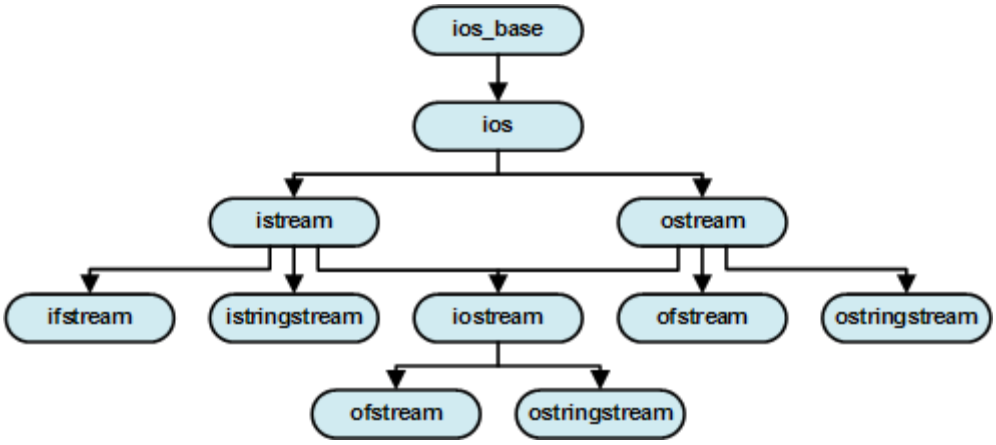
File Processing in C++

Introduction

Storage of data in memory is temporary. Files are used for data persistence—permanent retention of data.

Computers store files on secondary storage devices, such as hard disks, CDs, DVDs, flash drives and tapes



Review Streams

C++ comes with libraries that provide us with many ways for performing input and output.

In C++ input and output are performed in the form of a sequence of bytes or more commonly known as **streams .**

- **Input Stream:** If the direction of flow of bytes is from the device(for example, Keyboard) to the main memory then this process is called input.
- **Output Stream:** If the direction of flow of bytes is opposite, i.e. from main memory to device(display screen) then this process is called output.

Header files available in C++ for Input/Output operations are:

1. **iostream**: iostream stands for standard input-output stream. This header file contains definitions to objects like cin, cout, cerr etc.
2. **iomanip**: iomanip stands for input output manipulators. The methods declared in this files are used for manipulating streams. This file contains definitions of setw, setprecision, etc.
3. **fstream**: This header file mainly describes the file stream. This header file is used to handle the data being read from a file as input or data being written into the file as output.

The istream class: This class is responsible for handling input stream. It provides number of function for handling chars, strings and objects such as **get, getline, read, ignore, putback** etc..

```
#include <iostream>
```

```cpp
using namespace std;
int main()
{
    char x;
    // used to scan a single char
    cin.get(x);
    // used to put a single char onto the screen.
    cout.put(x);
}
```

The iostream: This class is responsible for handling both input and output stream as both **istream class** and o**stream class** is inherited into it. It provides function of both **istream class** and o**stream class** for handling chars, strings and objects such as **get, getline, read, ignore, putback, put, write** etc..

```cpp
#include <iostream>
using namespace std;
int main()
{
    // this function display
    // ncount character from array
    cout.write("geeksforgeeks", 5);
}
```

- **Standard output stream (cout)**: Usually the standard output device is the display screen. The C++ **cout** statement is the instance of the ostream class. It is used to produce output on the standard output device which is usually the display screen. The data needed to be displayed on the screen is inserted in the standard output stream (cout) using the insertion operator(**<<**).

```cpp
#include <iostream>
using namespace std;
int main()
{
    char sample[] = "Test";
    cout << sample << " - A computer science portal for test";
    return 0;
}
```

In the above program the insertion operator(**<<**) inserts the value of the string variable **sample** followed by the string “A computer science portal for geeks” in the standard output stream **cout** which is then displayed on screen.

- **standard input stream (cin)**: Usually the input device in a computer is the keyboard. C++ cin statement is the instance of the class **iostream** and is used to read input from the standard input device which is usually a keyboard.
  The extraction operator(**>>**) is used along with the object **cin** for reading inputs. The extraction operator extracts the data from the object **cin** which is entered using the keyboard.

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

```cpp
int main()
{
    int age;
    cout << "Enter your age:";
    cin >> age;
    cout << "\nYour age is: " << age;
    return 0;
}
```

The above program asks the user to input the age. The object cin is connected to the input device. The age entered by the user is extracted from cin using the extraction operator(**>>**) and the extracted data is then stored in the variable **age** present on the right side of the extraction operator.

- **Un-buffered standard error stream (cerr)**: The C++ cerr is the standard error stream which is used to output the errors. This is also an instance of the iostream class. As cerr in C++ is **un-buffered** so it is used when one needs to display the error message immediately. It does not have any buffer to store the error message and display later.

```cpp
#include <iostream>
using namespace std;
int main()
{
    cerr << "An error occured";
    return 0;
}
```

- **buffered standard error stream (clog)**: This is also an instance of iostream class and used to display errors but unlike cerr the error is first inserted into a buffer and is stored in the buffer until it is not fully filled. or the buffer is not explicitly flushed (using flush()). The error message will be displayed on the screen too.
-

What is a buffer?

A temporary storage area is called buffer. All standard input and output devices contain an input and output buffer. In standard C/C++, streams are buffered, for example in the case of standard input, when we press the key on keyboard, it isn't sent to your program, rather it is buffered by operating system till the time is allotted to that program.

Files and Streams

C++ views each file simply as a sequence of bytes.

When a file is opened, an object is created, and a stream is associated with the object

The streams associated with these objects provide communication channels between a program and a particular file or device.

For example, the cin object (standard input stream object) enables a program to input data from the keyboard or from other devices, the cout object (standard output stream object) enables a program to output data to the screen or other

devices, and the cerr and clog objects (standard error stream objects) enable a program to output error messages to the screen or other devices.

Creating a Sequential File

C++ imposes no structure on files. Thus, a concept like that of a record does not exist in C++. You must structure files to meet the application's requirements. The following example shows how you can impose a simple record structure on a file.

```cpp
// Fig. 14.2: Fig14_02.cpp
// Creating a sequential file.
#include <iostream>
#include <string>
#include <fstream> // contains file stream processing types
#include <cstdlib> // exit function prototype
using namespace std;

int main() {
   // ofstream constructor opens file
   ofstream outClientFile{"clients.txt", ios::out};

   // exit program if unable to create file
   if (!outClientFile) { // overloaded ! operator
      cerr << "File could not be opened" << endl;
      exit(EXIT_FAILURE);
   }

   cout << "Enter the account, name, and balance.\n"
      << "Enter end-of-file to end input.\n? ";

   int account; // the account number
   string name; // the account owner's name
   double balance; // the account balance

   // read account, name and balance from cin, then place in file
   while (cin >> account >> name >> balance) {
      outClientFile << account << ' ' << name << ' ' << balance << endl;
      cout << "? ";
   }
}
```

```
Enter the account, name, and balance.
Enter end-of-file to end input.
? 100 Jones 24.98
? 200 Doe 345.67
? 300 White 0.00
? 400 Stone -42.16
? 500 Rich 224.62
? ^Z
```

Opening a File

writes data to a file, so we open the file for output by creating an ofstream object.

Two arguments are passed to the object's constructor the filename and the file-open mode.

member constant	stands for	access
in	**in**put	Searches for the file and opens it in the read mode only(if the file is found).
out	**out**put	Searches for the file and opens it in the write mode. If the file is found, its content is **overwritten**. If the file is not found, a new file is **created**. Allows you to write to the file.
binary	**binary**	Searches for the file and opens the file(if the file is found) in a binary mode to perform **binary input/output** file operations.
ate	**at e**nd	Searches for the file, opens it and positions the pointer at the end of the file. This mode when used with ios::binary, ios::in and ios::out modes, allows you to modify the content of a file.
app	**app**end	Searches for the file and opens it in the append mode . this mode allows you to append new data to the **end of a file**. If the file is not found, a new file is created.
trunc	**trunc**ate	Searches for the file and opens it to truncate or **deletes all of its content**(if the file is found.

