**Details of some useful classes :**  
filebuf :Its purpose is to set the file buffer to read and write. Contains openprot constant used in the open() of the filestream classes. Also contains close() and open() as member functions.  
  
fstreambase:Provides operations common to the file streams. Serves as a base for fstream, ifstream and ofstream classes. Contains open() and close() functions.  
  
ifstream:Provides input operations. Contains open() with default input mode. Inherits the functions get(), getline(), read(), seekg() and tellg() functions from istream.  
  
ofstream:Provides output operations. Contains open() with default output mode. Inherits put(), seekp(), tellp(), and write() functions from ostream.  
  
fstream:Provides support for simultaneous input and output operations. Contains open() with default input mode. Inherits all the functions from istream and ostream classes through iostream.  
  
**The ifstream, ofstream and fstream classes are declared in the file fstream.h  
The istream and ostream classes are also included in the fstream.h file.**

Ios::app Append to end-of file

Ios::ate goto end of file on opening

Ios::binary Open in binary file

Ios::in Open existing file for reading.Open for input operations.

Ios::nocreate open fails if file doesn’t exist

Ios::noreplace open fails if file already exists

Ios::out Open for output operations

Ios::trunc Deletes contents if it exists

ios::ate Set the initial position at the end of the file. If this flag is not set, the initial position is the beginning of the file

ios::app All output operations are performed at the end of the file, appending the content to the current content of the file.

ios::trunc If the file is opened for output operations and it already existed, its previous content is deleted and replaced by the new one.

**ios::beg**          start of the file

**ios::cur**          current position of the pointer

**ios::end**          end of the file

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

**Functions for binary file handling**

get(): read a byte and point to the next byte to read

put(): write a byte and point to the next location for write

read(): block reading

write(): block writing

flush():Save data from the buffer to the output file.

**Checking state flags**

The following member functions exist to check for specific states of a stream (all of them return a bool value):

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

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

eof():Returns true if a file open for reading has reached the end.

good():It is the most generic state flag: it returns false in the same cases in which calling any of the previous functions would return true. Note that good and bad are not exact opposites (good checks more state flags at once).

