**Lab 12**

**File I/O**

**The C++ programming skills that should be acquired in this lab:**

* To understand the concept of saving data to files on disk
* To understand the syntax of opening and closing a file.
* To learn data I/O using **text** and **binary** files.

**File I/O**

C++ provides the following classes to perform output and input of characters to/from files:

* **ofstream:** Stream class to write on files
* **ifstream:** Stream class to read from files
* **fstream:** Stream class to both read and write from/to files.

These classes are derived directly or indirectly from the classes istream, and ostream. We have already used objects whose types were these classes: cin is an object of class istream and cout is an object of class ostream. Therfore, we have already been using classes that are related to our file streams. And in fact, we can use our file streams the same way we are already used to use cin and cout, with the only difference that we have to associate these streams with physical files.

**Example 12.1**

|  |  |
| --- | --- |
|  | // basic file operations  #include <iostream>  #include <fstream>  using namespace std;  int main () {  ofstream myfile;  myfile.open ("example.txt");  myfile << "Writing this to a file.\n";  myfile.close();  return 0;  } |
|  | [file example.txt]  Writing this to a file. |

This code creates a file called example.txt and inserts a sentence into it in the same way we are used to do with cout, but using the file stream myfile instead.

**Opening a File**

The first operation generally performed on an object of one of these classes is to associate it to a real file. This procedure is known as to *open a file*. An open file is represented within a program by a stream object (an instantiation of one of these classes, in the previous example this was myfile) and any input or output operation performed on this stream object will be applied to the physical file associated to it.

In order to open a file with a stream object we use its member function open():

***open (filename, mode);***

Where filename is a null-terminated character sequence of type const char \* (the same type that string literals have) representing the name of the file to be opened, and mode is an optional parameter with a combination of the following flags:

|  |  |
| --- | --- |
| ios::in | Open for input operations. |
| ios::out | Open for output operations. |
| ios::binary | Open in binary mode. |
| ios::app | All output operations are performed at the end of the file, appending the content to the current content of the file. This flag can only be used in streams open for output-only operations. |

All these flags can be combined using the bitwise operator OR (|). For example, if we want to open the file example.bin in binary mode to add data we could do it by the following call to member function open():

|  |  |
| --- | --- |
|  | ofstream myfile;  myfile.open ("example.bin", ios::out | ios::app | ios::binary); |

Each one of the open() member functions of the classes ofstream, ifstream and fstream has a default mode that is used if the file is opened without a second argument:

|  |  |
| --- | --- |
| **class** | **default mode parameter** |
| ofstream | ios::out |
| ifstream | ios::in |
| fstream | ios::in | ios::out |

For ifstream and ofstream classes, ios::in and ios::out are automatically and respectively assumed, even if a mode that does not include them is passed as second argument to the open() member function.

The default value is only applied if the function is called without specifying any value for the mode parameter. If the function is called with any value in that parameter the default mode is overridden, not combined.

File streams opened in binary mode perform input and output operations independently of any format considerations. Non-binary files are known as *text files*, and some translations may occur due to formatting of some special characters (like newline and carriage return characters).

Since the first task that is performed on a file stream object is generally to open a file, these three classes include a constructor that automatically calls the open() member function and has the exact same parameters as this member. Therefore, we could also have declared the previous myfile object and conducted the same opening operation in our previous example by writing:

|  |  |
| --- | --- |
|  | ***ofstream myfile ("example.bin", ios::out | ios::app | ios::binary);*** |

Combining object construction and stream opening in a single statement. Both forms to open a file are valid and equivalent.

To check if a file stream was successful opening a file, you can do it by calling to member is\_open() with no arguments. This member function returns a bool value of true in the case that indeed the stream object is associated with an open file, or false otherwise:

|  |  |
| --- | --- |
|  | if (myfile.is\_open()) { /\* ok, proceed with output \*/ } |

**Closing a File**

When we are finished with our input and output operations on a file we shall close it so that its resources become available again. In order to do that we have to call the stream's member function close(). This member function takes no parameters, and what it does is to flush the associated buffers and close the file:

|  |  |
| --- | --- |
|  | myfile.close(); |

Once this member function is called, the stream object can be used to open another file, and the file is available again to be opened by other processes.