For an ofstream object, the file-open mode can be either ios::out (the default) to output data to a file or ios::app to append data to the end of a file (without modifying any data already in the file).

```
        ofstream outClientFile("clients.txt");
```

Opening a File via the open Member Function

You can create an ofstream object without opening a specific file—in this case, a file can be attached to the object later. For example, the statement

```
      ofstream outClientFile;
      outClientFile.open("clients.txt", ios::out);
```

Testing Whether a File Was Opened Successfully

After creating an ofstream object and attempting to open it, the if statement uses the overloaded ios member function operator! to determine whether the open operation succeeded. Recall that operator! returns true if either the fail bit or the bad bit is set for the stream—in this case, one or both would be set because the open operation failed.

Some possible reasons are:

1. attempting to open a nonexistent file for reading
2. attempting to open a file for reading or writing from a directory that you don't have permission to access, and
3. opening a file for writing when no disk space is available.

```
#include <iostream>
#include<fstream>
using namespace std;
int main()
{
    ofstream outClientFile;
    outClientFile.open("clients.txt", ios::out);
    if(!outClientFile)
```

```
    {
        cerr << "error";
        exit(EXIT_FAILURE);
    }
}
```

The argument to exit is returned to the environment from which the program was invoked. Passing EXIT_SUCCESS (defined in) to exit indicates that the program terminated normally; passing any other value (in this case EXIT_FAILURE) indicates that the program terminated due to an error.

Processing Data

```
1. #include <iostream>
2. #include<fstream>
3. using namespace std;
4. int main()
5. {
6.    ofstream outClientFile;
7.    outClientFile.open("clients.txt", ios::out);
8.    if(!outClientFile)
9.    {
10.        cerr << "error";
11.        exit(EXIT_FAILURE);
12.    }
13.    cout << "Enter the account , name , and balance.\n"<<
14.    "Enter end-of-file to end input.\n?";
15.    int account;
16.    string name;
17.    double balance;
18.    while (cin>>account>>name>>balance)
19.    {
20.        outClientFile<<account<<' ' << name<<' '<<balance << endl;
21.        cout << "? ";
22.    }
23. }
```

if opens the file successfully, the program begins processing data. Lines 13–14 prompt the user to enter either the various fields for each record or the end-of-file indicator when data entry is complete. Figure 14.4 lists the keyboard combinations for entering end-of-file for various computer systems.

Computer system	Keyboard combination
UNIX/Linux/Mac OS X	*<Ctrl-d>* (on a line by itself)
Microsoft Windows	*<Ctrl-z>* (followed by pressing *Enter*)

Closing a File

Once the user enters the end-of-file indicator, main terminates. This implicitly invokes outClientFile 's destructor, which closes the clients.txt file. You also can close the ofstream object explicitly, using member function close as follows:

```
    outClientFile.close();
```

Always close a file as soon as it's no longer needed in a program

Reading Data from a Sequential File

Files store data so it may be retrieved for processing when needed.

We now discuss how to read data sequentially from a file.

Creating an ifstream object opens a file for input. The ifstream constructor can receive the filename and the file-open mode as arguments

an ifstream object called inClientFile and associates it with the clients.txt file. The arguments in parentheses are passed to the ifstream constructor, which opens the file and establishes a "line of communication" with the file.

```
1  // Fig. 14.5: Fig14_05.cpp
2  // Reading and printing a sequential file.
3  #include <iostream>
4  #include <fstream> // file stream
5  #include <iomanip>
6  #include <string>
7  #include <cstdlib>
8  using namespace std;
9
10 void outputLine(int, const string&, double); // prototype
11
12 int main() {
13    // ifstream constructor opens the file
14    ifstream inClientFile("clients.txt", ios::in);
15
16    // exit program if ifstream could not open file
17    if (!inClientFile) {
18       cerr << "File could not be opened" << endl;
19       exit(EXIT_FAILURE);
20    }
21
22    cout << left << setw(10) << "Account" << setw(13)
23       << "Name" << "Balance\n" << fixed << showpoint;
24
25    int account; // the account number
26    string name; // the account owner's name
27    double balance; // the account balance
28
29    // display each record in file
30    while (inClientFile >> account >> name >> balance) {
31       outputLine(account, name, balance);
32    }
33 }
34
35 // display single record from file
36 void outputLine(int account, const string& name, double balance) {
37    cout << left << setw(10) << account << setw(13) << name
38       << setw(7) << setprecision(2) << right << balance << endl;
39 }
```

```
Account   Name         Balance
100       Jones          24.98
200       Doe           345.67
300       White           0.00
400       Stone         -42.16
500       Rich          224.62
```

Opening a File for Input

Objects of class ifstream are opened for input by default, so the statement

```
    ifstream inClientFile("clients.txt");
```

Before attempting to retrieve data from the file , uses the condition !inClientFile to determine whether the file was opened successfully.

Reading from the File

Line 30 reads a set of data (i.e., a record) from the file. After line 30 executes the first time, account has the value 100 , name has the value "Jones" and balance has the value 24.98 . Each time line 30 executes, it reads another record into the variables account , name and balance . Line 31 displays the records, using function outputLine (lines 36–39), which uses parameterized stream manipulators to format the data for display. When the end of file is reached, the implicit call to operator bool in the while condition returns false , the ifstream destructor closes the file and the program terminates.

File-Position Pointers

Programs normally read sequentially from the beginning of a file and read all the data consecutively until the desired data is found. It might be necessary to process the file sequentially several times (from the beginning) during the execution of a program. istream and ostream provide member functions—seekg (“seek get”) and seekp (“seek put”), respectively—to reposition the file-position pointer (the byte number of the next byte in the file to be read or written). Each istream object has a get pointer, which indicates the byte number in the file from which the next input is to occur, and each ostream object has a put pointer, which indicates the byte number in the file at which the next output should be placed. The statement

```
inClientFile.seekg(0);
```

repositions the file-position pointer to the beginning of the file (location 0) attached to inClientFile . The argument to seekg is an integer. **A second** argument can be specified to indicate the **seek** direction, which can be ios::**beg** (the default) for positioning relative to the beginning of a stream, ios::**cur** for positioning relative to the current position in a stream or ios::end for positioning backward relative to the end of a stream. The file-position pointer is an integer value that specifies the location in the file as a number of bytes from the file’s starting location

member constant	seeking relative to
beg	**beginning** of sequence.
cur	**current** position within sequence.
end	**end** of sequence.

```
// position to the nth byte of fileObject (assumes ios::beg)
fileObject.seekg(n);
// position n bytes in fileObject
fileObject.seekg(n, ios::cur);
// position n bytes back from end of fileObject
fileObject.seekg(n, ios::end);
// position at end of fileObject
fileObject.seekg(0, ios::end);
```

The same operations can be performed using ostream member function seekp . Member functions tellg and tellp are provided to return the current locations of the get and put pointers, respectively. The following statement assigns the get file-position pointer value to variable location of type long

```
    long int location = outClientFile.tellp();
```

	Function	Description
ofstream fstream	Seekg(pos,mode)	sets the position of the next character to be extracted from the **input stream** from a given file
	tellg()	The tellg() function is used with **input streams**, and returns the current "get" position of the pointer in the stream.
ifstream fstream	seekp(pos, mode)	The seekp(pos) method of ostream in C++ is used to set the position of the pointer in the **output sequence** with the specified position
	tellp()	The tellp() function is used with **output streams**, and returns the current "put" position of the pointer in the stream

File state

iostate value (member constant)	indicates	functions to check state flags				
		good()	eof()	fail()	bad()	rdstate()
goodbit	No errors (zero value iostate)	true	false	false	false	goodbit
eofbit	End-of-File reached on input operation	false	true	false	false	eofbit
failbit	Logical error on i/o operation	false	false	true	false	failbit
badbit	Read/writing error on i/o operation	false	false	true	true	badbit