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| The member function clear() can be used to reset the state flags.  **tellg() and tellp()**  These two member functions with no parameters return a value of the member type streampos, which is a type representing the current *get position* (in the case of tellg) or the *put position* (in the case of tellp).  **seekg() and seekp()**  These functions allow to change the location of the *get* and *put positions*. Both functions are overloaded with two different prototypes. The first form is:  seekg ( position ); seekp ( position );  Using this prototype, the stream pointer is changed to the absolute position position (counting from the beginning of the file). The type for this parameter is streampos, which is the same type as returned by functions tellg and tellp.  The other form for these functions is:  seekg ( offset, direction ); seekp ( offset, direction );  **Buffer**  When the buffer is flushed, all the data contained in it is written to the physical medium (if it is an output stream). This process is called *synchronization* and takes place under any of the following circumstances:  **When the file is closed:** before closing a file, all buffers that have not yet been flushed are synchronized and all pending data is written or read to the physical medium.  **When the buffer is full:** Buffers have a certain size. When the buffer is full it is automatically synchronized.  **Explicitly, with manipulators:** When certain manipulators are used on streams, an explicit synchronization takes place. These manipulators are: [flush](file:///C:\flush) and [endl](file:///C:\endl).  **Explicitly, with member function sync():** Calling the stream's member function sync() causes an immediate synchronization. This function returns an int value equal to -1 if the stream has no associated buffer or in case of failure. Otherwise (if the stream buffer was successfully synchronized) it returns 0.  ***// writing on a text file***  *#include <iostream>*  *#include <fstream>*  *using* *namespace* std;  *int* main () {  ofstream myfile ("example.txt");  *if* (myfile.is\_open())  {  myfile << "This is a line.\n";  myfile << "This is another line.\n";  myfile.close();  }  *else* cout << "Unable to open file";  *return* 0;  }  **// reading a text file**  #include <iostream>  #include <fstream>  #include <string>  using namespace std;  int main () {  string line;  ifstream myfile ("example.txt");  if (myfile.is\_open())  {  while ( getline (myfile,line) )  {  cout << line << '\n';  }  myfile.close();  }  else cout << "Unable to open file";  return 0;  }   |  |  |  | | --- | --- | --- | | ***// obtaining file size***  *#include <iostream>*  *#include <fstream>*  *using* *namespace* std;  *int* main () {  streampos begin,end;  ifstream myfile ("example.bin", ios::binary);  begin = myfile.tellg();  myfile.seekg (0, ios::end);  end = myfile.tellg();  myfile.close();  cout << "size is: " << (end-begin) << " bytes.\n";  *return* 0;  } | size is: 40 bytes. |  |   Notice the type we have used for variables begin and end:  streampos size;  streampos is a specific type used for buffer and file positioning and is the type returned by file.tellg(). Values of this type can safely be subtracted from other values of the same type, and can also be converted to an integer type large enough to contain the size of the file.  These stream positioning functions use two particular types: streampos and streamoff. These types are also defined as member types of the stream class:   |  |  |  | | --- | --- | --- | | **Type** | **Member type** | **Description** | | [streampos](file:///C:\streampos) | [ios::pos\_type](file:///C:\ios#types) | Defined as [fpos<mbstate\_t>](file:///C:\fpos). It can be converted to/from [streamoff](file:///C:\streamoff) and can be added or subtracted values of these types. | | [streamoff](file:///C:\streamoff) | [ios::off\_type](file:///C:\ios#types) | It is an alias of one of the fundamental integral types (such as int or long long). |   Each of the member types above is an alias of its non-member equivalent (they are the exact same type). It does not matter which one is used. The member types are more generic, because they are the same on all stream objects (even on streams using exotic types of characters), but the non-member types are widely used in existing code for historical reasons.  **Binary files**  For binary files, reading and writing data with the extraction and insertion operators (<< and >>) and functions likegetline is not efficient, since we do not need to format any data and data is likely not formatted in lines.  File streams include two member functions specifically designed to read and write binary data sequentially: write andread. The first one (write) is a member function of ostream (inherited by ofstream). And read is a member function of istream (inherited by ifstream). Objects of class fstream have both. Their prototypes are:  write ( memory\_block, size ); read ( memory\_block, size );  Where memory\_block is of type char\* (pointer to char), and represents the address of an array of bytes where the read data elements are stored or from where the data elements to be written are taken. The size parameter is an integer value that specifies the number of characters to be read or written from/to the memory block.   |  |  |  |  | | --- | --- | --- | --- | | 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 | *// reading an entire binary file*  *#include <iostream>*  *#include <fstream>*  *using* *namespace* std;  *int* main () {  streampos size;  *char* \* memblock;  ifstream file ("example.bin", ios::in|ios::binary|ios::ate);  *if* (file.is\_open())  {  size = file.tellg();  memblock = *new* *char* [size];  file.seekg (0, ios::beg);  file.read (memblock, size);  file.close();  cout << "the entire file content is in memory";  *delete*[] memblock;  }  *else* cout << "Unable to open file";  *return* 0;  } | the entire file content is in memory |  |   In this example, the entire file is read and stored in a memory block. Let's examine how this is done:  First, the file is open with the ios::ate flag, which means that the get pointer will be positioned at the end of the file. This way, when we call to member tellg(), we will directly obtain the size of the file.  Once we have obtained the size of the file, we request the allocation of a memory block large enough to hold the entire file:   |  |  |  | | --- | --- | --- | |  | memblock = *new* *char*[size]; |  |   Right after that, we proceed to set the *get position* at the beginning of the file (remember that we opened the file with this pointer at the end), then we read the entire file, and finally close it:   |  |  |  | | --- | --- | --- | | 1 2 3 | file.seekg (0, ios::beg);  file.read (memblock, size);  file.close(); |  | |  |

**Open a file for reading :**

**#include<iostream>**

**#include<fstream>**

**using namespace** std;

**int** main() {

**char** data[100]; *// buffer to store a line read from file*

ifstream ifile; *// declaring an object of class ifstream*

ifile.open("file.txt"); *// open "file.txt" for reading*

cout << "Reading data from a file :-" << endl << endl;