In case that an object is destructed while still associated with an open file, the destructor automatically calls the member function close().

**Text Files**

Text file streams are those where we do not include the ios::binary flag in their opening mode. These files are designed to store text and thus all values that we input or output from/to them can suffer some formatting transformations, which do not necessarily correspond to their literal binary value.

Data output operations on text files are performed in the same way we operated with cout.

**Example 12.2**

|  |  |
| --- | --- |
|  | // writing on a text file  #include <iostream>  #include <fstream>  using namespace std;  int main () {  ofstream myfile ("example.txt");  if (myfile.is\_open())  {  myfile << "This is a line.\n";  myfile << "This is another line.\n";  myfile.close();  }  else cout << "Unable to open file";  return 0;  } |
|  | [file example.txt]  This is a line.  This is another line. |

Data input from a file can also be performed in the same way that we did with cin:

|  |  |
| --- | --- |
|  | // reading a text file  #include <iostream>  #include <fstream>  #include <string>  using namespace std;  int main () {  string line;  ifstream myfile ("example.txt");  if (myfile.is\_open())  {  while ( myfile.good() )  {  getline (myfile,line);  cout << line << endl;  }  myfile.close();  }  else cout << "Unable to open file";  return 0;  } |
|  | This is a line.  This is another line. |

This last example reads a text file and prints out its content on the screen. Notice how **we have used a new member function, called good() that returns true in the case that the stream is ready for input/output operations**. We have created a while loop that finishes when indeed myfile.good() is no longer true, which will happen either if the end of the file has been reached or if some other error occurred.

**Checking State Flags**

In addition to good(), which checks whether the stream is ready for input/output operations, other member functions exist to check for specific states of a stream (all of them return a bool value):

***bad***()

Returns true if a reading or writing operation fails. For example in the case that we try to write to a file that is not open for writing or if the device where we try to write has no space left.

***fail***()

Returns true in the same cases as bad(), but also in the case that a format error happens, like when an alphabetical character is extracted when we are trying to read an integer number.

***eof***()

Returns true if a file open for reading has reached the end.

***good***()

It is the most generic state flag: it returns false in the same cases in which calling any of the previous functions would return true.

In order to reset the state flags checked by any of these member functions we have just seen we can use the member function clear(), which takes no parameters.

**get and put Stream Pointers**

All i/o streams objects have, at least, one internal stream pointer:  
  
ifstream, like istream, has a pointer known as the *get pointer* that points to the element to be read in the next input operation.  
  
ofstream, like ostream, has a pointer known as the *put pointer* that points to the location where the next element has to be written.  
  
Finally, fstream, inherits both, the get and the put pointers, from iostream (which is itself derived from both istream and ostream).  
  
These internal stream pointers that point to the reading or writing locations within a stream can be manipulated using the following member functions:

**tellg() and tellp()**

These two member functions have no parameters and return a value of the member type pos\_type, which is an integer data type representing the current position of the get stream pointer (in the case of tellg) or the put stream pointer (in the case of tellp).

**seekg() and seekp()**

These functions allow us to change the position of the get and put stream pointers. Both functions are overloaded with two different prototypes. The first prototype is:  
seekg ( position );  
seekp ( position );  
Using this prototype the stream pointer is changed to the absolute position position (counting from the beginning of the file). The type for this parameter is the same as the one returned by functions tellg and tellp: the member type pos\_type, which is an integer value.  
The other prototype for these functions is:  
seekg ( offset, direction );  
seekp ( offset, direction );

Using this prototype, the position of the get or put pointer is set to an offset value relative to some specific point determined by the parameter direction. offset is of the member type off\_type, which is also an integer type. And direction is of type seekdir, which is an enumerated type (enum) that determines the point from where offset is counted from, and that can take any of the following values:

|  |  |
| --- | --- |
| ios::beg | offset counted from the beginning of the stream |
| ios::cur | offset counted from the current position of the stream pointer |
| ios::end | offset counted from the end of the stream |

The following example uses the member functions we have just seen to obtain the size of a file.