Width() and Fill

```cpp
// field width
#include <iostream>     // std::cout, std::left

int main() {
    std::cout << 100 << '\n';
    std::cout.width(10);
    std::cout << 100 << '\n';
    std::cout.fill('x');
    std::cout.width(15);
    std::cout << std::left << 100 << '\n';
    return 0;
}
```

clear

Sets a new value for the stream's internal *error state flags*.

```
/ clearing errors
```

```cpp
#include <iostream>     // std::cout
#include <fstream>      // std::fstream

int main () {
  char buffer [80];
  std::fstream myfile;

  myfile.open ("test.txt",std::fstream::in);

  myfile << "test";
  if (myfile.fail())
  {
    std::cout << "Error writing to test.txt\n";
    myfile.clear();
  }

  myfile.getline (buffer,80);
  std::cout << buffer << " successfully read from file.\n";

  return 0;
}
```

rdbuf

```cpp
#include <iostream>
#include <fstream>
#include <string>
using namespace  std;
int main()
{
    string filepath = "C:\\Users\\amirr\\Desktop\\AP_99\\Quize\\text.txt";
    fstream File1(filepath, ios::out | ios::app | ios::in );
    File1.seekp(1);
    cout << File1.tellp() << endl;
    cout << File1.tellg() << endl;
    string ss;
    cout << File1.rdbuf();

    if (File1.is_open())
    {
        string str;
        getline(cin, str);
        File1 << str << "\n";
    }
    File1.close();
}
```

Reading and Writing Quoted Text

Many text files contain quoted text, such as "C++ How to Program" . For example, in files representing HTML5 web pages, attribute values are enclosed in quotes. If you're building a web browser to display the contents of such a web page, you must be able to read those quoted strings and remove the quotes

Suppose you need to read from a text file, but with each account's data formatted as follows:

100 "Janie Jones" 24.98

Recall that the stream extraction operator >> treats white space as a delimiter.

```
inClientFile >> account >> name >> balance
```

the first stream extraction reads 100 into the int variable account and the second reads only "Janie into the string variable name (the opening double quote would be part of the string in name). The third stream extraction fails while attempting to read a value for the double variable balance , because the next token (i.e., piece of data) in the input stream— Jones"— is not a double .

Reading Quoted Text

C++14's new stream manipulator—quoted (header)— enables a program to read quoted text from a stream, including any white space characters in the quoted text, and discards the double quote delimiters. For example, if we read the preceding data using the expression

```
#include <iostream>
#include<fstream>
#include<iomanip>
using namespace std;
int main()
{
    int account;
    string name;
    double balance;
    ifstream inClientFile;
    inClientFile.open("clients.txt", ios::in);
    inClientFile >> account >> quoted(name) >> balance;
}
```

Writing Quoted Text

Similarly, you can write quoted text to a stream. For example, if name contains Janie Jones , the statement

```
#include <iostream>
#include<fstream>
#include<iomanip>
using namespace std;
```

```cpp
int main()
{
    int account;
    string name;
    double balance;
    ofstream ouClientFile;
    ouClientFile.open("clients.txt", ios::out);
    ouClientFile << account << quoted(name) << balance;
}
```

Writing Bytes with ostream Member Function write

Writing the integer number to a file using the statement

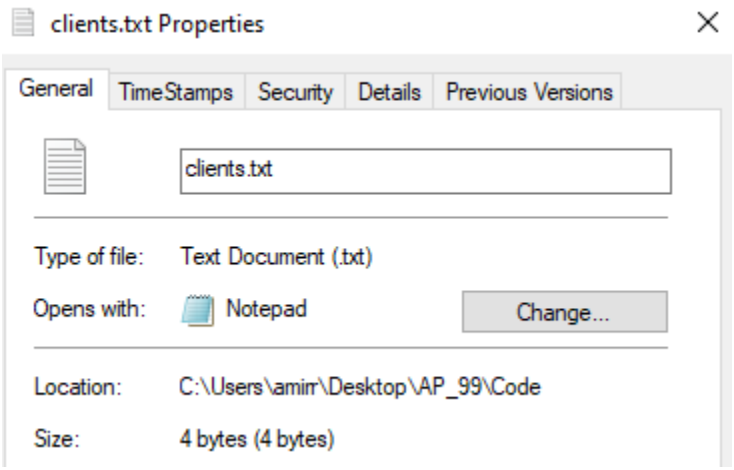
```
outFile << number;
```

for a four-byte integer could output 1 to 11 bytes—up to 10 digits for a number in the range –2,147,483,647 to 2,147,483,647, plus a sign for a negative number. Instead, we can use the statement

```
write( (char *) & ob, sizeof(ob));
```

which always writes in binary the four bytes used that represent the integer number . Function write treats its first argument as a group of bytes by viewing the object in memory as a const char* , which is a pointer to a byte (recall that a char is 1 byte). Starting from that location, function write outputs the number of bytes specified by its second argument—an integer of type size_t . As we'll see, istream function read can subsequently be used to read the four bytes back into integer variable number .

```cpp
#include <iostream>
#include<fstream>
#include<iomanip>
using namespace std;
int main()
{
    ofstream ouClientFile;
    ouClientFile.open("clients.txt", ios::out);
    int a = 5;
    ouClientFile.write( (char*)&a ,sizeof(int));
    ouClientFile.close();
}
```



Credit-Processing Program

```
#ifndef CLIENTDATA_H
#define CLIENTDATA_H

#include <string>

class ClientData {
public:
   // default ClientData constructor
   ClientData(int = 0, const std::string& = "",
      const std::string& = "", double = 0.0);

   // accessor functions for accountNumber
   void setAccountNumber(int);
   int getAccountNumber() const;

   // accessor functions for lastName
   void setLastName(const std::string&);
   std::string getLastName() const;

   // accessor functions for firstName
   void setFirstName(const std::string&);
   std::string getFirstName() const;

   // accessor functions for balance
   void setBalance(double);
   double getBalance() const;
private:
   int accountNumber;
```

```
   char lastName[15];
   char firstName[10];
   double balance;
};

#endif
```

```
#include <string>
#include "ClientData.h"
using namespace std;

// default ClientData constructor
ClientData::ClientData(int accountNumberValue, const string& lastName,
   const string& firstName, double balanceValue)
   : accountNumber(accountNumberValue), balance(balanceValue) {
   setLastName(lastName);
   setFirstName(firstName);
}

// get account-number value
int ClientData::getAccountNumber() const {return accountNumber;}

// set account-number value
void ClientData::setAccountNumber(int accountNumberValue) {
   accountNumber = accountNumberValue; // should validate
}

// get last-name value
string ClientData::getLastName() const {return lastName;}

// set last-name value
void ClientData::setLastName(const string& lastNameString) {
   // copy at most 15 characters from string to lastName
   size_t length{lastNameString.size()};
   length = (length < 15 ? length : 14);
   lastNameString.copy(lastName, length);
   lastName[length] = '\0'; // append null character to lastName
}

// get first-name value
string ClientData::getFirstName() const {return firstName;}

// set first-name value
void ClientData::setFirstName(const string& firstNameString) {
   // copy at most 10 characters from string to firstName
   size_t length{firstNameString.size()};
```

```
   length = (length < 10 ? length : 9);
   firstNameString.copy(firstName, length);
   firstName[length] = '\0'; // append null character to firstName
}

// get balance value
double ClientData::getBalance() const {return balance;}

// set balance value
void ClientData::setBalance(double balanceValue) {balance = balanceValue;}
```

Objects of class string do not have uniform size, rather they use dynamically allocated memory to accommodate strings of various lengths

We must maintain fixed-length records, so class ClientData stores the client's first and last name in fixed-length char arrays

Opening a File for Output in Binary Mode

```
#include <iostream>
#include <fstream>
#include <cstdlib>
#include "ClientData.h" // ClientData class definition
using namespace std;

int main() {
   ofstream outCredit{"credit.dat", ios::out | ios::binary};

   // exit program if ofstream could not open file
   if (!outCredit) {
      cerr << "File could not be opened." << endl;
      exit(EXIT_FAILURE);
   }
   ClientData blankClient; // constructor zeros out each data member

   // output 100 blank records to file
   for (int i{0}; i < 100; ++i) {
      outCredit.write(
         reinterpret_cast<const char*>(&blankClient), sizeof(ClientData));
   }
}
```