**while** (!ifile.eof()) { *// while the end of file [ eof() ] is not reached*

ifile.getline(data, 100); *// read a line from file*

cout << data << endl; *// print the file to console*

}

ifile.close(); *// close the file*

**return** 0;

}

Run this program in your system to read from a file

In the first program, a file "**file.txt**" is created and some data is written into it. The file is created in the same directory in which the program file is saved.  
In the second program, we read the file line by line using and then print each line on the console. The while loop continues till the end of file is reached. We ensure that using the condition **while( ! ifile.eof( ) )**. Note that we can simply use **while( ifile )** also.  
We can open a file using the constructors of **ifstream** and **ofstream** classes instead of using **open( )** member function. For e.g, **ifstream ifile( "file.text" );**. It is a good practice to check if file is opened successfully before proceeding with further operations.  
  
**File Operations using fstream**  
**Open a file and append data to the end of the file :**

**#include<iostream>**

**#include<fstream>**

**using namespace** std;

**int** main() {

**char** line[100];

fstream file; *// declare an object of fstream class*

file.open("file.txt", ios :: out | ios :: app); *// open file in append mode*

**if** (file.fail()) { *// check if file is opened successfully*

*// file opening failed*

cout << "Error Opening file ... " << endl;

}

**else** {

*// proceed with further operations*

cout << "Enter a line : ";

cin.getline(line, 100);

file << line << endl; *// Append the line to the file*

cout << "Line written into the file" << endl;

}

**return** 0;

}

Run this program in your system to read from a file

In the first program, we take a line as input from user and appends that line to the "**file.txt**".  
In the second program, we read the file character by character using **get( )** function. Similarly, we can write a single character to a file using **put** function.  
  
**Writing class objects to a file**  
There are member functions **read( )** and **write( )** in the **fstream** class which allows reading and writing of class objects. These functions can also be used to write **array** elements into the file. See the program below :

**#include<iostream>**

**#include<fstream>**

**using namespace** std;

*// define a class to store student data*

**class** student {

**int** roll;

**char** name[30];

**float** marks;

**public**:

student() { }

**void** getData(); *// get student data from user*

**void** displayData(); *// display data*

};

**void** student :: getData() {

cout << "\nEnter Roll No. : ";

cin >> roll;

cin.ignore(); *// ignore the newline char inserted when you press enter*

cout << "Enter Name : ";

cin.getline(name, 30);

cout << "Enter Marks : ";

cin >> marks;

}

**void** student :: displayData() {

cout << "\nRoll No. : " << roll << endl;

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

cout << "Marks : " << marks << endl;

}

**int** main() {

student s[3]; *// array of 3 student objects*

fstream file;

**int** i;

file.open("objects.txt", ios :: out); *// open file for writing*

cout << "\nWriting Student information to the file :- " << endl;

**for** (i = 0; i < 3; i++) {

s[i].getData();

*// write the object to a file*

file.write((**char** \*)&s[i], **sizeof**(s[i]));

}

file.close(); *// close the file*

file.open("objects.txt", ios :: in); *// open file for reading*

cout << "\nReading Student information to the file :- " << endl;

**for** (i = 0; i < 3; i++) {

*// read an object from a file*

file.read((**char** \*)&s[i], **sizeof**(s[i]));

s[i].displayData();

}

file.close(); *// close the file*

**return** 0;

}

Run this program in your system to perform read-write operations on a file

Whenever we are taking input from console, we press "enter" key. This leads to a newline character being inserted in the input buffer. **cin** ignores this character to take further inputs but **cin.getline( )** doesn't ignore it. Thus, we need to use the function **cin.ignore( )** otherwise, we won't be able to take further inputs.  
  
