Fundamentals of Programming in C++
CoSc 2031

Chapter 4

File Operations (File I/O) in C++

Mekonnen K.
(MSc)

Email: mekonnen307@gmail.com

Topics

Topics	Subtopics
4: File Operation (File I/O)	4.1. Introduction 4.2. Stream classes 4.3. Writing and reading modes 4.4. Writing to and reading from files 4.5. Types of files (Text and Binary) 4.6. File access methods (sequential and random-access files) 4.7. Object Oriented Programming Concept 4.7.1 Programming Paradigm 4.7.2. Overview of programming principal 4.7.3. Class and Object 4.7.4. Pillars of Object-Oriented Programming Concept 4.7.4.1. Encapsulation, 4.7.4.2. Abstraction, 4.7.4.3. Inheritance 4.7.4.5. Polymorphism

File

- A file is a collection of related data stored in a particular area on the disk . The data is stored in disk using the concept of file .
- A computer file is stored on a secondary storage device (e.g., disk);
  - is permanent;
  - can be used to
    - provide input data to a program
    - or receive output data from a program
    - or both;
  - should reside in Project directory for easy access;
  - must be opened before it is used.

Why File

- Permanent storage of data : - (all the message or value printed with help of any output statements like cout, printf , putchar are never available for future use) .
- If there is a large amount of data generated as output by a program, storing that output in file will help in easy handling /analysis of the output , as user can see the whole output at any time even after complete execution of the program.

- If we need lot of data to be inputted, user cannot keep on typing that again and again for repeated execution of program. In that case, all input data can be once written in a file and then that file can be easily used as the input file.
- The transfer of input – data or output – data from one computer to another can be easily done by using files.

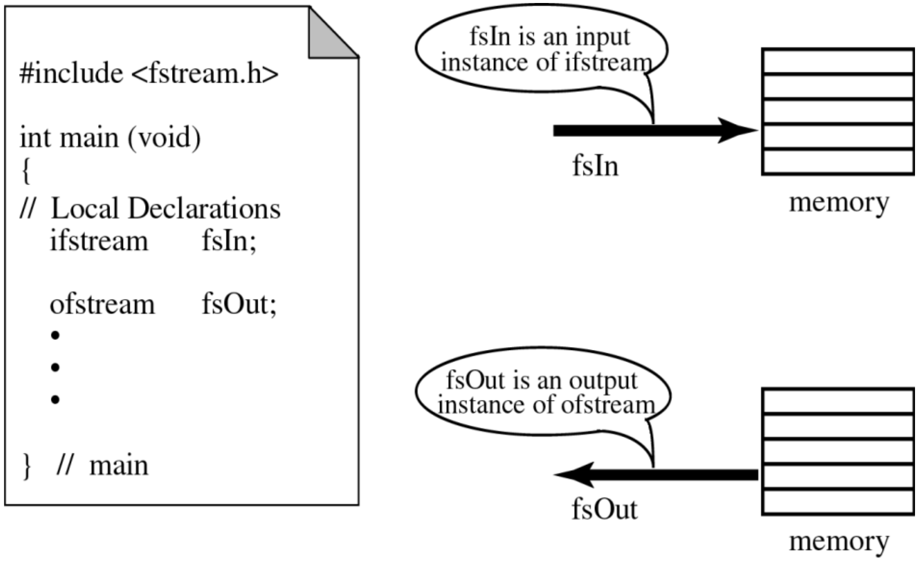
File Stream classes

- Filebuf :- its purpose is to set the file buffer to read and write . Contain **open rot** constant used in the **open()** of file stream classes . Also contain **close()** and **open()** as method.
- Fstreambase :- provides operations common to the file stream. Serves as a base for fstream, ifstream and ofsteram class. Contains **open()** and **close()** function
- Ifstream :- provides input operations and used for reading from files. Contains open() with default input mode. Inherits the functions **get()**, **getline()**, **read()**, **seekg()** and **tellg()** function from istream.

- **Ofstream :-** provides output operations and used for writing in files. Contains **open()** with default output mode. Inherits **put()**, **seekp()**, **teelp()** and **write()** function from ostream.
- **Fsream :-** provides support for simultaneous input and output operations. Contains **open()** with default input mode. Inherits all the function from **isteram** and ostream classes through **iostream**.
- It is used for read from files and write to files.