Opening a File for Input and Output in Binary Mode

```
#include <iostream>
#include <fstream>
#include <cstdlib>
#include "ClientData.h" // ClientData class definition
using namespace std;
```

```
int main() {
   fstream outCredit{"credit.dat", ios::in | ios::out | ios::binary};

   // exit program if fstream cannot open file
   if (!outCredit) {
      cerr << "File could not be opened." << endl;
      exit(EXIT_FAILURE);
   }

   cout << "Enter account number (1 to 100, 0 to end input)\n? ";

   int accountNumber;
   string lastName;
   string firstName;
   double balance;

   cin >> accountNumber; // read account number

   // user enters information, which is copied into file
   while (accountNumber > 0 && accountNumber <= 100) {
      // user enters last name, first name and balance
      cout << "Enter lastname, firstname and balance\n? ";
      cin >> lastName;
      cin >> firstName;
      cin >> balance;
      // create ClientData object
      ClientData client{accountNumber, lastName, firstName, balance};

      // seek position in file of user-specified record
      outCredit.seekp(
         (client.getAccountNumber() - 1) * sizeof(ClientData));
      // write user-specified information in file
      outCredit.write(
         reinterpret_cast<const char*>(&client), sizeof(ClientData));

      // enable user to enter another account
      cout << "Enter account number\n? ";
      cin >> accountNumber;
   }
}
```

writes data to the file credit.dat and uses the combination of fstream functions seekp and write to store data at exact locations in the file. Function seekp sets the put file-position pointer to a specific position in the file, then function write outputs the data. Line 6 includes the header ClientData.h defined, so the program can use ClientData objects.

Opening a File for Input and Output in Binary Mode

uses the fstream object outCredit to open the existing credit.dat file. The file is opened for input and output in binary mode by combining the file-open modes ios::in , ios::out and ios::binary . Note that because we include ios::in , if the file does not exist, an error will occur and the file will not be opened. Opening the existing credit.dat file in this manner ensures that this program can manipulate the records written to the file by the program, rather than creating the file from scratch.

Reading from a RandomAccess File Sequentially

In the previous sections, we created a random-access file and wrote data to that file. In this section, we develop a program that reads the file sequentially and prints only those records that contain data. These programs produce an additional benefit. See if you can determine what it is; we'll reveal it at the end of this section

The istream function read inputs a specified number of bytes from the current position in the stream into an object. For example, line 28 (Fig. 14.12) reads the number of sizeof(ClientData) bytes from the file associated with inCredit and stores the data in client . Function read requires a first argument of type char *. Since &client is of type ClientData *, &client must be cast to char * using the cast operator reinterpret_cast .

```
#include <iostream>
#include <iomanip>
#include <fstream>
#include <cstdlib>
#include "ClientData.h" // ClientData class definition
using namespace std;

void outputLine(ostream&, const ClientData&); // prototype

int main() {
   ifstream inCredit{"credit.dat", ios::in | ios::binary};

   // exit program if ifstream cannot open file
   if (!inCredit) {
      cerr << "File could not be opened." << endl;
      exit(EXIT_FAILURE);
   }

   // output column heads
   cout << left << setw(10) << "Account" << setw(16) << "Last Name"
      << setw(11) << "First Name" << setw(10) << right << "Balance\n";

   ClientData client; // create record

   // read first record from file
   inCredit.read((char*)(&client), sizeof(ClientData));

   // read all records from file
   while (inCredit) {
      // display record
      if (client.getAccountNumber() != 0) {
```

```
         outputLine(cout, client);
      }

      // read next from file
      inCredit.read((char*)(&client), sizeof(ClientData));
   }
}

// display single record
void outputLine(ostream& output, const ClientData& record) {
   output << left << setw(10) << record.getAccountNumber()
      << setw(16) << record.getLastName()
      << setw(11) << record.getFirstName()
      << setw(10) << setprecision(2) << right << fixed
      << showpoint << record.getBalance() << endl;
}
```

Binary File Example

```
#include<iostream>
#include<fstream>
#include<cstdio>
using namespace std;

class Student
{
    int admno;
    char name[50];
public:
    void setData()
    {
        cout << "\nEnter admission no. ";
        cin >> admno;
        cout << "Enter name of student ";
        cin.getline(name,50);
    }

    void showData()
    {
        cout << "\nAdmission no. : " << admno;
        cout << "\nStudent Name : " << name;
    }

    int retAdmno()
    {
        return admno;
    }
```

```
};

/*
* function to write in a binary file.
*/

void write_record()
{
    ofstream outFile;
    outFile.open("student.dat", ios::binary | ios::app);

    Student obj;
    obj.setData();

    outFile.write((char*)&obj, sizeof(obj));

    outFile.close();
}

/*
* function to display records of file
*/


void display()
{
    ifstream inFile;
    inFile.open("student.dat", ios::binary);

    Student obj;

    while(inFile.read((char*)&obj, sizeof(obj)))
    {
        obj.showData();
    }

    inFile.close();
}

/*
* function to search and display from binary file
*/

void search(int n)
{
    ifstream inFile;
    inFile.open("student.dat", ios::binary);
```

```
    Student obj;

    while(inFile.read((char*)&obj, sizeof(obj)))
    {
        if(obj.retAdmno() == n)
        {
            obj.showData();
        }
    }

    inFile.close();
}

/*
* function to delete a record
*/

void delete_record(int n)
{
    Student obj;
    ifstream inFile;
    inFile.open("student.dat", ios::binary);

    ofstream outFile;
    outFile.open("temp.dat", ios::out | ios::binary);

    while(inFile.read((char*)&obj, sizeof(obj)))
    {
        if(obj.retAdmno() != n)
        {
            outFile.write((char*)&obj, sizeof(obj));
        }
    }

    inFile.close();
    outFile.close();

    remove("student.dat");
    rename("temp.dat", "student.dat");
}

/*
* function to modify a record
*/

void modify_record(int n)
```

```
{
    fstream file;
    file.open("student.dat",ios::in | ios::out);

    Student obj;

    while(file.read((char*)&obj, sizeof(obj)))
    {
        if(obj.retAdmno() == n)
        {
            cout << "\nEnter the new details of student";
            obj.setData();

            int pos = -1 * sizeof(obj);
            file.seekp(pos, ios::cur);

            file.write((char*)&obj, sizeof(obj));
        }
    }

    file.close();
}

int main()
{
    //Store 4 records in file
    for(int i = 1; i <= 4; i++)
       write_record();

    //Display all records
    cout << "\nList of records";
    display();

    //Search record
    cout << "\nSearch result";
    search(100);

    //Delete record
    delete_record(100);
    cout << "\nRecord Deleted";

    //Modify record
    cout << "\nModify Record 101 ";
    modify_record(101);

    return 0;
}
```

Write Json

```cpp
#include "json.hpp"
#include <fstream>
#include <iostream>
#include <iomanip>
using json = nlohmann::json;
using namespace std;
enum Gender
{
    Male,
    Female
};
struct Student
{
    int id;
    string fname;
    string lname;
    int age;
    Gender gender;
    json parser()
    {
        json newParsering;
        newParsering["id"] = id;
        newParsering["name"]["fname"] = fname;
        newParsering["name"]["lname"] = lname;
        newParsering["age"] = age;
        newParsering["gender"] = gender == Gender::Male ? "Male" : "Female";

     return newParsering;
    }
    void dparser(json newParsering)
    {
        id = newParsering["id"];
        fname = newParsering["name"]["fname"] ;
        lname =   newParsering["name"]["lname"];
        age =  newParsering["age"];
        gender =  newParsering["gender"] == "Male"  ? Gender::Male : Gender::Female;
    return ;
    }

};
int main() {
    std::ofstream writer("file.json" , ios::out);
    if(!writer.is_open())
    {
        cerr << "Error";
```

```
        exit(EXIT_FAILURE);
    }
    Student st1{1,"Amirreza","Tavakoli",12,Gender::Male};
    auto js  =  st1.parser();
    writer <<  js.dump(4);
    Student st2{2,"Amirreza","Tavakoli",12,Gender::Male};
    auto js2  =  st1.parser();
    writer << js2.dump(4);
    writer.close();

    Student* Allstudent;
    std::ifstream reader("file.json" , ios::in);
    if(!reader.is_open())
    {
        cerr << "Error";
        exit(EXIT_FAILURE);
    }
    json jsr;
    reader.seekg(0,ios::end);
    long size = reader.tellg();
    reader.seekg(0,ios::beg);
    while (!reader.eof())
    {
        try
        {
            reader >> jsr;
            Student tmp;
            tmp.dparser(jsr);
        }
        catch(...)
        {
            // ignor
        }
    }
    reader.close();
    return 0;
}
```