**Manipulation of file pointers**  
The read operation from a file involves **get** pointer. It points to a specific location in the file and reading starts from that location. Then, the **get** pointer keeps moving forward which lets us read the entire file. Similarly, we can start writing to a location where **put** pointer is currently pointing. The **get** and **put** are known as file position pointers and these pointers can be manipulated or repositioned to allow random access of the file. The functions which manipulate file pointers are as follows :

|  |  |
| --- | --- |
| **Function** | **Description** |
| seekg( ) | Moves the **get** pointer to a specific location in the file |
| seekp( ) | Moves the **put** pointer to a specific location in the file |
| tellg( ) | Returns the position of **get** pointer |
| tellp( ) | Returns the position of **put** pointer |

The function **seekg( n, ref\_pos )** takes two arguments :   
**n** denotes the number of bytes to move and **ref\_pos** denotes the reference position relative to which the pointer moves. **ref\_pos** can take one of the three constants : **ios :: beg** moves the **get** pointer **n** bytes from the beginning of the file, **ios :: end** moves the **get** pointer **n** bytes from the end of the file and **ios :: cur** moves the **get** pointer **n** bytes from the current position. If we don't specify the second argument, then **ios :: beg** is the default reference position.  
The behaviour of **seekp( n, ref\_pos )** is same as that of **seekg( )**. Following program illustrates random access of file using file pointer manipulation functions :

**#include<iostream>**

**#include<fstream>**

**using namespace** std;

**int** main() {

fstream fp;

**char** buf[100];

**int** pos;

*// open a file in write mode with 'ate' flag*

fp.open("random.txt", ios :: out | ios :: ate);

cout << "\nWriting to a file ... " << endl;

fp << "This is a line" << endl; *// write a line to a file*

fp << "This is a another line" << endl; *// write another file*

pos = fp.tellp();

cout << "Current position of put pointer : " << pos << endl;

*// move the pointer 10 bytes backward from current position*

fp.seekp(-10, ios :: cur);

fp << endl << "Writing at a random location ";

*// move the pointer 7 bytes forward from beginning of the file*

fp.seekp(7, ios :: beg);

fp << " Hello World ";

fp.close(); *// file write complete*

cout << "Writing Complete ... " << endl;

*// open a file in read mode with 'ate' flag*

fp.open("random.txt", ios :: in | ios :: ate);

cout << "\nReading from the file ... " << endl;

fp.seekg(0); *// move the get pointer to the beginning of the file*

*// read all contents till the end of file*

**while** (!fp.eof()) {

fp.getline(buf, 100);

cout << buf << endl;

}

pos = fp.tellg();

cout << "\nCurrent Position of get pointer : " << pos << endl;

**return** 0;

}

Run this program in your system to perform file operations

**Using Constructor**Create a file stream object to manage the stream using the appropriate class. That is, the class ofstream is used to create the output stream and the class ifstream to create the input stream.  
  
Initialize the file object with the desired filename, e.g.:  
  
ofstream outfile(“sample.txt”);  
  
The above statement creates an object outfile of class ofstream that manages the output stream. This statement also opens the file sample.txt and attaches it to the output stream for writing.  
  
Similarly, the statement declared in as an ifstream object and attaches to the file “sample.txt” for reading.  
  
ifstream infile(“sample.txt”);  
  
  
Program: Writing and reading data into file, using constructors  
  
# include  
void main()  
{  
ofstream outfile(“sample.txt”); // create file for output  
char ch = ‘a’;  
int i = 12;  
float f = 4356.15;  
char arr[ ] = “hello”;  
outfile << ch << endl <*<< endl << f << endl << arr; //send the data to file  
outfile.close();  
  
ifstream infile(“sample.txt”);  
  
  
infile >> ch >> i >> f >> arr; // read data from file  
  
cout << ch << i << f << arr; // send data to screen  
}  
  
To write data into the file, character by character.  
  
#include  
#include  
void main()  
{  
char str[]=“C++ is superset of C. It is an object-oriented /  
programming language.”;  
  
ofstream outfile(“sample2.txt”); // Open the file in write mode  
  
for(int i = 0; i < strlen(str); i++)  
outfile.put(str[i]); // write data into the file, character by character.  
}****Writing and reading Objects of a class :*** *So far we have done I/O of basic data types. Since the class objects are the central elements of C++ programming, it is quite natural that the language supports features for writing and reading from the disk files objects directly.  
  
The binary input and output functions read() and write() are designed to do exactly this job.  
  
The write() function is used to write the object of a class into the specified file and read() function is used to read the object of the class from the file.  
  
Both these functions take two arguments:  
1. address of object to be written.  
2. size of the object.  
  
The address of the object must be cast to the type pointer to char.  
  