**Example 12.3**

|  |  |
| --- | --- |
|  | // obtaining file size  #include <iostream>  #include <fstream>  using namespace std;  int main () {  long begin,end;  ifstream myfile ("example.txt");  begin = myfile.tellg();  myfile.seekg (0, ios::end);  end = myfile.tellg();  myfile.close();  cout << "size is: " << (end-begin) << " bytes.\n";  return 0;  } |
|  | size is: 40 bytes. |

**Binary Files**

In binary files, to input and output data with the extraction and insertion operators (<< and >>) and functions like getline is not efficient, since we do not need to format any data, and data may not use the separation codes used by text files to separate elements (like space, newline, etc...).

File streams include two member functions specifically designed to input and output binary data sequentially: write and read. The first one (write) is a member function of ostream inherited by ofstream. And read is a member function of istream that is inherited by ifstream. Objects of class fstream have both members. Their prototypes are:

***write ( memory\_block, size );***

***read ( memory\_block, size );***

Where memory\_block is of type "pointer to char" (char\*), and represents the address of an array of bytes where the read data elements are stored or from where the data elements to be written are taken. The size parameter is an integer value that specifies the number of characters to be read or written from/to the memory block.

**Example 12.4**

Program to write an object to the disk

|  |  |
| --- | --- |
|  | #include <iostream>  #include <fstream>  using namespace std;  //////////////////////////////////////////////////////////class person //class of persons  {  protected:  char name[80]; //person’s name  short age; //person’s age  public:  void getData() //get person’s data  {  cout << “Enter name: “; cin >> name;  cout << “Enter age: “; cin >> age;  }  };  //////////////////////////////////////////////////////////  int main()  {  person pers; //create a person  pers.getData(); //get data for person  //create ofstream object  ofstream outfile(“PERSON.DAT”, ios::binary);  //write to it  outfile.write(reinterpret\_cast<char\*>(&pers), sizeof(pers));  return 0;  } |
|  |  |

In this example, an object of person class is written to a file. We can write another program to read this object. Note that the data members of the two classes (written/read) should be exactly the same while member functions may vary as member functions are not save to the disk.

Program to read an object from the disk

|  |  |
| --- | --- |
|  | #include <fstream> //for file streams  #include <iostream>  using namespace std;  //////////////////////////////////////////////////////////  class person //class of persons  {  protected:  char name[80]; //person’s name  short age; //person’s age  public:  void showData()  {  cout << “Name: “ << name << endl;  cout << “Age: “ << age << endl;  }  };  //////////////////////////////////////////////////////////  int main()  {  person pers; //create person variable  ifstream infile(“PERSON.DAT”, ios::binary);  //read stream  infile.read(reinterpret\_cast<char\*>(&pers), sizeof(pers) );  pers.showData(); //display person  return 0;  } |
|  |  |

**Exercise 1**

|  |
| --- |
| Create a class Employee with name, Id and salary as data members. Provide appropriate constructors, set, get and display methods in the class.  In the main program, use a do-while loop to enter data for employees as long as the user desires and save all data to a file.  Once the user is done with data entry, read the data for employees from the file and display the information of each employee. |

**Code :**

#include <iostream>

#include <fstream>

using namespace std;

class Employee {

string name;

int ID;

int salary;

public :

Employee(string a= " ", int b = 0,int c=0) : name(a),ID(b),salary(c){}

void setName(string a) {

name = a;

}

void setID(int a) {

ID = a;

}

void setSalary(int a) {

salary = a;

}

int getID() {

return ID;

}

int getSalary() {

return salary;

}

string getName() {

return name;

}

void getData() {

cout << "Enter name : ";

cin >> name;

cout << "Enter ID : ";

cin >> ID;

cout << "Enter Salary : ";

cin >> salary;