Read txt File Line

```
#include <fstream>
#include <iostream>
#include<string>
using namespace std;
int main() {
    string text = "";
    ifstream reader("Text.txt" , ios::in);
    if(!reader)
```

```
    {
        cerr << "error";
        exit(EXIT_FAILURE);
    }
    while (!reader.eof())
    {
        try
        {
            string temp;
            getline(reader,temp);
            text += temp + "\n";
        }
        catch(...)
        {
           // ignor
        }
    }
    text = text.substr(0,text.length()-1);// remove last \n
    return 0;
}
```

Advanced

➢ stringstream in C++ and its applications

A stringstream associates a string object with a stream allowing you to read from the string as if it were a stream (like cin).

clear() — to clear the stream
str() — to get and set string object whose content is present in stream.
operator << — add a string to the stringstream object.
operator >> — read something from the stringstream object,

```
// CPP program to count words in a string
// using stringstream.
#include <bits/stdc++.h>
using namespace std;

int countWords(string str)
{
    // breaking input into word using string stream
    stringstream s(str); // Used for breaking words
    string word; // to store individual words

    int count = 0;
    while (s >> word)
        count++;
    return count;
}

// Driver code
int main()
{
    string s = "geeks for geeks geeks "
            "contribution placements";
    cout << " Number of words are: " << countWords(s);
    return 0;
}
```

Consider the following short code:

```
#include <sstream>
#include <string>
using namespace std;
int main()
{
ostringstream ss;
```

```
ss << "the answer to everything is " << 42;
const string result = ss.str();
}
```

The line

```
ostringstream ss;
```

creates such an object. This object is first manipulated like a regular stream:

```
ss << "the answer to everything is " << 42;
```

Following that, though, the resulting stream can be obtained like this:

```
const string result = ss.str();
```

(the string result will be equal to "the answer to everything is 42")

```
#include <string>
#include <fstream>
#include <sstream>

using namespace std;

int main()
{
    ostringstream text;
    ifstream in_file("Text.txt");

    text << in_file.rdbuf();

    string str = text.str();

    string str_search = "Fortunato";

    string str_replace = "NotFortunato";

    size_t pos = str.find(str_search);

    str.replace(pos, string(str_search).length(), str_replace);

    in_file.close();
}
```

➢ std::istream_iterator and std::ostream_iterator in C++ STL

The STL is a very powerful library in C++. It is strongly built on the principles of template programming.
The STL library has three main components :

1. **Containers:** These classes define the data structures which are used to contain the data. The data may be stored in linked lists, or trees or arrays. The containers provided in the STL are vector, dequeue, list, forward list, set, multiset, map and multimap.
2. **Algorithms:** The STL library also provides us functions to process the data stored in containers. These functions are provided in the *algorithm* header file
3. **Iterators:** Iterators are the link between the Containers and the Algorithms. They are the common interface to these classes. An Iterator is an object which can be used to iterator over the elements in a container. Thus iterators are used by Algorithms to modify the containers.

All three components are so designed that they confirm to the principles of data abstraction. Thus any object which holds data and behaves like a container, is a container. Similarly, any iterator which sweeps through the elements in a container is an iterator.

If an iterator can be used to access elements of a data container, then what about streams? In keeping with the design, Streams too are data containers and so C++ provides us with iterators to iterate over the elements present in any stream. These iterators are called Stream Iterators. To use these iterators the iterator header file must be included.

Stream iterators are either input stream iterator or output stream iterator. The classes for these iterators are istream_iterator and ostream_iterator. These iterators behave like input iterators and output iterators respectively .

➢ istream_iterator

Syntax :

```
istream_iterator<T>(stream);
```

1. T: Template parameter for specifying type of data
2. stream: The object representing the stream

Using an iterator we can only access elements of one type only.
istream_iterator has a special state end of stream iterator which is acquired when the end of the stream is reached or when an input operation fails. The end of stream iterator is returned by the default constructor.

➢ ostream_iterator

Syntax :

```
ostream_iterator<T>(stream, delim).
```

1. stream: The object representing the stream.
2. T: Template Parameter for data type in the stream

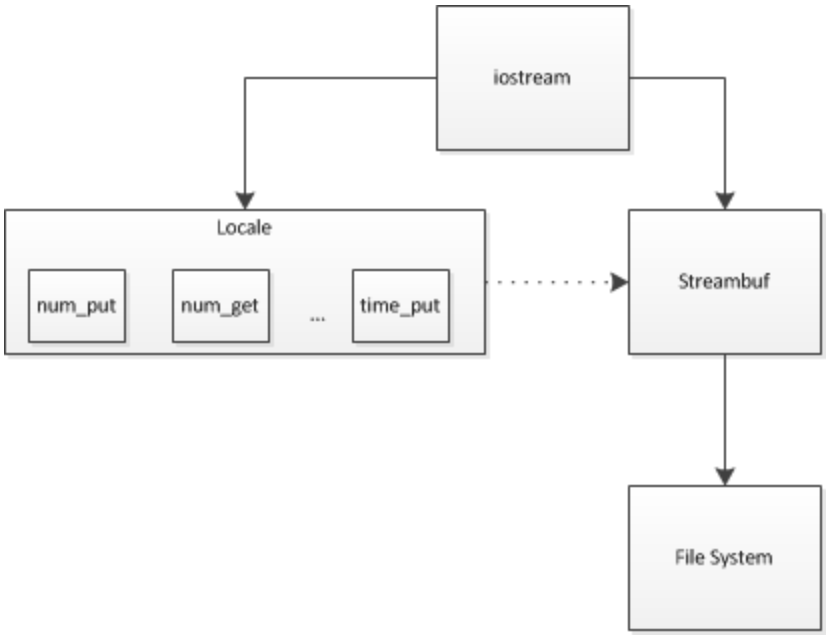
3. delim: An optional char sequence that is used to separate the data items while displaying them.

➢ istreambuf_iterator

1. Istreambuf iterators are input iterators that read successive elements from a stream buffer.
2. This kind of iterator has a special state as an end-of-stream iterator, which is acquired when the end of the stream is reached, and is also the resulting value of a default-constructed object: This value can be used as the end of a range in any function accepting iterator ranges to indicate that the range includes all the elements up to the end of the input buffer.
3. controls input and output to a character sequence

➢ ostreambuf_iterator

1. Ostreambuf iterators are output iterators that write sequentially to an stream buffer.
2. They are constructed from a basic_streambuf object open for writing, to which they become associated.
3. controls input and output to a character sequence



The basic difference between an `istream_iterator` and an `istreambuf_iterator` is that the data coming out of an istreambuf_iterator hasn't gone through (most of the) transformations done by the locale(Main Memory), but data coming out of an `istream_iterator` has been transformed by the locale.

- Note : suggested to
- Use istreambuf_iterator for cin , ifstream , …
- Use ostreambuf_iterator for cout , ofstream , …
- Use istream_iterator for vector , map , …
- Use ostream_iterator for vector , map , …
-

➢ copy in C++ STL

1. . copy(strt_iter1, end_iter1, strt_iter2)

The generic copy function used to **copy** a **range** of elements from one container to another. It takes 3 arguments:

- **strt_iter1 :** The pointer to the beginning of the source container, from where elements have to be started copying.
- **end_iter1 :** The pointer to the end of source container, till where elements have to be copied.
  - The iterator becomes equivalent to an iterator created using the default constructor
- **strt_iter2 :** The pointer to the beginning of destination container, to where elements have to be started copying.