One important point to remember is that only data members are written to the disk file and the member functions are not.  
  
  
Writing an object into the file  
  
#include  
class Person  
{  
private:  
char name[40];  
int age;  
public:  
void getData()  
{  
cout << “\n Enter name:”; cin >> name;  
cout << “\n Enter age:”; cin >> age;  
}  
} ; // End of the class definition  
  
void main()  
{  
Person per ; // Define an object of Person class  
  
per.getData(); // Enter the values to the data members of the class.  
  
ofstream outfile(“Person.txt”); // Open the file in output mode  
  
outfile.write((char\*)&per, sizeof(per)); // Write the object into the file  
}  
  
  
fstream object can be used for both input & output.  
  
In the open() function we include several mode bits to specify certain aspects of the file object.  
  
app -> To preserve whatever was in the file before. Whatever we write to the file will be appended to the existing contents.  
  
We use in and out because we want to perform both input and output on the file.  
  
eof() is a member function of ios class. It returns a nonzero value if EOF is encountered and a zero otherwise.*

***Parameters of open() function******File pointers and their manipulations:*** *Each file has two associated pointers known as the file pointers.  
  
One of them is called the input pointer or get pointer.  
  
Other is called the output pointer or put pointer.  
  
We can use these pointers to move through the files while reading or writing.  
  
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.****Functions for manipulation of file pointers*** *seekg() Moves get pointer (input) to a specified location.  
seekp() Moves put pointer (output) to a specified location.  
tellg() Gives the current position of the get pointer.  
tellp() Gives the current position of the put pointer.  
  
infile.seekg(10);  
Moves the file pointer to the byte number 10.  
  
The bytes in a file are numbered beginning from zero.  
  
Thus, the pointer will be pointing to the 11th byte in the file.****Specifying the offset :*** *The seek functions seekg() and seekp() can also be used with two arguments as follows:  
seekg(offset, refposition);  
seekp(offset, refposition);  
The parameter offset represents the number of bytes the file pointer to be moved from the location specified by the parameter refposition.  
The refposition takes one of the following these constant defined in the ios class.  
ios::beg start of the file  
ios::cur current position of the pointer  
ios::end end of the file.****This program counts the number of objects already written into the file “Person.txt”. Then is reads the second object and displays the values of its data members.*** *class person  
{  
private:  
char name[40];  
int age;  
public:  
void showData()  
{  
cout << “\n Name = “ << name;  
cout << “\n Age = “ << age;  
}  
};  
  
void main()  
{  
person pers; // create person object  
ifstream infile; // create input file  
infile.open(“Person.txt”); // open the file  
infile.seekg(0, ios::end); // go to end from 0 byte  
int endposition = infile.tellg(); // find where we are  
int n = endposition/sizeof(person); // number of persons  
cout << “\n There are “ << n << “ persons in file: “;  
cout << “\n Enter person number: “;  
cin >> n;  
int position = (n-1) \* sizeof(person); // number times size  
infile.seekg(position);  
infile.read( (char\*)&pers, sizeof(pers) );  
pers.showData(); // display the person  
}*

File handling with C++ streams

I/O in C++ example from a Well House Consultants training course  
More on I/O in C++ [[link]](file:///C:\resources\C235.html)

Source code: file01.cpp Module: C235

#include <iostream.h>  
#include <fstream.h>  
  
/\*  
\* Copy one file onto the end of another, adding line numbers  
\*/  
  
int main () {  
        char myline[256];  
        int lc = 0;  
  
        ofstream outfile("demo.txt",ios::app);  
  
        ifstream infile("stdcodes.xyz");  
        if (! infile) {  
                cerr << "Failed to open input file\n";  
                exit(1);  
                }  
  
        while (1) {  
                infile.getline(myline,256);  
                if (infile.eof()) break;  
                lc++;  
                outfile << lc << ": " << myline << "\n";  
                }  
        infile.close();  
        outfile.close();  
  
        cout << "Output " << lc << " records" << endl;  
  
}  
  
/\* Sample Output  
  
munchkin:c235 grahamellis$ ./file01  
Output 110 records  
munchkin:c235 grahamellis$  
  
\*/