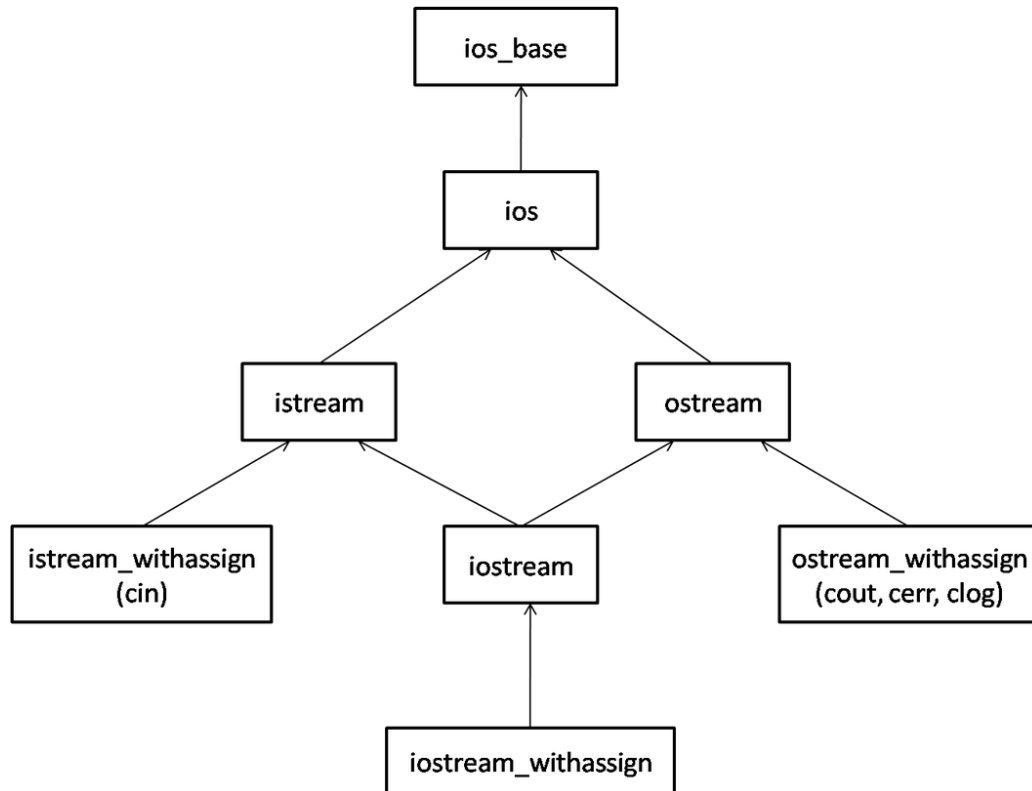# What are stream classes in c++

All these classes are declared in **iostream.h** header file. So, we have to include this header file in order to use the stream classes. As shown in the above figure, **ios** is the base class for all the stream classes from which **istream** and **ostream** classes are derived. The **iostream** class is derived from both **istream** and **ostream** classes.

Following table lists out some of the stream classes and their contents:

Class	Content / Functions
ios	• It is an input and output stream class. • It is used to implement a buffer. It is a pointer to the buffer, streambuf. • ios matains information about the state of the buffer.
istream	• istream provides formatted input. • It is used to handle formatted as well as unformatted conversion of character from a streambuf. • istream declares functions such as peek(), tellg(), seekg(), getline(), read() etc. • istream class overloads the ">>" operator.
ostream	• It is used for general purpose output. • ostream declares functions such as tellp(), put(), write(), seekp(), etc. • ostream overloads the "<<" operator.
iostream	• It is used to handle both input and output streams.
istream_withassign	• It is derived from istream. • It is used for cin input.
iostream_withassign	• It is a bidirectional stream.

Input/output with files

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

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

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

```
1  // basic file operations
2  #include <iostream>
3  #include <fstream>
4  using namespace std;
5
6  int main () {                              [file example.txt]
7    ofstream myfile;                         Writing this to a file.
8    myfile.open ("example.txt");
9    myfile << "Writing this to a file.\n";
10   myfile.close();
11   return 0;
12 }
```

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

But let's go step by step:

Open a file

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

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

```
open (filename, mode);
```

Where `filename` is a string representing the name of the file to be opened, and `mode` is an optional parameter with a combination of the following flags:

`ios::in`	Open for input operations.
`ios::out`	Open for output operations.
`ios::binary`	Open in binary mode.
`ios::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.