2. copy_n(strt_iter1, num, strt_iter2)

This version of copy gives the freedom to choose how many elements have to be copied in the destination container. IT also takes 3 arguments:

1. **strt_iter1 :** The pointer to the beginning of the source container, from where elements have to be started copying.
2. **num :** Integer specifying how many numbers would be copied to destination container starting from strt_iter1. If a negative number is entered, no operation is performed.
3. **strt_iter2 :** The pointer to the beginning of destination container, to where elements have to be started copying.

```
#include<iostream>
#include<algorithm> // for copy() and copy_n()
#include<vector>
using namespace std;
int main()
{
   // initializing source vector
   vector<int> v1 = { 1, 5, 7, 3, 8, 3 };
   // declaring destination vectors
   vector<int> v2(6);
   vector<int> v3(6);
   // using copy() to copy 1st 3 elements
   copy(v1.begin(), v1.begin()+3, v2.begin());
   // printing new vector
   cout << "The new vector elements entered using copy() : ";
   for(int i=0; i<v2.size(); i++)
```

```
    cout << v2[i] << " ";
    cout << endl;
    // using copy_n() to copy 1st 4 elements
    copy_n(v1.begin(), 4, v3.begin());
    // printing new vector
    cout << "The new vector elements entered using copy_n() : ";
    for(int i=0; i<v3.size(); i++)
    cout << v3[i] << " ";
}
```

3. copy_if(): As the name suggests, this function copies according to the result of a "**condition**".This is provided with the help of a **4th argument**, a **function returning a boolean value**.

This function takes 4 arguments, 3 of them similar to copy() and an additional function, which when returns true, a number is copied, else number is not copied.

4. copy_backward(): This function starts copying elements into the destination container **from backward** and keeps on copying till all numbers are not copied. The copying starts from the "**strt_iter2**" but in the backward direction. It also takes similar arguments as copy().

```
// C++ code to demonstrate the working of copy_if()
// and copy_backward()
#include<iostream>
#include<algorithm> // for copy_if() and copy_backward()
#include<vector>
using namespace std;
int main()
{
  // initializing source vector
  vector<int> v1 = { 1, 5, 6, 3, 8, 3 };
  // declaring destination vectors
  vector<int> v2(6);
  vector<int> v3(6);
  // using copy_if() to copy odd elements
  copy_if(v1.begin(), v1.end(), v2.begin(), [](int i){return i%2!=0;});
  // printing new vector
  cout << "The new vector elements entered using copy_if() : ";
  for(int i=0; i<v2.size(); i++)
  cout << v2[i] << " ";
  cout << endl;
  // using copy_backward() to copy 1st 4 elements
  // ending at second last position
  copy_backward(v1.begin(), v1.begin() + 4, v3.begin()+ 5);
  // printing new vector
  cout << "The new vector elements entered using copy_backward() : ";
  for(int i=0; i<v3.size(); i++)
```

```
  cout << v3[i] << " ";
}
```

Using the copy() function, we can easily transfer data from a stream to a container and vice-versa. Here are a few example programs to demonstrate working with stream iterators

```
// Read a bunch of integers from the input stream
// and print them to output stream

#include <algorithm>
#include <iostream>
#include <iterator>

using namespace std;
int main()
{

    // Get input stream and end of stream iterators
    istreamb_iterator<int> cin_it(cin);
    istream_iterator<int> eos;

    // Get output stream iterators
    ostream_iterator<int> cout_it(cout, " ");

    // We have both input and output iterators, now we can treat them
    // as containers. Using copy function we transfer data from one
    // container to another.
    // Copy elements from input to output using copy function
    copy(cin_it, eos, cout_it);

    return 0;
}
```

➢ Vector

```
// Cpp program to illustrate
// Read a bunch of strings from a file
// sort them lexicographically and print them to output stream

#include <algorithm>
#include <fstream>
#include <iostream>
#include <iterator>
#include <string>
#include <vector>

using namespace std;
```

```cpp
int main()
{

    // Define a vector to store the strings received from input
    vector<string> strings_v;

    // Define the filestream object used to read data from file
    ifstream fin("input_file.txt");

    // Get input stream and end of stream iterators
    istream_iterator<string> fin_it(fin);
    istream_iterator<string> eos;

    // Get output stream iterators
    ostream_iterator<string> cout_it(cout, " ");

    // Copy elements from input to vector using copy function
    copy(fin_it, eos, back_inserter(strings_v));

    // Sort the vector
    sort(strings_v.begin(), strings_v.end());

    // Copy elements from vector to output
    copy(strings_v.begin(), strings_v.end(), cout_it);

    return 0;
}
```

➢ Vector

```cpp
#include <algorithm>
#include <iostream>
#include <iterator>
#include <vector>
using namespace std;
int main()
{
    // Define a vector to store the even integers received from input
    vector<int> vi;
    // Get input stream and end of stream iterators
    istream_iterator<int> cin_it(cin);
    istream_iterator<int> eos;
    // Get output stream iterators
    ostream_iterator<int> cout_it(cout, " ");
    // Copy even integer elements from input to vector using for_each function
    for_each(cin_it, eos, [&](int a) {
        if (a % 2 == 0) {
```

```
            // if a is even push it to vector
            vi.push_back(a);
        }
    });
    // Sort the vector
    sort(vi.begin(), vi.end());
    // Copy elements from vector to output
    copy(vi.begin(), vi.end(), cout_it);
    return 0;
}
```

➢ Reading a file into a buffer at once

Finally, let's read the file from the beginning till the end without stopping at any character, including
whitespaces and newlines. If we know the exact file size or upper bound of the length is acceptable, we can resize the string and then read:

```
#include <algorithm>
#include <iostream>
#include <iterator>
#include <vector>
#include <sstream>
#include <fstream>
using namespace std;
int main()
{
    ostringstream str ;
    ifstream ifs("YourText.txt");
    istreambuf_iterator<char> inputstream (ifs);
    istreambuf_iterator<char> eos;
    std::ostreambuf_iterator<char> outputstream (str);
    copy(inputstream  , eos , outputstream);
    string ss = str.str();
    cout << str.str();
    return 0;
}
```

➢ Reading a file into a buffer at once’

Finally, let's read the file from the beginning till the end without stopping at any character, including
whitespaces and newlines. If we know the exact file size or upper bound of the length is acceptable, we can resize the string and then read:

➢ Copying streams

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

```cpp
using namespace std;
int main() {
    string source;
    string Destination;
    cout << "Enter Source path : ";
    cin >> source;
    cout << "Enter Destination path : ";
    cin >> Destination;
    ifstream reader(source , ios::ate | ios::binary);
    long size = reader.tellg();
    char *buffer = new char[size]();
    reader.seekg(0,ios::beg);
    reader.read(buffer,size);
    reader.close();
    ofstream writer(Destination , ios::out | ios::binary);
    writer.write(buffer,size);
    writer.close();
    return 0;
}
```

➢ Simple way

```cpp
#include <algorithm>
#include <iostream>
#include <iterator>
#include <vector>
#include <sstream>
#include <fstream>
using namespace std;
int main()
{
    ifstream ifs("Text.txt");
    istreambuf_iterator<char> inputstream (ifs);
    istreambuf_iterator<char> eos;
    ofstream ofs("desText.txt");
    ostreambuf_iterator<char> output(ofs);
    std::copy(inputstream  , eos , output);}
```

➢ Read Buffer

```cpp
#include<iostream>
#include<fstream>
#include<cstdio>
#include<algorithm>
#include<iterator>
#include<sstream>
#include<string>
```

```cpp
using namespace std;
#define BUFFERLENGTH 4
int main()
{
    char buffer[BUFFERLENGTH] = { 0 };
    long Current = 0;
    ifstream ifs("Text.txt");
    long size = ifs.seekg(0, ios::end).tellg();
    ifs.seekg(0, ios::beg);
    istreambuf_iterator<char> inputstream(ifs);
    istreambuf_iterator<char> eos;
    cout << buffer;
    while (Current < size)
    {
        long readindex = size - Current < BUFFERLENGTH ? size - Current : BUFFERLENGTH;
        copy_n(inputstream, readindex, buffer);
        ifs.seekg(1, ios::cur);
        string str(buffer, readindex);
        cout << str;
        cout << endl;
        Current += readindex;
    }
    return 0;
}
```