}

void showData() {

cout << "Name : " << getName() << endl;

cout << "ID : " << getID() << endl;

cout << "Salary : " << getSalary() << endl;

}

};

int main() {

Employee e1;

char choice;

ofstream out("Employees.dat", ios::binary | ios::app);

do {

e1.getData();

out.write((char\*)&e1, sizeof(e1));

cout << "Do you want to add another enployee? (Y/N)" << endl;

cin >> choice;

} while (choice == 'Y' || choice == 'y');

out.close();

system("CLS");

cout << "Employee Information : " << endl;

ifstream in("Employees.dat", ios::binary);

while (in.read((char\*)&e1, sizeof(e1))) {

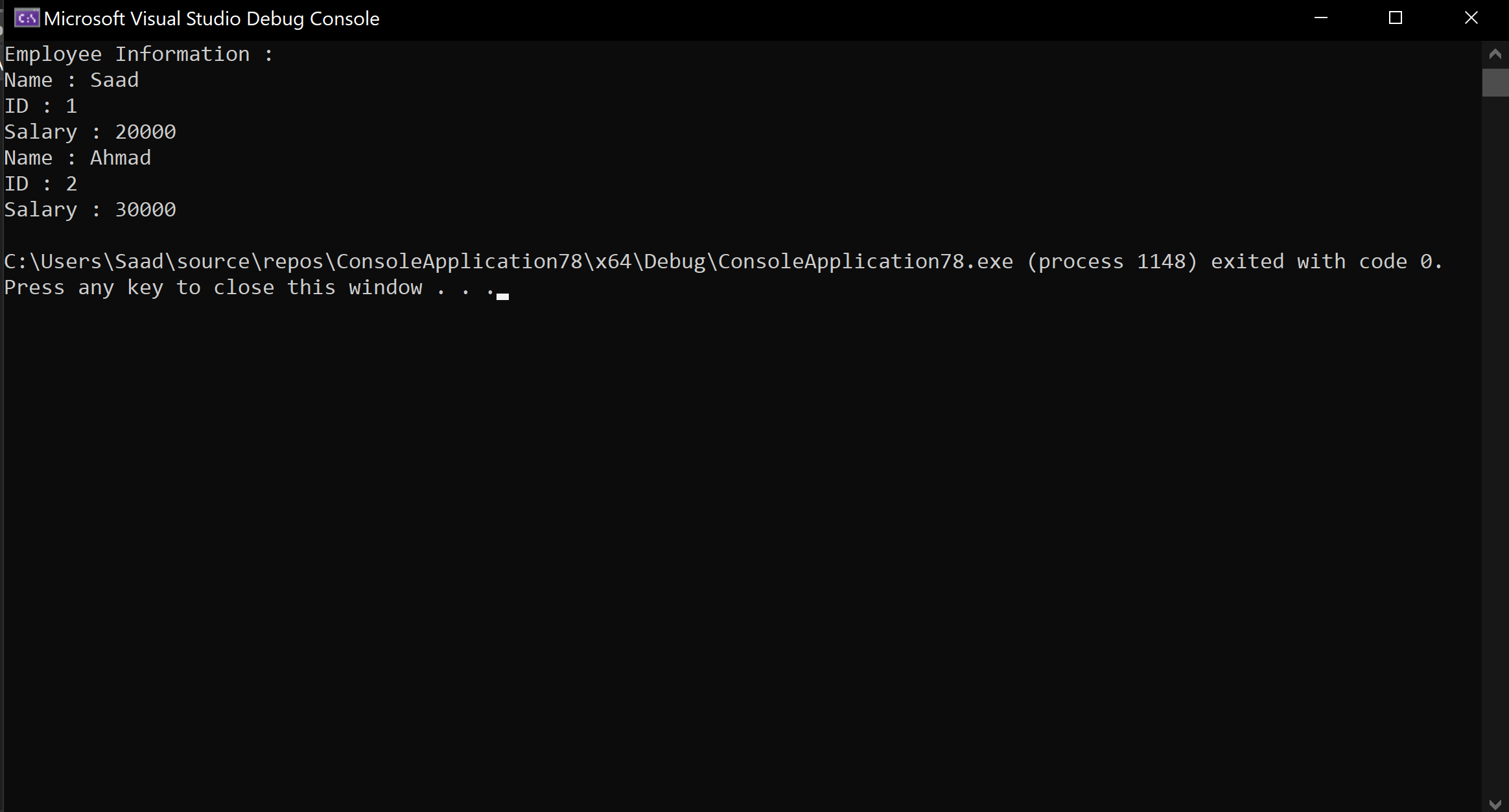
e1.showData();