C++ streams

```
//Add additional header files you use
#include <fstream>
int main ()
{ /* Declare file stream variables such as
 the following  */
ifstream  fsIn;//input
ofstream fsOut; // output
fstream both //input & output
//Open the files
fsIn.open("prog1.txt"); //open the input file
fsOut.open("prog2.txt"); //open the output file
//Code for data manipulation
.
.
//Close files
fsIn.close();
fsOut.close();
return 0; }
```



```
#include <fstream.h>

int main (void)
{
//  Local Declarations
    ifstream      fsIn;

    ofstream      fsOut;
    •
    •
    •

}   //  main
```

fsIn is an input instance of ifstream
fsIn
memory
fsOut is an output instance of ofstream
fsOut
memory

General File I/O Steps

1. Include the header file fstream in the program.
2. Declare file stream variables.
3. Associate the file stream variables with the input/output sources.
4. Open the file
5. Use the file stream variables with >>, <<, or other input/output functions.
6. Close the file.

Opening a file using Open()

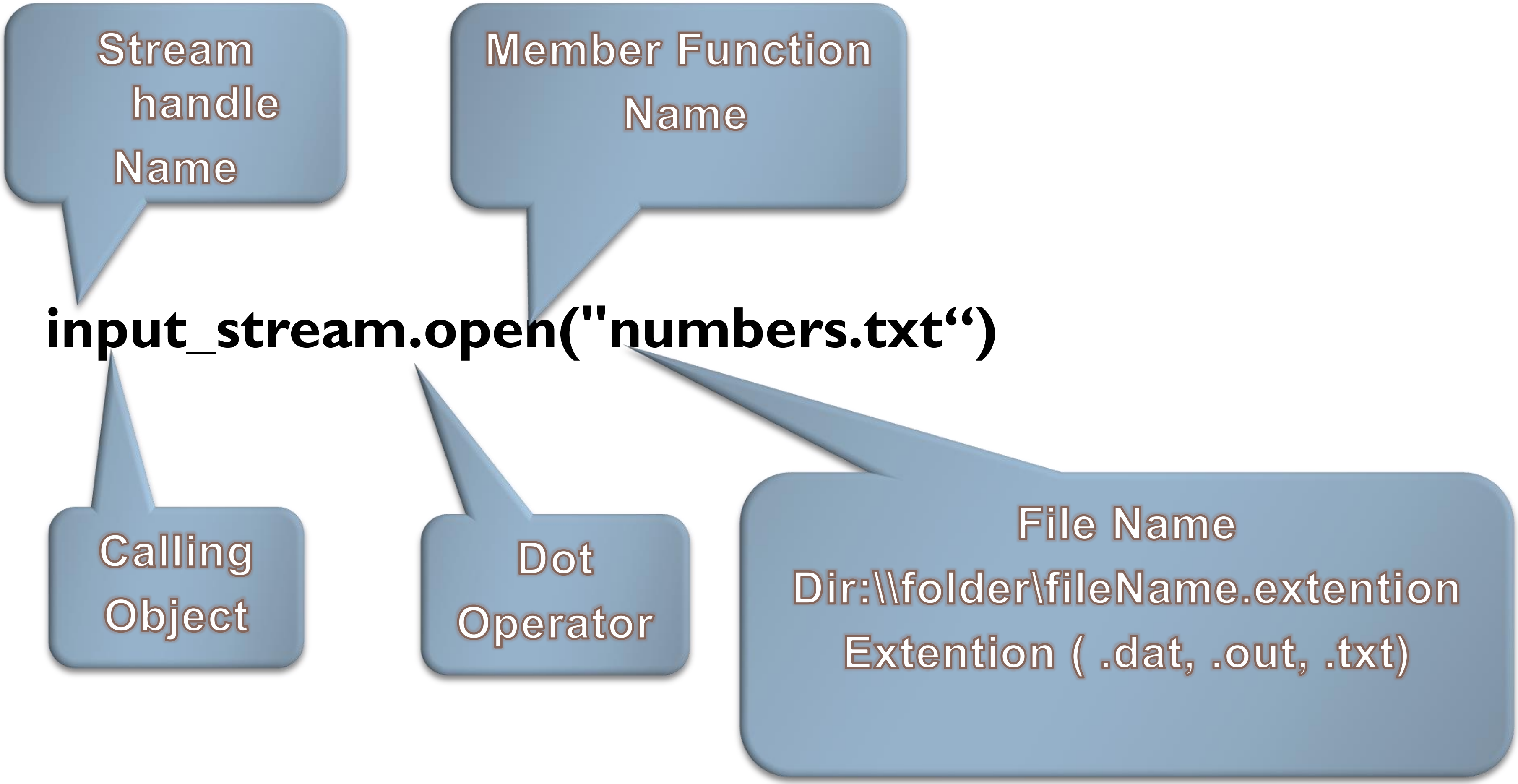
- **Opening a file** associates a file stream variable declared in the program with a physical file at the source, such as a disk.
- The function **open()** can be used to open multiple files that use the same stream object.

```
file-stream-class   stream-object;

stream-object . open (“filename”);
```

- In the case of an input file:
  - the file must exist before the open statement executes.
  - If the file does not exist, the open statement fails and the input stream enters the fail state
- An output file does not have to exist before it is opened;
  - if the output file does not exist, the computer prepares an empty file for output.
  - If the designated output file already exists, by default, the old contents are erased when the file is opened.

Object and Member Functions



Stream handle Name

Member Function Name

```
input_stream.open("numbers.txt“)
```

Calling Object

Dot Operator

File Name

Dir:\\folder\fileName.extention

Extention (.dat, .out, .txt)

Open and close a file

eg:-

```
 ofstream  outfile;            // create stream
 outfile . open ("DATA1");  // connect stream to DATA1
 ………………………………
 ………………………………
 outfile . Close();             //disconnect stream from
  DATA1
 outfile . Open("DATA2"); //connect stream to DATA2
 ………………………………
 ………………………………
 outfile . close();
 ………………………………
```

Validate the file before trying to access

First method

By checking the stream variable;

```
If ( ! Mystream)
{
Cout << “Cannot open file.\n ”;
}
```

Second method

By using bool is_open() function.

```
If ( ! Mystream.is_open())
{
Cout << “File is not open.\n ”;
}
```

File I/O Example: Open the file with validation

First Method (use the constructor)

```
#include <fstream>
using namespace std;
int main()
{
//declare and automatically
open the file
ofstream outFile(“fout.txt");
// Open validation
if(! outFile) {
Cout << “Cannot open file.\n ”;
return 1;
}
return 0;
}
```

Second Method (use Open function)

```
#include <fstream>
using namespace std;
int main()
{
//declare output file variable
ofstream outFile;
// open an exist file fout.txt
outFile.open(“fout.txt”);
// Open validation
if(! outFile.is_open() ) {
Cout << “Cannot open file.\n ”;
return 1;
}
return 0;
}
```

Mode of file opening

- ios :: out = open file for write only
- ios :: in = open file for read only
- ios :: app = append to end-of-file
- ios :: ate = take us to the end of the file when it is opened
- Both ios :: app and ios :: ate take us to the end of the file when it is opened. The difference between the two parameters is that the ios :: app allows us to add data to the end of file only, while ios :: ate mode permits us to add data or to modify the existing data any where in the file.

Mode of file opening cont'd…

- The mode can combine two or more parameters using the bitwise **OR** operator (symbol |)

  eg :-

  ```
  fstream file;
  file . Open(" data . txt", ios :: out | ios :: in);
  ```

File Open Mode

```
#include <fstream>
int main(void)
{
ofstream outFile("file1.txt", ios::out);
outFile << "That's new!\n";
outFile.close();
        Return 0;
}
```

If you want to set more than one open mode, just use the **OR** operator- |. This way:

```
ios::in | ios::out
```

Writing to a File

- While doing C++ programming, you write information to a file from your program using the stream insertion operator << just as you use that operator to output information to the screen.
- The only difference is that you use an ofstream or fstream object instead of the cout object.
- we are sending output to a file. So, we use ios::out. As given in the program, information typed inside the quotes after **"FilePointer <<"** will be passed to output file.

File I/O Example: Writing

First Method (use the constructor)

```
#include <fstream>
using namespace std;
int main()
{/* declare and automatically open
              the file*/
ofstream outFile("fout.txt");

//behave just like cout, put the word into
the file
outFile << "Hello World!";

outFile.close();

return 0;
}
```

Second Method (use Open function)

```
#include <fstream>
using namespace std;
int main()
{// declare output file variable
ofstream outFile;
// open an exist file fout.txt
   outFile.open("fout.txt”);

//behave just like cout, put the word into
the file
outFile << "Hello World!";

outFile.close();

return 0;
}
```

Reading from a File

- You read information from a file into your program using the stream extraction operator >> just as you use that operator to input information from the keyboard.
- The only difference is that you use an ifstream or fstream object instead of the cin object.
- we are reading input from a file. So, we use ios::in. As given in the program, information from the output file is obtained with the help of following syntax **"FilePointer >>variable"**.