➢ Way 1 :Read and Write Buffer

```cpp
#include<iostream>
#include<fstream>
#include<cstdio>
#include<algorithm>
#include<iterator>
#include<sstream>
#include<string>
using namespace std;
int main()
{
    long Current = 0;
    ifstream ifs("Text.txt",ios::binary);

    long size = ifs.seekg(0, ios::end).tellg();
    ifs.seekg(0, ios::beg);
    istreambuf_iterator<char> inputstream(ifs);
    istreambuf_iterator<char> eos;

    ofstream ofs("NewText.txt", ios::binary);
    ostreambuf_iterator<char> outfile(ofs);
```

```
    while (Current < size)
    {
        long readindex = size - Current < BUFFERLENGTH ? size - Current : BUFFERLENGTH;
        copy_n(inputstream, readindex, buffer);
        ifs.seekg(1, ios::cur);
        ofs.write(buffer, readindex);
        Current += readindex;
    }
    ofs.close();
    return 0;
}
```

➢ Way 2 :Read and Write Buffer

```
#include<iostream>
#include<fstream>
#include<cstdio>
#include<algorithm>
#include<iterator>
#include<sstream>
#include<string>

using namespace std;
#define BUFFERLENGTH 4
int main()
{
    char buffer[BUFFERLENGTH] = { 0 };
    long Current = 0;
    ifstream ifs("Text.txt",ios::binary);

    long size = ifs.seekg(0, ios::end).tellg();
    ifs.seekg(0, ios::beg);
    istreambuf_iterator<char> inputstream(ifs);
    istreambuf_iterator<char> eos;

    ofstream ofs("NewText.txt",ios::binary);
    ostreambuf_iterator<char> outfile(ofs);

    while (Current < size)
    {
        long readindex = size - Current < BUFFERLENGTH ? size - Current : BUFFERLENGTH;
        copy_n(inputstream, readindex, outfile);
        ifs.seekg(1,ios::cur);
        Current += readindex;
    }
    ofs.close();
    return 0;
}
```

➢ Arrays

```cpp
#include <algorithm>
#include <iostream>
#include <iterator>
#include <vector>
#include <sstream>
#include <fstream>

using namespace std;
int main()
{
    ifstream ifs("Text.txt");
    istream_iterator<char> inputstream (ifs);
    istream_iterator<char> eos;

    int arr[100];
    std::copy(inputstream, eos, arr);
    cout << arr;
}
```

➢ Parsing files into STL containers

```cpp
#include <algorithm>
#include <iostream>
#include <iterator>
#include <vector>
#include <sstream>
#include <fstream>

using namespace std;
int main()
{
    ifstream ifs("Text.txt");
    istream_iterator<string> inputstream (ifs);
    istream_iterator<string> eos ;

    ofstream ofs("Text2.txt");
    ostream_iterator<string> outputstream(ofs,"\n");

    vector<string> v(100);
    copy(inputstream , eos , v.begin());

    copy(v.begin() , v.end() , outputstream);
    ofs.close();
}
```

➢ Saving Json

```cpp
#include<iostream>
#include<fstream>
#include<cstdio>
#include<algorithm>
#include<iterator>
#include "json.hpp"
#include<sstream>
#include<string>
using namespace std;
using json = nlohmann::json;

int main()
{

    json newjson;
    ofstream ofs("Text.json");
    newjson["FName"] = "Amirreza";
    newjson["LName"] = "Tavekoli";
    newjson["age"] = 12;

    stringstream savingsream;
    savingsream <<  newjson;

    istream_iterator<char> inputstream(savingsream) ;
    istream_iterator<char> eos ;
    ostream_iterator<char> outputstream(ofs);
    copy(inputstream , eos , outputstream);
    ofs.close();

    return 0;
}
```

➢ C++ rename()

The rename() function in C++ renames a specified file.

rename() Prototype

```cpp
int rename( const char *oldname, const char *newname );
```

The `rename()` function takes a two arguments: `oldname`, `newname` and returns an integer value.

It renames the file represented by the string pointed to by oldname to the string pointed to by `newname`.

It is defined in <cstdio> header file.

rename() Parameters

- `oldname`: Pointer to the string containing the old name of the file along with the path to rename.
- `newname`: Pointer to the string containing the new name of the file along with the path.

rename() Return value

The rename() function returns:

- Zero if the file is successfully renamed.
- Non zero if error occurs.

Example 1: How rename() function works

```
#include <iostream>
#include <cstdio>

using namespace std;

int main()
{
        char oldname[] = "file_old.txt";
        char newname[] = "file_new.txt";

        /*      Deletes the file if exists */
        if (rename(oldname, newname) != 0)
                perror("Error renaming file");
        else
                cout << "File renamed successfully";

        return 0;
}
```

When you run the program, the output will be:

- If the file is renamed successfully:

```
File renamed successfully
```

- If the file is not present:

```
Error renaming file: No such file or directory
```

The `rename()` function can also be used to move a file to a different location. This can be done by providing a different path for new name of the file.

Example 2: rename() function to move a file

```
#include <iostream>
#include <cstdio>

using namespace std;

int main()
{
        char oldname[] = "C:\\Users\\file_old.txt";
        char newname[] = "C:\\Users\\New Folder\\file_new.txt";

        /*      Deletes the file if exists */
        if (rename(oldname, newname) != 0)
                perror("Error moving file");
        else
                cout << "File moved successfully";

        return 0;
}
```

When you run the program, the output will be:

- If the file is moved successfully:

```
File moved successfully
```

- If the file is not present:

```
Error moving file: No such file or directory
```

➢ C++ remove()

The remove() function in C++ deletes a specified file.

remove() prototype

```
int remove(const char* filename);
```

The `remove()` function takes a single argument filename and returns an integer value. It deletes the file pointed by the parameter.
Incase the file to be deleted is opened by a process, the behaviour of `remove()` function is implementation-defined.
In POSIX systems, if the name was the last link to a file, but any processes still have the file open, the file will remain in existence until the last running process closes the file. In windows, the file won't be allowed to delete if it remains open by any process.

It is defined in <cstdio> header file.

remove() Parameters

`filename`: Pointer to the string containing the name of the file along with the path to delete.

remove() Return value

The remove() function returns:

- Zero if the file is successfully deleted.
- Non zero if error occurs.

Example: How remove() function works

```
#include <iostream>
```

```cpp
#include <cstdio>

using namespace std;

int main()
{
	char filename[] = "C:\\Users\\file.txt";

	/*	Deletes the file if exists */
	if (remove(filename) != 0)
		perror("File deletion failed");
	else
		cout << "File deleted successfully";

	return 0;
}
```

When you run the program, the output will be:

```
If the file is deleted successfully:
File deleted successfully
If the file is not present:
File deletion failed: No such file or directory
```

➢ Temp File (way 1)

```cpp
#define _CRT_SECURE_NO_WARNINGS
#include <iostream>
#include <cstdio>
#include<iostream>
#include<fstream>
#include<cstdio>
#include<algorithm>
#include<iterator>
#include<sstream>
#include<string>
#include <cstdio>

using namespace std;
#define BUFFERLENGTH 1024*1000000
void ReadDataANDSaveTMP(string source, string despath)
{
```

```
    string path = despath;
    ifstream ifs(source, ios::binary);
    long size = ifs.seekg(0, ios::end).tellg();
    ifs.seekg(0, ios::beg);
    istreambuf_iterator<char> inputstream(ifs);
    long Current = 0;
    ofstream ofstmp(path, ios::binary);
    ostreambuf_iterator<char> outfile(ofstmp);
    while (Current < size)
    {
        long readindex = size - Current < BUFFERLENGTH ? size - Current : BUFFERLENGTH;
        copy_n(inputstream, readindex, outfile);
        ifs.seekg(1, ios::cur);
        Current += readindex;
    }
    ofstmp.close();
    ifs.close();
}

string GetFileName(string path)
{
    for (int i = path.length(); i >= 0; i--)
        if (path[i] == '\\')
            return path.substr(i+1);
    return path;
}
string GetFileExtention(string path)
{
    string filename = GetFileName(path);
    size_t i = filename.rfind('.', filename.length());
    if (i != string::npos) {
        return(filename.substr(i , filename.length() - i));
    }
}
string GetFileNameWhitOutExtention(string path)
{
    string filename = GetFileName(path);
    size_t i = filename.rfind('.', filename.length());
    if (i != string::npos) {
        return(filename.substr(0, i));
    }
}
inline string TempPath(string Filename,string Extention = ".tmp")
{
    return Filename + Extention;
}
inline bool IsFileExist(const string& name) {
```

```
    ifstream f(name.c_str());
    return f.good();
}
bool ConvertTmpToStableFile(const string& tmppath , const string& destinationpath)
{
    if (!IsFileExist(tmppath))
        return;
    if (rename(tmppath.c_str(), destinationpath.c_str()) != 0)
        ConvertTmpToStableFile(tmppath, GetFileNameWhitOutExtention(destinationpath) + "_Copy
" + GetFileExtention(destinationpath));
    else
        return 1;
}
int main()
{

     string source ="C:\\Users\\amirr\\Downloads\\Music\\Test.mp3";
     string tmp = TempPath(GetFileNameWhitOutExtention(source));
     string destination = GetFileName(source);

     ReadDataANDSaveTMP(source, tmp);

     ConvertTmpToStableFile(tmp, destination);

    return 0;
}
```