}

return 0;

}

**Output :**

****

**Exercise 2**

|  |
| --- |
| Using the data file generated by the program in Exercise 1, write a program which allows user to enter the name of an Employee whose record is searched in the data file. Once found, display the information of the concerned employee. If the record is not found, display an error message. |

**Code :**

#include <iostream>

#include <fstream>

using namespace std;

class Employee {

string name;

int ID;

int salary;

public :

void showData() {

cout << "Name : " << name << endl;

cout << "ID : " << ID << endl;

cout << "Salary : " << salary << endl;

}

string getName() {

return name;

}

};

int main() {

string name;

bool check = false;

cout << "Enter the name of the person you want to search : " << endl;

cin >> name;

Employee e1;

cout << "Employee Information : " << endl;

ifstream in("Employees.dat", ios::binary);

while (in.read((char\*)&e1, sizeof(e1))) {

if (e1.getName() == name) {

e1.showData();

check = true;

break;

}

}

if (!check) {

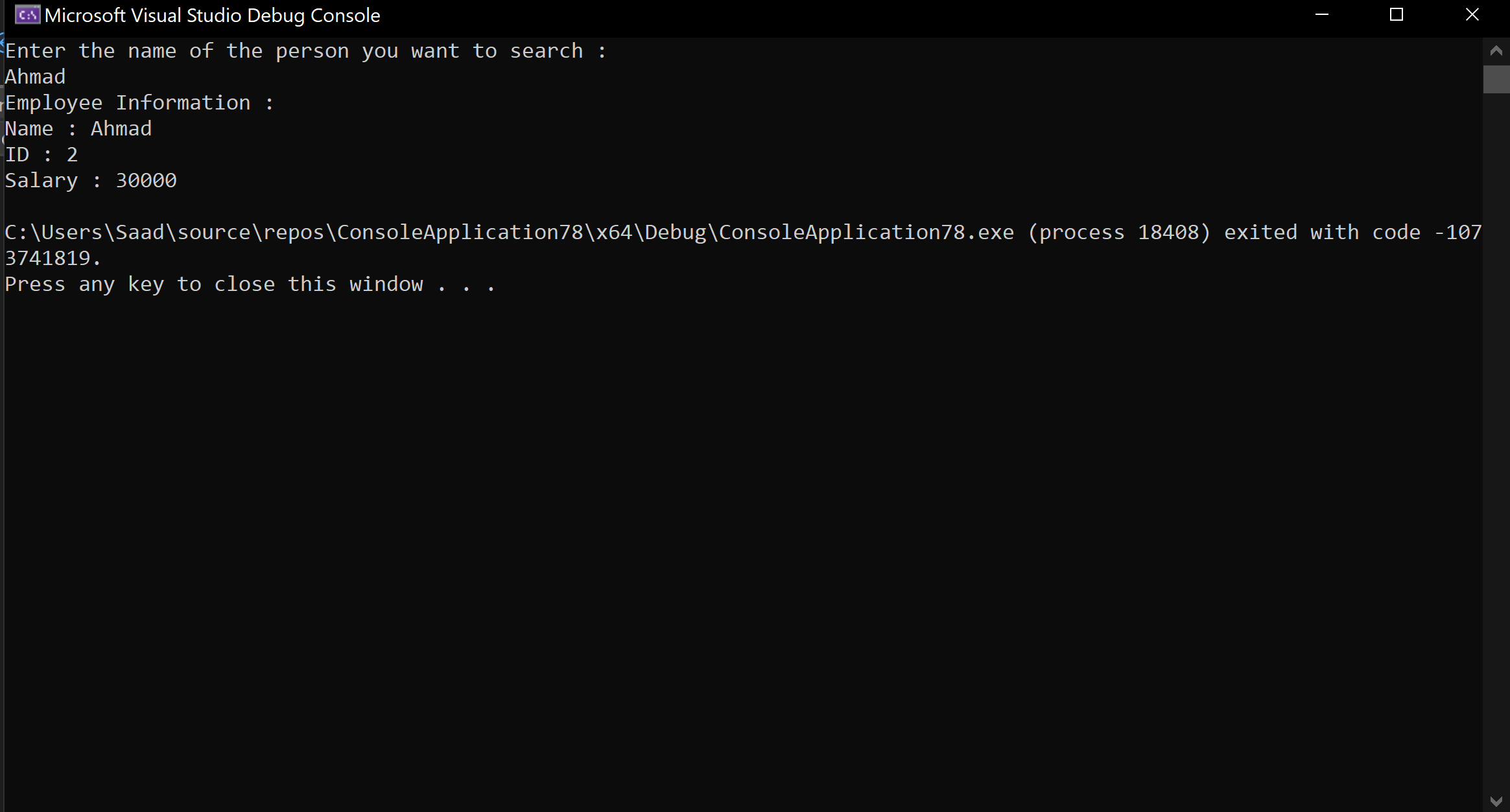
cout << "Error! Employee does not exist" << endl;