File I/O Example: Reading

Read char by char

```
#include <iostream>
#include <fstream>

int main()
{           //Declare and open a text file
       ifstream openFile(“data.txt");
           char ch;
       //do until the end of file
       while( ! OpenFile.eof() )
      {
           OpenFile.get(ch); // get one character
           cout << ch;   // display the character
      }
      OpenFile.close(); // close the file

   return 0;
}
```

Read a line

```
#include <iostream>
#include <fstream>
#include <string>
int main()
{//Declare and open a text file
     ifstream openFile("data.txt");
     string line;
     while(!openFile.eof())
     { //fetch line from data.txt and put
           it in a string
           getline(openFile, line);
           cout << line;
     }
     openFile.close(); // close the file
  return 0;
}
```

Example writing into and Reading from file

```
#include <iostream>
#include <fstream>
#include <string>
using namespace std;
int main ()
{
        ofstream MyWriteFile("filename.txt"); // Create a text file
         MyWriteFile << "Files can be tricky, but it is fun enough!"; // Write to the file
        MyWriteFile.close(); // Close the file
        // Create a text string, which is used to output the text file
        string myText;
        ifstream MyReadFile("filename.txt"); // Read from the text file
        // Use a while loop together with the getline() function to read the file line by line
        while (getline (MyReadFile, myText))
        {
                cout << myText; // Output the text from the file
        }
        MyReadFile.close(); // Close the file
}
```

Closing a File

- A file must be close after completion of all operation related to 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.
- This member function takes no parameters, and what it does is to flush the associated buffers and close the file. For closing file we need **close()** function.

```
	st.close();
```

- Syntax

```
		st.close();
```

File pointer

- Each file have two associated pointers known as the **file pointers**. One of them is called the input pointer (or **get pointer**) and the other is called the output pointer (or **put pointer**).
- The **input pointer** is used for reading the contents of a given file location and the **output pointer** is used for writing to a given file location.

Function for manipulation of file pointer

- When we want to move file pointer to desired position then use these function for manage the file pointers.

- Seekg () = is used to move the get pointer to a desired location with respect to a reference point.

Syntax: file_pointer.seekg (number of bytes ,Reference point);
Example: fin.seekg(10,ios::beg);

- tellg () = gives the current position of the get pointer

  = is used to know where the get pointer is in a file.

Syntax: file_pointer.tellg();

Example: int posn = fin.tellg();

Function for manipulation of file pointer

- Seekp () = is used to move the put pointer to a desired location with respect to a reference point.

  Syntax: file_pointer.seekp(number of bytes ,Reference point);

  Example: fout.seekp(10,ios::beg);

- tellp () = gives the current position of the put pointer

  = is used to know where the put pointer is in a file.

  Syntax: file_pointer.tellp();

  Example: int posn=fout.tellp();

Function for manipulation of file pointer

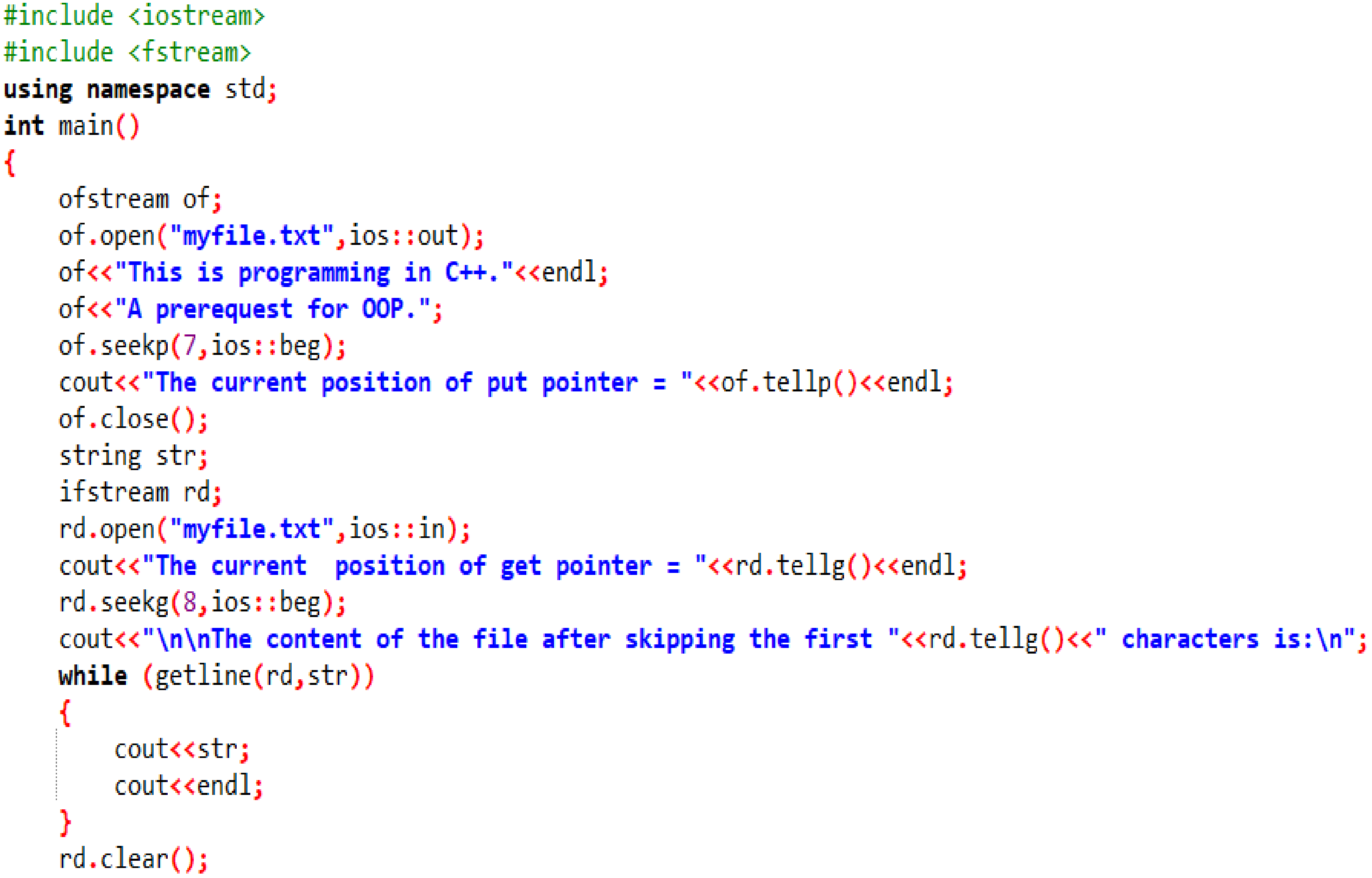
- The reference points are:
  - ios::beg – from beginning of file
  - ios::end – from end of file
  - ios::cur – from current position in the file.

- fout . seekg(0, ios :: beg) -- go to start
- fout . seekg(0, ios :: cur) -- stay at current position
- fout . seekg(0, ios :: end) -- go to the end of file
- fout . seekg(m, ios :: beg) -- move to m+1 byte in the file
- fout . seekg(m, ios :: cur) -- go forward by m bytes from the current position
- fout . seekg(-m, ios :: cur) -- go backward by m bytes from the current position
- fout . seekg(-m, ios :: end) -- go backward by m bytes from the end

Function for manipulation of file pointer

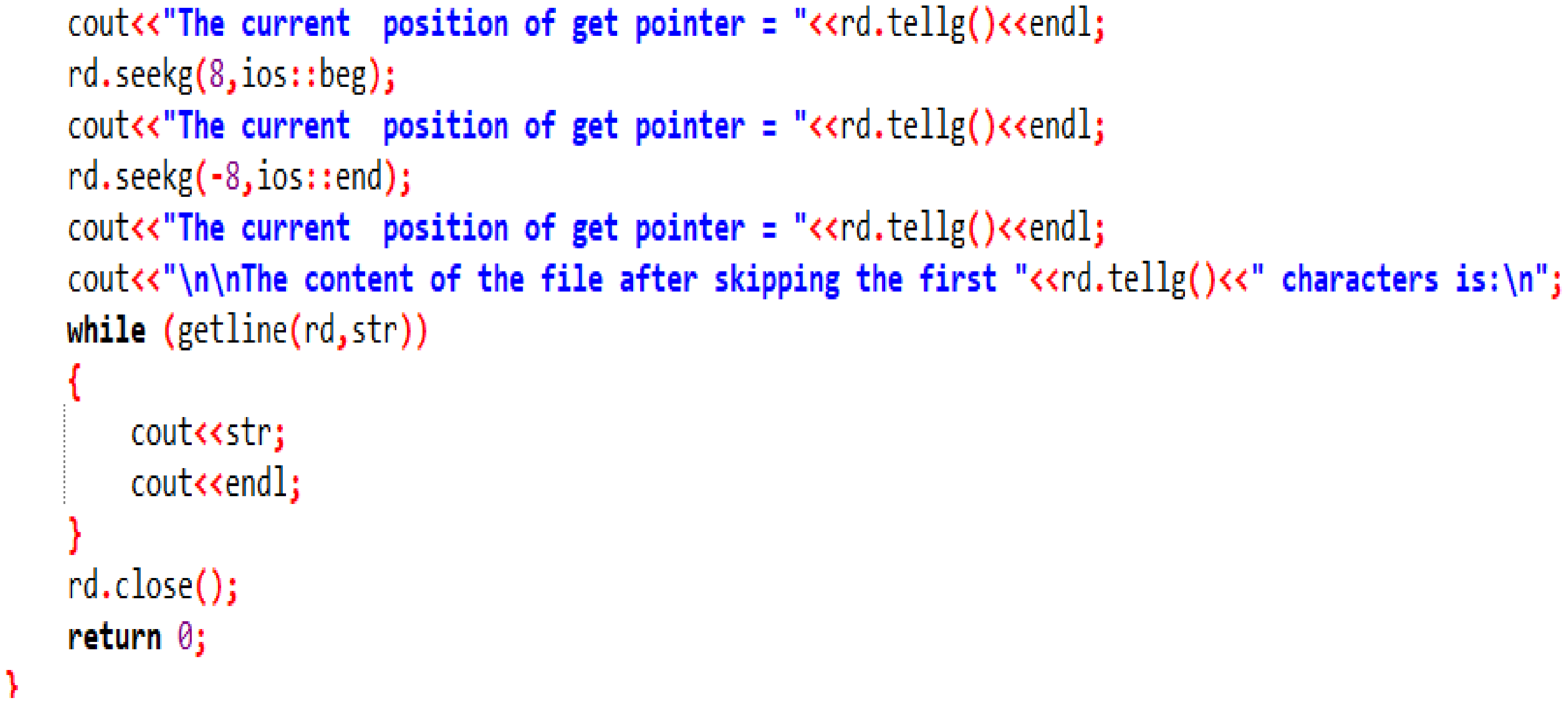
- Here is an example.
  - fin.seekg(30, ios::beg); // Go to byte no. 30 from the beginning of the file linked with "fin."
  - fin.seekg(-2, ios::cur); // Back up 2 bytes from the get pointer's current position.
  - fin.seekg(0, ios::end); // Go to the end of the file.
  - fin.seekg(-4, ios::end); // Backup 4 bytes from the end of the file.

Function for manipulation of file pointer



```
#include <iostream>
#include <fstream>
using namespace std;
int main()
{
    ofstream of;
    of.open("myfile.txt",ios::out);
    of<<"This is programming in C++."<<endl;
    of<<"A prerequest for OOP.";
    of.seekp(7,ios::beg);
    cout<<"The current position of put pointer = "<<of.tellp()<<endl;
    of.close();
    string str;
    ifstream rd;
    rd.open("myfile.txt",ios::in);
    cout<<"The current  position of get pointer = "<<rd.tellg()<<endl;
    rd.seekg(8,ios::beg);
    cout<<"\n\nThe content of the file after skipping the first "<<rd.tellg()<<" characters is:\n";
    while (getline(rd,str))
    {
        cout<<str;
        cout<<endl;
    }
    rd.clear();
```

Function for manipulation of file pointer



```
    cout<<"The current  position of get pointer = "<<rd.tellg()<<endl;
    rd.seekg(8,ios::beg);
    cout<<"The current  position of get pointer = "<<rd.tellg()<<endl;
    rd.seekg(-8,ios::end);
    cout<<"The current  position of get pointer = "<<rd.tellg()<<endl;
    cout<<"\n\nThe content of the file after skipping the first "<<rd.tellg()<<" characters is:\n";
    while (getline(rd,str))
    {
        cout<<str;
        cout<<endl;
    }
    rd.close();
    return 0;
}
```

put() and get() function

- The function put() write a single character to the associated stream. Similarly, the function get() reads a single character from the associated stream.

Types of Files

- The C++ language supports two types of files:
  - Text files
  - Binary files
- The basic difference between text files and binary files is that in text files various character translations are performed such as “\r+\f” is converted into “\n”, whereas in binary files no such translations are performed.
- By default, C++ opens the files in text mode.

Cont'd

- A **text file** stores data in the form of alphabets, digits and other special symbols by storing their ASCII values and are in a human readable format. For example, any file with a .txt, .c, etc extension.
- Whereas, a **binary file** contains a sequence or a collection of bytes which are not in a human readable format. For example, files with .exe, .mp3, etc extension. It represents custom data.
- A small error in a **textual file** can be recognized and eliminated when seen. Whereas, a small error in a **binary file** corrupts the file and is not easy to detect.

File Access Modes

- **Sequential Access File**:- **Sequential Access** to a data **file** means that the computer system reads or writes information to the **file sequentially**, starting from the beginning of the **file** and proceeding step by step.
- **Random Access /Direct Access**:- **Random Access** to a **file** means that the computer system can **read** or write information anywhere in the data **file**.

Object Oriented Programming Concept

- OOP stands for Object-Oriented Programming.
- Procedural programming is about writing procedures or functions that perform operations on the data, while object-oriented programming is about creating objects that contain both data and functions.
- Object-oriented programming has several advantages over procedural programming:
  - OOP is faster and easier to execute
  - OOP provides a clear structure for the programs
  - OOP helps to keep the C++ code DRY "Don't Repeat Yourself", and makes the code easier to maintain, modify and debug
  - OOP makes it possible to create full reusable applications with less code and shorter development time

What are Classes and Objects?

- Classes and objects are the two main aspects of object-oriented programming.
- Look at the following illustration to see the difference between class and objects:
- example:
- **class**
  - Fruit
- **objects**
  - Apple
  - Banana
  - Mango

Another example:

class

- Car

objects

- Volvo
- Audi
- Toyota

- So, a class is a template for objects, and an object is an instance of a class. When the individual objects are created, they inherit all the variables and functions from the class.

C++ Classes/Objects

- C++ is an object-oriented programming language.
- Everything in C++ is associated with **classes** and **objects**, along with its attributes and methods. For example: in real life, a car is an **object**. The car has **attributes**, such as weight and color, and **methods**, such as drive and brake.
- Attributes and methods are basically **variables** and **functions** that belongs to the class. These are often referred to as "class members".
- A class is a user-defined data type that we can use in our program, and it works as an object constructor, or a "blueprint" for creating objects.

C++ Classes/Objects

- **Create a Class**
- To create a class, use the class keyword:

Example

Create a class called "MyClass":

```
class MyClass // The class
{
        public:               // Access specifier
        int myNum;             // Attribute (int variable)
        string myString;  // Attribute (string variable)
};
```

C++ Classes/Objects

- **Create an Object**
- In C++, an object is created from a class. We have already created the class named MyClass, so now we can use this to create objects.
- To create an object of MyClass, specify the class name, followed by the object name.
- To access the class attributes (myNum and myString), use the dot syntax (.) on the object:

C++ Classes/Objects

- **Example**
- Create an object called "myObj" and access the attributes:

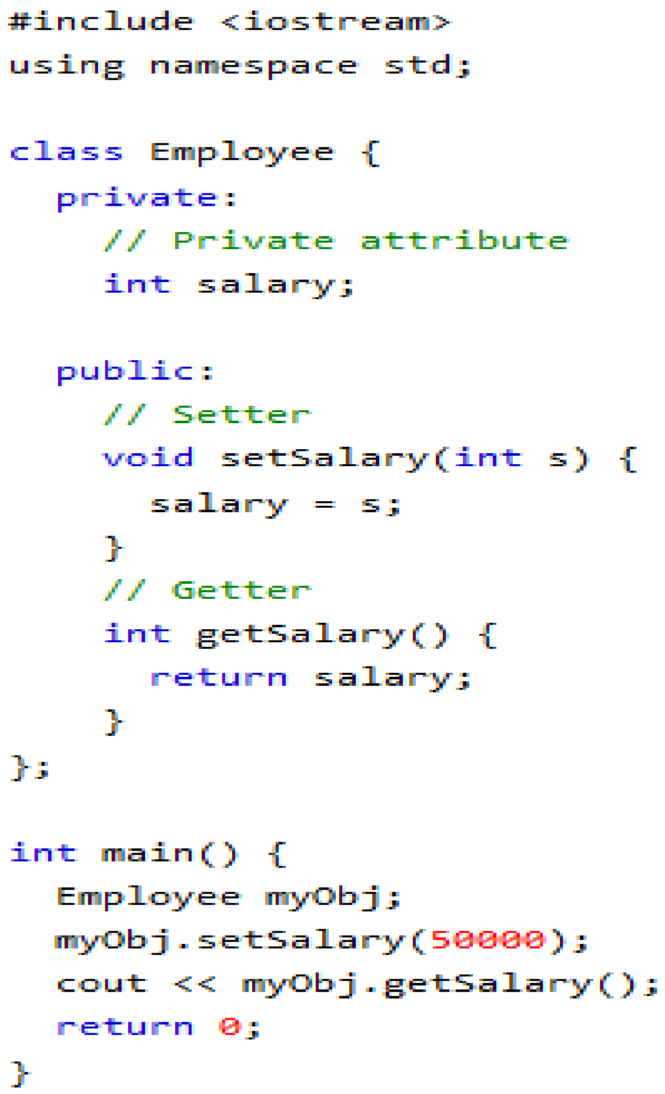
```
lass MyClass {       // The class
  public:            // Access specifier
    int myNum;        // Attribute (int variable)
    string myString;  // Attribute (string variable)
};
int main() {
        MyClass myObj;  // Create an object of MyClass
        // Access attributes and set values
        myObj.myNum = 15;
        myObj.myString = "Some text";
        // Print attribute values
        cout << myObj.myNum << "\n";
        cout << myObj.myString;
        return 0;
}
```

C++ Encapsulation

- The meaning of **Encapsulation**, is to make sure that "sensitive" data is hidden from users. To achieve this, you must declare class variables/attributes as private (cannot be accessed from outside the class). If you want others to read or modify the value of a private member, you can provide public **get** and **set** methods.
- **Access Private Members**
- To access a private attribute, use public "get" and "set" methods:

C++ Encapsulation

- Example



```cpp
#include <iostream>
using namespace std;

class Employee {
  private:
    // Private attribute
    int salary;

  public:
    // Setter
    void setSalary(int s) {
      salary = s;
    }
    // Getter
    int getSalary() {
      return salary;
    }
};

int main() {
  Employee myObj;
  myObj.setSalary(50000);
  cout << myObj.getSalary();
  return 0;
}
```

C++ Encapsulation

- **Example explained**
- The salary attribute is private, which have restricted access.
- The public setSalary() method takes a parameter (s) and assigns it to the salary attribute (salary = s).
- The public getSalary() method returns the value of the private salary attribute.
- Inside main(), we create an object of the Employee class. Now we can use the setSalary() method to set the value of the private attribute to 50000. Then we call the getSalary() method on the object to return the value.
- **Why Encapsulation?**
- It is considered good practice to declare your class attributes as private (as often as you can). Encapsulation ensures better control of your data, because you (or others) can change one part of the code without affecting other parts
- Increased security of data

C++ Inheritance

- In C++, it is possible to inherit attributes and methods from one class to another. We group the "inheritance concept" into two categories:
  - **derived class** (child) - the class that inherits from another class
  - **base class** (parent) - the class being inherited from
- To inherit from a class, use the : symbol.

C++ Inheritance

- In the example below, the Car class (child) inherits the attributes and methods from the Vehicle class (parent):

```
// Base class
class Vehicle {
  public:
    string brand = "Ford";
    void honk() {
      cout << "Tuut, tuut! \n" ;
    }
};
```

```
// Derived class
class Car: public Vehicle {
  public:
    string model = "Mustang";
};
int main() {
  Car myCar;
  myCar.honk();
  cout << myCar.brand + " " +
myCar.model;
  return 0;}
```

C++ Polymorphism

- Polymorphism means "**many forms**", and it occurs when we have many classes that are related to each other by inheritance.
- Like we specified in the previous chapter; **Inheritance** lets us inherit attributes and methods from another class. **Polymorphism** uses those methods to perform different tasks.
- This allows us to perform a single action in different ways.
- For example, think of a base class called Animal that has a method called animalSound(). Derived classes of Animals could be Pigs, Cats, Dogs, Birds - And they also have their own implementation of an animal sound (the pig oinks, and the cat meows, etc.):

C++ Polymorphism

- **Example**

```
// Base class
class Animal {
  public:
    void animalSound() {
      cout << "The animal makes a sound \n";
    }
};
// Derived class
class Pig : public Animal {
  public:
    void animalSound() {
      cout << "The pig says: wee wee \n";
    }
};
```

```
// Derived class
class Dog : public Animal {
  public:
    void animalSound() {
      cout << "The dog says:
bow wow \n";
    }
};
```

C++ Polymorphism

- Now we can create Pig and Dog objects and override the animalSound() method:

- **Example**

-
  ```
  // Base class
  class Animal {
    public:
      void animalSound() {
        cout << "The animal makes a sound \n";
      }
  };
  // Derived class
  class Pig : public Animal {
    public:
      void animalSound() {
        cout << "The pig says: wee wee \n";
      }
  };
  ```

```
// Derived class
class Dog : public Animal {
  public:
    void animalSound() {
      cout << "The dog says: bow wow \n";
    }
};

int main() {
  Animal myAnimal;
  Pig myPig;
  Dog myDog;

  myAnimal.animalSound();
  myPig.animalSound();
  myDog.animalSound();
  return 0;
}
```

END OF CHAPTER FOUR!!!