➢ Temp File (way 1)

```
#define _CRT_SECURE_NO_WARNINGS
#include <iostream>
#include <cstdio>
#include<iostream>
#include<fstream>
#include<cstdio>
#include<algorithm>
#include<iterator>
#include<sstream>
#include<string>
#include <cstdio>

using namespace std;
#define BUFFERLENGTH 1024*1000000
void ReadDataANDSaveTMP(string source, string despath)
{
```

```
    string path = despath;
    ifstream ifs(source, ios::binary);
    long size = ifs.seekg(0, ios::end).tellg();
    ifs.seekg(0, ios::beg);
    istreambuf_iterator<char> inputstream(ifs);
    long Current = 0;
    ofstream ofstmp(path, ios::binary);
    ostreambuf_iterator<char> outfile(ofstmp);
    while (Current < size)
    {
        long readindex = size - Current < BUFFERLENGTH ? size - Current : BUFFERLENGTH;
        copy_n(inputstream, readindex, outfile);
        ifs.seekg(1, ios::cur);
        Current += readindex;
    }
    ofstmp.close();
    ifs.close();
}
void SaveDataOnRealFile(string tmp, string des)
{
    ifstream tempinput(tmp, ios::binary);
    istreambuf_iterator<char> eos;
    istreambuf_iterator<char> TempInput(tempinput);
    ofstream ofsreal(des, ios::binary);
    ostreambuf_iterator<char> RealWrite(ofsreal);
    copy(TempInput, eos, RealWrite);
    ofsreal.close();
    tempinput.close();
}

string GetFileName(string path)
{
    for (int i = path.length(); i >= 0; i--)
        if (path[i] == '\\')
            return path.substr(i+1);
    return path;
}
string GetFileExtention(string path)
{
    string filename = GetFileName(path);
    size_t i = filename.rfind('.', filename.length());
    if (i != string::npos) {
        return(filename.substr(i , filename.length() - i));
    }
}
string GetFileNameWhitOutExtention(string path)
{
```

```
    string filename = GetFileName(path);
    size_t i = filename.rfind('.', filename.length());
    if (i != string::npos) {
        return(filename.substr(0, i));
    }
}
inline string TempPath(string Filename,string Extention = ".tmp")
{
    return Filename + Extention;
}
inline bool IsFileExist(const string& name) {
    ifstream f(name.c_str());
    return f.good();
}
bool RemoveTemp(const string& tmppath )
{
    if (remove(tmppath.c_str()) != 0)
        return false;
    else
        return true;
}
int main()
{

     string source ="C:\\Users\\amirr\\Pictures\\test.png";
     string tmp = TempPath(GetFileNameWhitOutExtention(source));
     string destination = GetFileName(source);

     ReadDataANDSaveTMP(source, tmp);
     SaveDataOnRealFile(tmp, destination);
     RemoveTemp(tmp);

    return 0;
}
```

Summary

So far, we have been using the **iostream** standard library, which provides **cin** and **cout** methods for reading from standard input and writing to standard output respectively.

This tutorial will teach you how to read and write from a file. This requires another standard C++ library called **fstream**, which defines three new data types −

Sr.No	Data Type & Description
1	**ofstream** This data type represents the output file stream and is used to create files and to write information to files.
2	**ifstream** This data type represents the input file stream and is used to read information from files.
3	**fstream** This data type represents the file stream generally, and has the capabilities of both ofstream and ifstream which means it can create files, write information to files, and read information from files.

To perform file processing in C++, header files <iostream> and <fstream> must be included in your C++ source file.

Opening a File

A file must be opened before you can read from it or write to it. Either **ofstream** or **fstream** object may be used to open a file for writing. And ifstream object is used to open a file for reading purpose only.

Following is the standard syntax for open() function, which is a member of fstream, ifstream, and ofstream objects.

```
void open(const char *filename, ios::openmode mode);
```

Here, the first argument specifies the name and location of the file to be opened and the second argument of the **open()** member function defines the mode in which the file should be opened.

Sr.No	Mode Flag & Description
1	**ios::app**

	Append mode. All output to that file to be appended to the end.
2	**ios::ate** Open a file for output and move the read/write control to the end of the file.
3	**ios::in** Open a file for reading.
4	**ios::out** Open a file for writing.
5	**ios::trunc** If the file already exists, its contents will be truncated before opening the file.

You can combine two or more of these values by **OR**ing them together. For example if you want to open a file in write mode and want to truncate it in case that already exists, following will be the syntax −

```
ofstream outfile;
outfile.open("file.dat", ios::out | ios::trunc );
```

Similar way, you can open a file for reading and writing purpose as follows −

```
fstream  afile;
afile.open("file.dat", ios::out | ios::in );
```

Closing a File

When a C++ program terminates it automatically flushes all the streams, release all the allocated memory and close all the opened files. But it is always a good practice that a programmer should close all the opened files before program termination.

Following is the standard syntax for close() function, which is a member of fstream, ifstream, and ofstream objects.

```
void close();
```

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.

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.

Read and Write Example

Following is the C++ program which opens a file in reading and writing mode. After writing information entered by the user to a file named afile.dat, the program reads information from the file and outputs it onto the screen −

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

int main () {
   char data[100];

   // open a file in write mode.
   ofstream outfile;
   outfile.open("afile.dat");

   cout << "Writing to the file" << endl;
   cout << "Enter your name: ";
   cin.getline(data, 100);

   // write inputted data into the file.
   outfile << data << endl;

   cout << "Enter your age: ";
   cin >> data;
   cin.ignore();

   // again write inputted data into the file.
   outfile << data << endl;

   // close the opened file.
   outfile.close();

   // open a file in read mode.
   ifstream infile;
   infile.open("afile.dat");

   cout << "Reading from the file" << endl;
   infile >> data;

   // write the data at the screen.
   cout << data << endl;

   // again read the data from the file and display it.
   infile >> data;
   cout << data << endl;
```

```
   // close the opened file.
   infile.close();

   return 0;
}
```

When the above code is compiled and executed, it produces the following sample input and output −

```
$./a.out
Writing to the file
Enter your name: Zara
Enter your age: 9
Reading from the file
Zara
9
```

Above examples make use of additional functions from cin object, like getline() function to read the line from outside and ignore() function to ignore the extra characters left by previous read statement.

File Position Pointers

Both **istream** and **ostream** provide member functions for repositioning the file-position pointer. These member functions are **seekg** ("seek get") for istream and **seekp** ("seek put") for ostream.

The argument to seekg and seekp normally is a long integer. A second argument can be specified to indicate the seek direction. The seek direction can be **ios::beg** (the default) for positioning relative to the beginning of a stream, **ios::cur** for positioning relative to the current position in a stream or **ios::end** for positioning relative to the end of a stream.

The file-position pointer is an integer value that specifies the location in the file as a number of bytes from the file's starting location. Some examples of positioning the "get" file-position pointer are −

```
// position to the nth byte of fileObject (assumes ios::beg)
fileObject.seekg( n );

// position n bytes forward in fileObject
fileObject.seekg( n, ios::cur );

// position n bytes back from end of fileObject
fileObject.seekg( n, ios::end );

// position at end of fileObject
fileObject.seekg( 0, ios::end );
```