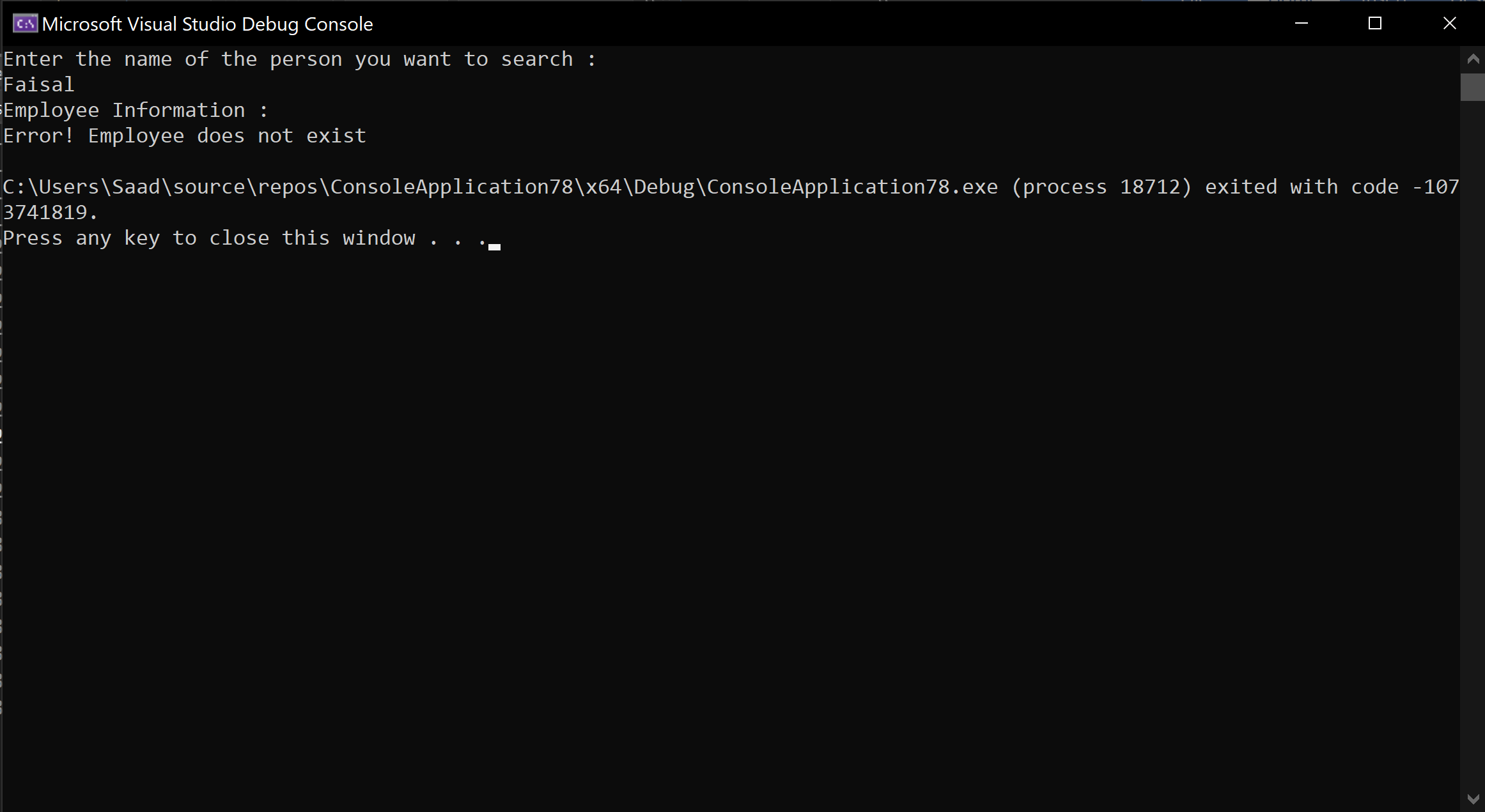
}

return 0;

}

**Output :**

****

****

**Exercise 3**

|  |
| --- |
| Creat a class Event with two integer data members, the ID of event and the number of participants in the event. In addition to the standard functions in the class, provide a member function writeToDisk() to write the current object to a file on the disk.  Also provide a member function readFromDisk(int recordNo) which reads the event number passed as an argument from the file into current object.  In addition, provide a member function that counts and returns the total number of event objects in the file.  Program should allow statements like :  Event e1, e2 ;  e1.input() ; e2.input() ;  e1.writeToDisk() ;  e2.writeToDisk() ;  …..  Event e ;  e.readFromDisk(1) ;  ….  In the main program, write objects of class Event in a do-while loop as long as user desires. Later, find the total number of events written to the file. Using a for loop, read details of each event and display the read data to the user. |

Code:

#include <iostream>

#include <fstream>

using namespace std;

class Event {

int ID;

int no\_of\_participants;

public:

Event( int b = 0, int c = 0) : ID(b), no\_of\_participants(c) {}

void setID(int a) {

ID = a;

}

void setNOP(int a) {

no\_of\_participants = a;

