirements.

Whenever we want to explicitly throw an exception, we use the throw keyword. The throw keyword is also used to generate the custom exception.

Following example demonstrates the overall working and syntax of try, catch, and throw in exception handling.

### Example:



```
1  #include <iostream>
2  using namespace std;
3  int main(){
4      int x = 99;
5      cout << "Before the try block." << endl;
6      try{
7          cout << "Inside the try block." << endl;
8          // Throwing an exception if the
9          // value of x is smaller than 100.
10         if (x < 100){
11             // Throwing the value of x as exception as x is now less than 100.
12             throw x;
13             cout << "After throw the throw block." << endl;
14         }
15     }
16     // Catching the value of x thrown by the throw keyword from the try block.
17     catch (int x){
18         cout << "Exception caught in the catch block." << endl;
19     }
20     return 0;
21 }
```

Fig. 10 (Exception Handling)

### Output:



Fig. 11 (Exception Handling)

If you do not know the type of throw used in the try block, we can always use the "three dots" syntax (...) inside the catch block, which will handle any type of exception.

### Example:

```

1  #include <iostream>
2  using namespace std;
3  int main(){
4      try{
5          int age = 25;
6          if (age <= 18){
7              cout << "Access denied.";
8          }
9          else{
10             // throwing any random value as exception as age is less than 18.
11             throw 505;
12         }
13     }
14     // Catching the thrown exception and displaying access denied!
15     catch (...){
16         cout << "Access denied!" << endl;
17     }
18     return 0;
19 }
```

Fig. 12 (Exception Handling)

### Output:



Fig. 13 (Exception Handling)

### Standard Exceptions:

C++ comes with a list of standard exceptions defined in <exception> class given in the table below:

Exception	Description
exception	This is an exception and the parent class of all standard C++ exceptions.
bad_alloc	This exception is thrown by a new keyword.
bad_cast	This is an exception thrown by dynamic_cast.
bad_exception	A useful device for handling unexpected exceptions in C++ programs.
bad_typeid	An exception thrown by typeid.
logic_error	This exception is theoretically detectable by reading code.
domain_error	This is an exception thrown after using a mathematically invalid domain.
invalid_argument	An exception thrown for using invalid arguments.
length_error	An exception thrown after creating a big std::string.
out_of_range	Thrown by at method.
runtime_error	This is an exception that cannot be detected via reading the code.
overflow_error	This exception is thrown after the occurrence of a mathematical overflow.
range_error	This exception is thrown when you attempt to store an out-of-range value.
underflow_error	An exception thrown after the occurrence of mathematical underflow.

**User Defined Exceptions:**

There may be situations where you want to generate some user/program specific exceptions which are not pre-defined. In such cases C++ provided us with the mechanism to create our own exceptions by inheriting the exception class and overriding its functionality according to our needs.

**Example:**

```

1  #include <iostream>
2  #include <exception>
3  using namespace std;
4  class OverSpeed : public exception {
5      int speed;
6  public:
7      const char* what() {
8          return "Check out ur car speed.";
9      }
10 };
11 int main(){
12     int carspeed = 0;
13     try{
14         while (1){
15             carspeed += 10;
16             if (carspeed > 100){
17
18                 OverSpeed s;
19                 throw s;
20             }
21             cout << "Carspeed: " << carspeed << endl;
22         }
23     }
24     catch (OverSpeed ex)
25     {
26         cout << ex.what();
27     }
28     return 0;
29 }
30

```

Fig. 14 (User Defined Exception)

**Output:**

```

Microsoft Visual Studio Debug Console
Carspeed: 10
Carspeed: 20
Carspeed: 30
Carspeed: 40
Carspeed: 50
Carspeed: 60
Carspeed: 70
Carspeed: 80
Carspeed: 90
Carspeed: 100
Check out ur car speed.

```

Fig. 15 (User Defined Exception)

**Rethrowing an Exception:**

In the program execution, when an exception received by catch block is passed to another exception handler then such situation is referred to as rethrowing of exception.

This is done with the help of following statement,

**throw;**

The above statement does not contain any arguments. This statement throws the exception to next try catch block.

### Example:

```

1  #include<iostream>
2  using namespace std;
3
4  void subtract(int p, int q){ //A function called subtract with two arguments is defined
5      cout << "The subtract() function contains\n";
6      try{ //try block1
7          if (p == 0) //This condition checks whether p is equal to zero or not.
8              //if yes throw an exception else perform subtraction
9              throw p; //This statement throws an exception
10         else
11             cout << "The result of subtraction is:" << p - q << "\n";
12     }
13     catch (int){ //This statement catches the exception throw it again to the next try block
14         cout << "NULL value is caught\n";
15         throw;
16     }
17     cout << "End of subtract()\n\n";
18 }
19 int main(){
20     cout << "The main() function contains\n";
21     try{
22         subtract(5, 3); //passing integer values to subtract
23         subtract(0, 2);
24     }
25     catch (int){ //This statement catches the rethrown exception
26         cout << "NULL value caught inside main()\n";
27     }
28     cout << "End of main() function \n";
29     return 0;
30 }

```

Fig. 16 (Rethrowing Exception)

### Output:



```

Microsoft Visual Studio Debug Console
The main() function contains
The subtract() function contains
The result of subtraction is:2
End of subtract()

The subtract() function contains
NULL value is caught
NULL value caught inside main()
End of main() function

```

Fig. 17 (Rethrowing Exception)

### Constructor Destructor and Exception Handling:

Consider the following example:

### Example:

```
1  #include <iostream>
2  using namespace std;
3
4  class Test {
5  public:
6      //Constructor
7      Test() { cout << "Constructing an object of Test class" << endl; }
8      //Destructor
9      ~Test() { cout << "Destructing an object of Test class " << endl; }
10 };
11 int main() {
12     try {
13         Test t1;
14         //Throw passes an argument of the exception handler.
15         //Here int 10 is passed as an argument.
16         throw 10;
17     }
18     catch (int i) {
19         cout << "Caught " << i << endl;
20     }
21 }
```

Fig. 18 (Constructor Destructor and Exception Handling)

**Output:**

```
Microsoft Visual Studio Debug Console
Constructing an object of Test class
Destructing an object of Test class
Caught 10
```

Fig. 19 (Constructor Destructor and Exception Handling)

**Explanation:**

When an exception is thrown, the objects' destructors (whose scope ends with the try block) are automatically triggered before the catch block is performed. As a result, the preceding software outputs "Destructing an item of Test" before "Caught 10".

This happens when an exception is thrown from a Constructor:

**Example:**

```

1  #include <iostream>
2  using namespace std;
3  class Test1 {
4  public:
5      Test1() { cout << "Constructing an Object of Test1" << endl; }
6      ~Test1() { cout << "Destructing an Object of Test1" << endl; }
7  };
8  class Test2 {
9  public:
10     // Following constructor throws an integer exception
11     Test2() {
12         cout << "Constructing an Object of Test2" << endl;
13         throw 10;
14     }
15     ~Test2() { cout << "Destructing an Object of Test2" << endl; }
16 };
17 int main() {
18     try {
19         Test1 t1; // Constructed and destructed
20         Test2 t2; // Partially constructed
21         Test1 t3; // t3 is not constructed as this statement never gets executed
22     }
23     catch (int i) {
24         cout << "Caught " << i << endl;
25     }
26 }

```

Fig. 20 (Constructor Destructor and Exception Handling)

**Output:**

Microsoft Visual Studio Debug Console

```

Constructing an Object of Test1
Constructing an Object of Test2
Destructing an Object of Test1
Caught 10

```

Fig. 21 (Constructor Destructor and Exception Handling)

**Explanation:**

Destructors are only invoked for properly developed constructed objects. When a function object throws an exception, the object's destructor is not invoked.



**Task 01: Employee Database****[Estimated time 40 minutes / 30 marks]**

- Write a program that implements a database of employee records using a random-access file called **“employees.dat”**.
- Each record in the file should contain the following fields:  

```
class Employee {  
    int id;  
    char name[50];  
    char department[50];  
    float salary;  
    int numSubordinates;  
    int* subordinateIds; // dynamically allocated array of length numSubordinates  
};
```
- The subordinateIds array stores the IDs of each employee's subordinates, with the length of the array specified by the numSubordinates field.
- The program should allow the user to perform the following operations:
  - Add a new employee record to the file at the end.
  - Delete an employee record from the file based on their ID.
  - Update the fields of an employee record in the file based on their ID.
  - Print the information of an employee record in the file based on their ID.
  - Print the information of all employee records in the file, sorted by salary in ascending order.

**Task 02: Shopping Cart****[Estimated time 40 minutes / 30 marks]**

- Write a program that simulates a simple shopping cart system. The program consists of an Item class with data members:
  - Name
  - Price
  - Total quantity
- The program should allow the user to:
  - Add items to their cart
  - Remove items from their cart
  - View the contents of their cart
  - Calculate the total cost of the items in their cart
- The program should handle any errors that may occur during input or calculation, such as:
  - Invalid input
  - Out-of-stock items
  - Insufficient funds.

**Task 03: Sort Names****[Estimated time 30 minutes / 20 marks]**

- Create a program that reads in a list of names from a file and sorts them alphabetically.
- If any of the names in the file contain a non-alphabetic character, the program should throw a custom exception called **“InvalidNameException”**.
- To accomplish this task, you'll need to create a **“FileReader”** class that reads the contents of the file. Inside the FileReader class, you should define a member function called **“readNames”** that takes the file name as a parameter and reads the names from the file one by one. If any of the names contain a non-alphabetic character, the FileReader class should throw the InvalidNameException.
- Next, you'll need to create a **“NameSorter”** class that takes the list of names as a parameter and sorts them alphabetically. If any of the name length is greater than 15, the NameSorter class should throw the custom InvalidLength exception.

**Post-Lab Activities:****Task 01: Inventory Management****[Estimated time 60 minutes / 50 marks]**

- Create a program that simulates a simple inventory management system. The program should be able to read in a list of items and their corresponding quantity from a file, allow the user to add or remove items from the inventory, and output the updated inventory to a new file.
- The detailed description of the requirements are:
  - The program should read in the inventory information from a file. Each line of the file should contain the following information:
    - Item name (string)
    - Item quantity (integer)
  - The program should then display the inventory information to the user, along with options to add or remove items from the inventory.
  - If the user chooses to add an item to the inventory, they should be prompted to enter the item name and quantity. The program should then update the inventory accordingly.
  - If the user chooses to remove an item from the inventory, they should be prompted to enter the item name. The program should then update the inventory accordingly, but only if there are enough items in the inventory to remove.
  - If the user enters an invalid item name, the program should throw a custom exception that informs the user of the error and prompts them to try again.
  - After each modification to the inventory, the program should write the updated inventory information to a new file using random-access file handling.
  - The program should continue to display the updated inventory and prompt the user for actions until the user chooses to exit the program.

**Submissions:**

- For In-Lab Activity:
  - Save the files on your PC.
  - TA's will evaluate the tasks offline.
- For Pre-Lab & Post-Lab Activity:
  - Submit the .cpp file on Google Classroom and name it to your roll no.

**Evaluations Metric:**

- All the lab tasks will be evaluated offline by TA's
- **Division of Pre-Lab marks:** [20 marks]
  - Task 01: Read Records [20 marks]
- **Division of In-Lab marks:** [80 marks]
  - Task 01: Employee Database [30 marks]
  - Task 02: Shopping Cart [30 marks]
  - Task 03: Sort Names [20 marks]
- **Division of Post-Lab marks:** [50 marks]
  - Task 01: Inventory Management [50 marks]

**References and Additional Material:**

- **C++ Files and Streams**  
[https://www.tutorialspoint.com/cplusplus/cpp\\_files\\_streams.htm](https://www.tutorialspoint.com/cplusplus/cpp_files_streams.htm)
- **Exception Handling**  
[https://www.tutorialspoint.com/cplusplus/cpp\\_exceptions\\_handling.htm](https://www.tutorialspoint.com/cplusplus/cpp_exceptions_handling.htm)
- **Rethrowing Exceptions**  
<https://www.learncpp.com/cpp-tutorial/rethrowing-exceptions/>
- **Exception Handling and Object Destruction**  
<https://www.geeksforgeeks.org/exception-handling-and-object-destruction-in-cpp/>

**Lab Time Activity Simulation Log:**

- Slot – 01 – 00:00 – 00:15: Class Settlement
- Slot – 02 – 00:15 – 00:40: In-Lab Task
- Slot – 03 – 00:40 – 01:20: In-Lab Task
- Slot – 04 – 01:20 – 02:20: In-Lab Task
- Slot – 05 – 02:20 – 02:45: Evaluation of Lab Tasks
- Slot – 06 – 02:45 – 03:00: Discussion on Post-Lab Task