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

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

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

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

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

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

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

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

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

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

To check if a file stream was successful opening a file, you can do it by calling to member `is_open`. This member function returns a `bool` value of `true` in the case that indeed the stream object is associated with an open file, or `false` otherwise:

```
if (myfile.is_open()) { /* ok, proceed with output */ }
```

Closing a file

When we are finished with our input and output operations on a file we shall close it so that the operating system is notified and its resources become available again. For that, we call the stream's member function `close`. This member function takes flushes the associated buffers and closes the file:

```
myfile.close();
```

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

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

Text files

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

Writing operations on text files are performed in the same way we operated with `cout`:

```
1  // writing on a text file
2  #include <iostream>
3  #include <fstream>
4  using namespace std;
5
6  int main () {
7    ofstream myfile ("example.txt");          [file example.txt]
8    if (myfile.is_open())                     This is a line.
9    {                                         This is another line.
10     myfile << "This is a line.\n";
11     myfile << "This is another line.\n";
12     myfile.close();
13   }
14   else cout << "Unable to open file";
15   return 0;
16 }
```

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

```
1  // reading a text file
2  #include <iostream>
3  #include <fstream>
4  #include <string>
5  using namespace std;
6
7  int main () {
8    string line;
9    ifstream myfile ("example.txt");
10   if (myfile.is_open())
11   {                                    This is a line.
12     while ( getline (myfile,line) )    This is another line.
13     {
14       cout << line << '\n';
15     }
16     myfile.close();
17   }
18
19   else cout << "Unable to open file";
20
21   return 0;
22 }
```

This last example reads a text file and prints out its content on the screen. We have created a while loop that reads the file line by line, using getline. The value returned by getline is a reference to the stream object itself, which when evaluated as a boolean expression (as in this while-loop) is `true` if the stream is ready for more operations, and `false` if either the end of the file has been reached or if some other error occurred.

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.

get and put stream positioning

All i/o streams objects keep internally -at least- one internal position:

`ifstream`, like `istream`, keeps an internal *get position* with the location of the element to be read in the next input operation.

`ofstream`, like `ostream`, keeps an internal *put position* with the location where the next element has to be written.

Finally, `fstream`, keeps both, the *get* and the *put position*, like `iostream`.

These internal stream positions point to the locations within the stream where the next reading or writing operation is performed. These positions can be observed and modified using the following member functions:

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 );
```

Using this prototype, the *get* or *put position* is set to an offset value relative to some specific point determined by the parameter `direction`. `offset` is of type `streamoff`. And `direction` is of type `seekdir`, which is an *enumerated type* that determines the point from where offset is counted from, and that can take any of the following values:

`ios::beg` offset counted from the beginning of the stream

`ios::cur` offset counted from the current position

`ios::end` offset counted from the end of the stream

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

```
1  // obtaining file size
2  #include <iostream>
3  #include <fstream>
4  using namespace std;
5
6  int main () {
7    streampos begin,end;
8    ifstream myfile ("example.bin", ios::binary);      size is: 40 bytes.
9    begin = myfile.tellg();
10   myfile.seekg (0, ios::end);
11   end = myfile.tellg();
12   myfile.close();
13   cout << "size is: " << (end-begin) << " bytes.\n";
14   return 0;
15 }
```

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	ios::pos_type	Defined as fpos<mbstate_t>. It can be converted to/from streamoff and can be added or subtracted values of these types.

streamoff	ios::off_type	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 like `getline` 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` and `read`. 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  // reading an entire binary file
2  #include <iostream>
3  #include <fstream>
4  using namespace std;
5
6  int main () {
7    streampos size;
8    char * memblock;
9
10   ifstream file ("example.bin",
11 ios::in|ios::binary|ios::ate);                    the entire file content
12   if (file.is_open())                             is in memory
13   {
14     size = file.tellg();
15     memblock = new char [size];
16     file.seekg (0, ios::beg);
17     file.read (memblock, size);
18     file.close();
19
20     cout << "the entire file content is in
21 memory";
22
```

```
23     delete[] memblock;
24   }
25   else cout << "Unable to open file";
     return 0;
   }
```

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 file.seekg (0, ios::beg);
2 file.read (memblock, size);
3 file.close();
```

At this point we could operate with the data obtained from the file. But our program simply announces that the content of the file is in memory and then finishes.

Buffers and Synchronization

When we operate with file streams, these are associated to an internal buffer object of type `streambuf`. This buffer object may represent a memory block that acts as an intermediary between the stream and the physical file. For example, with an `ofstream`, each time the member function `put` (which writes a single character) is called, the character may be inserted in this intermediate buffer instead of being written directly to the physical file with which the stream is associated.

The operating system may also define other layers of buffering for reading and writing to files.

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` and `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`.