}

int getID() {

return ID;

}

int getNOP() {

return no\_of\_participants;

}

void input() {

cout << "Enter ID : ";

cin >> ID;

cout << "Enter number of participants : ";

cin >> no\_of\_participants;

}

void showData() {

cout << "ID : " << getID() << endl;

cout << "Number of participants : " << no\_of\_participants << endl;

}

void writeToDisk() {

ofstream out("Event.dat", ios::binary | ios::app);

out.write((char\*)this,sizeof(this));

out.close();

}

void readFromDisk(int recordNo) {

ifstream in("Event.dat", ios::binary);

while (in.read((char\*)this, sizeof(this))) {

if (ID == recordNo) {

showData();

break;

}

}

in.close();

}

int eventCount() {

int counter=0;

ifstream in("Event.dat", ios::binary);

while (in.read((char\*)this, sizeof(this))) {

counter++;

}

in.close();

return counter;

}

};

int main() {

Event e1,e2;

char choice;

ofstream out("Event.dat", ios::binary | ios::app);

do {

e1.input();

out.write((char\*)&e1, sizeof(e1));

cout << "Do you want to add another event? (Y/N)" << endl;

cin >> choice;

} while (choice == 'Y' || choice == 'y');

out.close();

int noofevents = e2.eventCount();

cout << "Total number of events : " << noofevents << endl;

for (int i = 0; i <= noofevents; i++) {

e1.readFromDisk(i);

}

return 0;

}

Output:

