
15.9. File Access

Streams can be used to access files. This section discusses the corresponding features provided.

15.9.1. File Stream Classes

The C++ standard library provides four class templates for which the following standard specializations are predefined:

1. The class template `basic_ifstream<>` with the specializations `ifstream` and `wifstream` is for read access to files (“input file stream”).
2. The class template `basic_ofstream<>` with the specializations `ofstream` and `wofstream` is for write access to files (“output file stream”).
3. The class template `basic_fstream<>` with the specializations `fstream` and `wfstream` is for access to files that should be both read and written.
4. The class template `basic_filebuf<>` with the specializations `filebuf` and `wfilebuf` is used by the other file stream classes to perform the actual reading and writing of characters.

The classes are related to the stream base classes, as depicted in Figure 15.2, and are declared in the header file `<fstream>` as follows:

```
namespace std {
    template <typename charT,
              typename traits = char_traits<charT> >
             class basic_ifstream;
    typedef basic_ifstream<char>    ifstream;
    typedef basic_ifstream<wchar_t> wifstream;

    template <typename charT,
              typename traits = char_traits<charT> >
             class basic_ofstream;
    typedef basic_ofstream<char>    ofstream;
    typedef basic_ofstream<wchar_t> wofstream;

    template <typename charT,
              typename traits = char_traits<charT> >
             class basic_fstream;
    typedef basic_fstream<char>     fstream;
    typedef basic_fstream<wchar_t>  wfstream;

     template <typename charT,
               typename traits = char_traits<charT> >
              class basic_filebuf;
     typedef basic_filebuf<char>    filebuf;
     typedef basic_filebuf<wchar_t> wfilebuf;
}
```

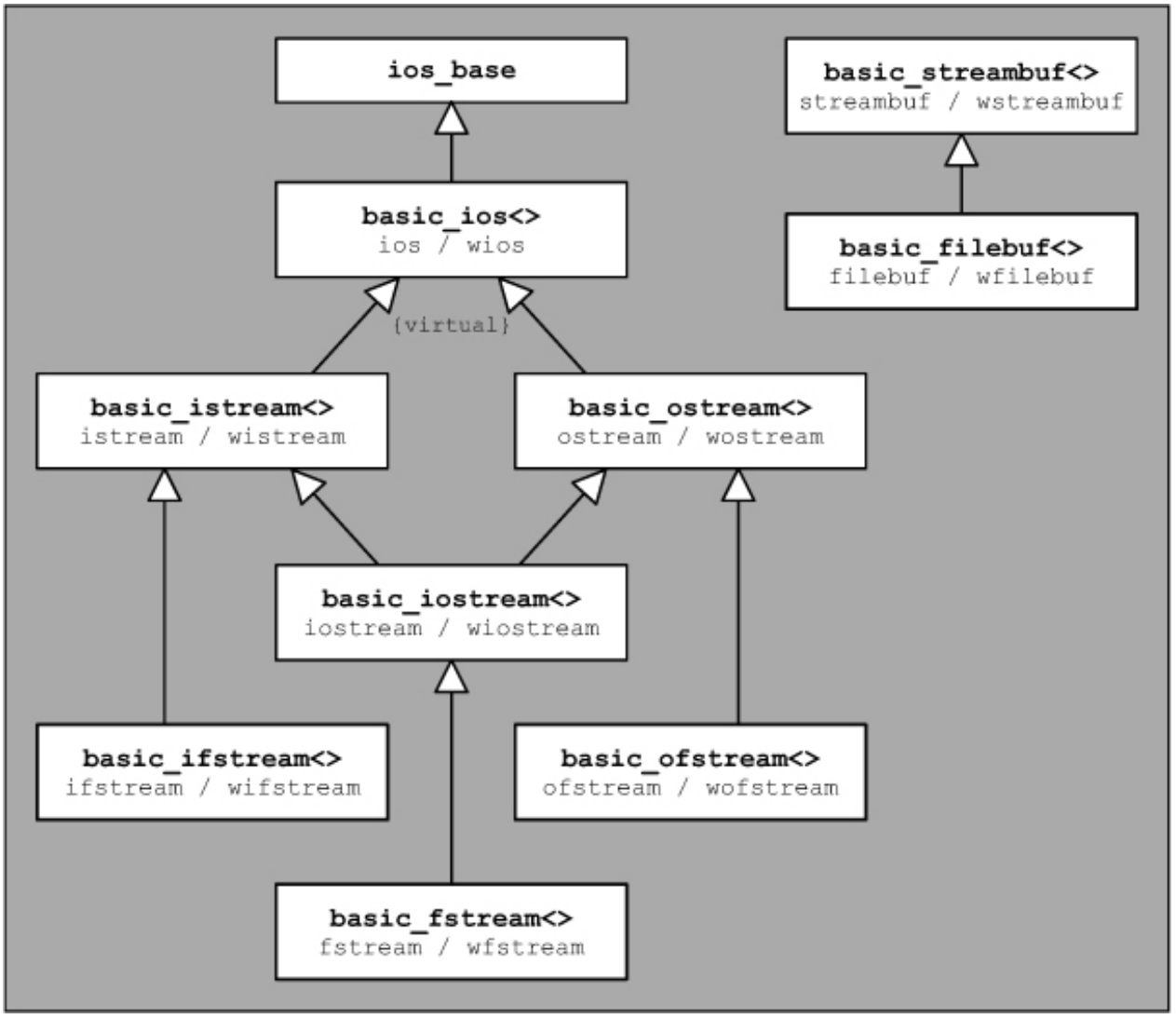

Figure 15.2. Class Hierarchy of the File Stream Classes

Compared with the mechanism of C, a major advantage of the file stream classes for file access is the automatic management of files. The files are automatically opened at construction time and closed at destruction time. This is possible, of course, through appropriate definitions of corresponding constructors and destructors.

For streams that are both read and written, it is important to note that it is *not* possible to switch arbitrarily between reading and writing.[12] Once you start to read or write a file you have to perform a seek operation, potentially to the current position, to switch from reading to writing or vice versa. The only exception to this rule is if you have read until end-of-file. In this case, you can continue writing characters immediately. Violating this rule can lead to all kinds of strange effects.

[12] This restriction is inherited from C. However, it is likely that implementations of the standard C++ library make use of this restriction.

If a file stream object is constructed with a string or C-string as an argument, opening the file for reading and/or writing is attempted automatically. Whether this attempt was successful is reflected in the stream's state. Thus, the state should be examined after construction.

The following program first opens the file `charset.out`, writes the current character set — all characters for the values between `32` and `255` — into this file, and outputs its contents:

Click here to view code image

```
// io/fstream1.cpp

#include <string>          // for strings
#include <iostream>        // for I/O
#include <fstream>         // for file I/O
#include <iomanip>         // for setw()
#include <cstdlib>         // for exit()
using namespace std;

// forward declarations
void writeCharsetToFile (const string& filename);
void outputFile (const string& filename);

int main ()
{
    writeCharsetToFile("charset.out");
```

```
    outputFile("charset.out");
}

void writeCharsetToFile (const string& filename)
{
    // open output file
    ofstream file(filename);

    // file opened?
    if (! file) {
        // NO, abort program
        cerr << "can't open output file \"" << filename << "\""
             << endl;
        exit(EXIT_FAILURE);
    }

    // write character set
    for (int i=32; i<256; ++i) {
        file << "value: " << setw(3) << i << "   "
             << "char:  " << static_cast<char>(i) << endl;
    }

}   // closes file automatically

void outputFile (const string& filename)
{
    // open input file
    ifstream file(filename);

    // file opened?
    if (! file) {
        // NO, abort program
        cerr << "can't open input file \"" << filename << "\""
             << endl;
        exit(EXIT_FAILURE);
    }

    // copy file contents to cout
    char c;
    while (file.get(c)) {
        cout.put(c);
    }

}   // closes file automatically
```

In `writeCharsetToFile()`, the constructor of the class `ofstream` takes care of opening the file named by the given filename:

```
std::ofstream file(filename);   // open file for writing
```

Unfortunately, before C++11, the file stream classes did not provide a constructor taking a `string` as argument. So, before C++11, you had to write:

Click here to view code image

```
std::ofstream file(filename.c_str());   // open file for writing before C++11
```

After this declaration, it is determined whether the stream is in a good state:

```
if (! file) {
    ...
}
```

If opening the stream was not successful, this test will fail. After this check, a loop prints the values `32` to `255` together with the corresponding characters.

In the function `outputFile()`, the constructor of the class `ifstream` opens the file for reading. Then the contents of the file are read and output characterwise.

At the end of both functions, the file opened locally is closed automatically when the corresponding stream goes out of scope. The destructors of the classes `ifstream` and `ofstream` take care of closing the file if it is still open at destruction time.

Instead of copying the file contents character by character, you could also output the whole contents in one statement by passing a pointer to the stream buffer of the file as an argument to operator `<<`:

```
// copy file contents to cout
std::cout << file.rdbuf();
```

See Section 15.14.3, page 846, for details.

15.9.2. Rvalue and Move Semantics for File Streams

Since C++11, file streams provide rvalue and move semantics. In fact, ostreams provide an output operator, and istreams provide an input iterator that accepts an rvalue reference for the stream. The effect is that you can now use temporarily created stream objects, and they behave as expected. For example, you can write to a temporarily created file stream:[13]

[13] Thanks to Daniel Krügler for providing this example.

Click here to view code image

```
// io/fstream2.cpp

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

int main()
{
    // write string to a temporarily created file stream (since C++11)
    std::string s("hello");
    std::ofstream("fstream2.tmp") << s << std::endl;

    // write C-string to a temporarily created file stream
    // - NOTE: wrote a pointer value before C++11
    std::ofstream("fstream2.tmp", std::ios::app) << "world" <<
std::endl;
}
```

Since C++11, this writes `"hello"` and `"world"` to the file `"fstream2.tmp"`. Before C++11, instead of the first output statement you had to write the following:

```
std::string s("hello");
std::ofstream os("fstream2.tmp");
os << s << std::endl;
```

Note that before C++11, the second output statement compiled but did something very unexpected: It wrote a *pointer value* to `"fstream2.tmp"`. The reason for the old behavior was that the following member function was called (see Section 15.3.3, page 756):

```
ostream& ostream::operator<< (const void* ptr);
```

In addition, file streams now have move and swap semantics providing a move constructor, a move assignment operator, and `swap()`. So you can pass a file stream as argument or return a file stream from a function. For example, if a file should be used longer than the scope in which it was created, you can return it as follows since C++11 (see Section 3.1.5, page 22, for details about returning values with move semantics):

Click here to view code image

```
std::ofstream openFile (const std::string& filename)
{
    std::ofstream file(filename);
    ...
    return file;
}

std::ofstream file;
file = openFile("xyz.tmp");            // use returned file stream (since C++11)
file << "hello, world" << std::endl;
```

Before C++11, you had to — and as an alternative you still can — allocate the file object on the heap and delete it later when it is no longer needed:

```
std::ofstream* filePtr = new std::ofstream("xyz.tmp");
...
delete filePtr;
```

For such a case, some smart pointer classes (see Section 5.2, page 76) should be used.

15.9.3. File Flags

A set of flags is defined in the class `ios_base` for precise control over the processing mode of a file (Table 15.33). These flags are of type `openmode`, which is a bitmask type similar to `fmtflags`.

Table 15.33. Flags for Opening Files

Flag	Meaning
`in`	Opens for reading (default for `ifstream`)
`out`	Opens for writing (default for `ofstream`)
`app`	Always appends at the end when writing
`ate`	Positions at the end of the file after opening ("at end")
`trunc`	Removes the former file contents
`binary`	Does not replace special characters

The flag `binary` configures the stream to suppress conversion of special characters or character sequences, such as end-of-line or end-of-file. In operating systems, such as Windows or OS/2, a line end in text files is represented by two characters (CR and LF). In normal text mode (`binary` is not set), newline characters are replaced by the two-character sequence, and vice versa, when reading or writing to avoid special processing. In binary mode (`binary` is set), none of these conversions take place.

The flag `binary` should always be used if the contents of a file do not consist of a character sequence but are processed as binary data. An example is the copying of files by reading the file to be copied character by character and writing those characters without modifying them. If the file is processed as text, the flag should not be set, because special handling of newlines would be required. For example, a newline would still consist of two characters.

Some implementations provide additional flags, such as `nocreate` (the file must exist when it is opened) and `noreplace` (the file must not exist). However, these flags are not standard and thus are not portable.

The flags can be combined by using operator `|` . The resulting `openmode` can be passed as an optional second argument to the constructor. For example, the following statement opens a file for appending text at the end:

```
std::ofstream file("xyz.out", std::ios::out|std::ios::app);
```

Table 15.34 correlates the various combinations of flags with the strings used in the interface of C's function for opening files: `fopen()` . The combinations with the `binary` and the `ate` flags set are not listed. A set `binary` corresponds to strings with `b` appended, and a set `ate` corresponds to a seek to the end of the file immediately after opening. Other combinations not listed in the table, such as `trunc|app` , are not allowed. Note that before C++11, `app` , `in|app` , and `in|out|app` were not specified.

Table 15.34. Meaning of Open Modes in C++

ios_base Flags	Meaning	C Mode
`in`	Reads (file must exist)	`"r"`
`out`	Empties and writes (creates if necessary)	`"w"`
`out|trunc`	Empties and writes (creates if necessary)	`"w"`
`out|app`	Appends (creates if necessary)	`"a"`
`app`	Appends (creates if necessary)	`"a"`
`in|out`	Reads and writes; initial position is the start (file must exist)	`"r+"`
`in|out|trunc`	Empties, reads, and writes (creates if necessary)	`"w+"`
`in|app`	Updates at end (creates if necessary)	`"a+"`
`in|out|app`	Updates at end (creates if necessary)	`"a+"`

Whether a file is opened for reading and/or for writing is independent of the corresponding stream object's class. The class determines only the default open mode if no second argument is used. This means that files used only by the class `ifstream` or the class `ofstream` can be opened for reading *and* writing. The open mode is passed to the corresponding stream buffer class, which opens the file. However, the operations possible for the object are determined by the stream's class.

The file owned by a file stream can also be opened or closed explicitly. For this, three member functions are defined (Table 15.35). These functions are useful mainly if a file stream is created without being initialized.

Table 15.35. Member Functions to Open and Close Files

Member Function	Meaning
`open(`*name*`)`	Opens a file for the stream, using the default mode
`open(`*name*`,`*flags*`)`	Opens a file for the stream, using *flags* as the mode
`close()`	Closes the streams file
`is_open()`	Returns whether the file is opened

To demonstrate their use, the following example opens all files with names that are given as arguments to the program and writes their

contents (this corresponds to the UNIX program `cat`).

```
// io/cat1.cpp

// header files for file I/O
#include <fstream>
#include <iostream>
using namespace std;

// for all filenames passed as command-line arguments
// - open, print contents, and close file
int main (int argc, char* argv[])
{
    ifstream file;

    // for all command-line arguments
    for (int i=1; i<argc; ++i) {

        // open file
        file.open(argv[i]);

        // write file contents to cout
        char c;
        while (file.get(c)) {
            cout.put(c);
        }

        // clear eofbit and failbit set due to end-of-file
        file.clear();

        // close file
        file.close();
    }
}
```

Note that after the processing of a file, `clear()` must be called to clear the state flags that are set at end-of-file. This is required because the stream object is used for multiple files. Note that `open()` *never* clears any state flags. Thus, if a stream was not in a good state after closing and reopening it, you still have to call `clear()` to get to a good state. This is also the case if you open a different file.

Instead of processing character by character, you could also print the entire contents of the file in one statement by passing a pointer to the stream buffer of the file as an argument to operator `<<`:

```
// write file contents to cout
std::cout << file.rdbuf();
```

See Section 15.14.3, page 846, for details.

15.9.4. Random Access

Table 15.36 lists the member functions defined for positioning within C++ streams. These functions distinguish between read and write position (`g` stands for *get* and `p` stands for *put*). Read-position functions are defined in `basic_istream<>`, and write-position functions are defined in `basic_ostream<>`. However, not all stream classes support positioning. For example, positioning the streams `cin`, `cout`, and `cerr` is not defined. The positioning of files is defined in the base classes because, usually, references to objects of type `istream` and `ostream` are passed around.

Table 15.36. Member Functions for Stream Positions

Class	Member Function	Meaning
`basic_istream<>`	`tellg()`	Returns the read position
	`seekg(`*pos*`)`	Sets the read position as an absolute value
	`seekg(`*offset*`,`*rpos*`)`	Sets the read position as a relative value
`basic_ostream<>`	`tellp()`	Returns the write position
	`seekp(`*pos*`)`	Sets the write position as an absolute value
	`seekp(`*offset*`,`*rpos*`)`	Sets the write position as a relative value

The functions `seekg()` and `seekp()` can be called with absolute or relative positions. To handle absolute positions, you must use `tellg()` and `tellp()`, which return an absolute position as a value of type `pos_type`. This value is *not* an integral value or simply the position of the character as an index, because the logical position and the real position can differ. For example, in Windows text files, newline characters are represented by two characters in the file, even though it is logically only one character. Things are

even worse if the file uses some multibyte representation for the characters.

The exact definition of `pos_type` is a bit complicated: The C++ standard library defines a global class template `fpos<>` for file positions. Class `fpos<>` is used to define types `streampos` for `char` and `wstreampos` for `wchar_t` streams. These types are used to define the `pos_type` of the corresponding character traits (see Section 16.1.4, page 855), and this `pos_type` member of the traits is used to define `pos_type` of the corresponding stream classes. Thus, you could also use `streampos` as the type for the stream positions. However, using `long` or `unsigned long` is wrong because `streampos` is *not* an integral type (anymore).[14] For example:

[14] Formerly, `streampos` was used for stream positions, and it was simply defined as `unsigned long`.

```
// save current file position
std::ios::pos_type pos = file.tellg();
...
// seek to file position saved in pos
file.seekg(pos);
```

Instead of

```
std::ios::pos_type pos;
```

you could also write:

```
std::streampos pos;
```

For relative values, the offset can be relative to three positions, for which corresponding constants are defined (Table 15.37). The constants are defined in class `ios_base` and are of type `seekdir`.

Table 15.37. Constants for Relative Positions

Constant	Meaning
beg	Position is relative to the beginning ("beginning")
cur	Position is relative to the current position ("current")
end	Position is relative to the end ("end")

The type for the offset is `off_type`, which is an indirect definition of `streamoff`. Similar to `pos_type`, `streamoff` is used to define `off_type` of the traits (see Section 16.1.4, page 855) and the stream classes. However, `streamoff` is a signed integral type, so you can use integral values as stream offsets. For example:

```
// seek to the beginning of the file
file.seekg (0, std::ios::beg);
...
// seek 20 characters forward
file.seekg (20, std::ios::cur);
...
// seek 10 characters before the end
file.seekg (-10, std::ios::end);
```

In all cases, care must be taken to position only within a file. If a position ends up before the beginning of a file or beyond the end, the behavior is undefined.

The following example demonstrates the use of `seekg()`. It uses a function that writes the contents of a file twice:

```
// io/cat2.cpp

// header files for file I/O
#include <iostream>
#include <fstream>

void printFileTwice (const char* filename)
{
    // open file
    std::ifstream file(filename);

    // print contents the first time
    std::cout << file.rdbuf();

    // seek to the beginning
    file.seekg(0);
```

```
    // print contents the second time
    std::cout << file.rdbuf();
}

int main (int argc, char* argv[])
{
    // print all files passed as a command-line argument twice
    for (int i=1; i<argc; ++i) {
        printFileTwice(argv[i]);
    }
}
```

Note that `file.rdbuf()` is used to print the contents of `file` (see Section 15.14.3, page 846). Thus, you operate directly on the stream buffer, which can't manipulate the state of the stream. If you print the contents of `file` by using the stream interface functions, such as `getline()` (see Section 15.5.1, page 768), you'd have to `clear()` the state of `file` before it could be manipulated in any way (including changes of the read position), because these functions set `ios::eofbit` and `ios::failbit` when end-of-file is reached.

Different functions are provided for the manipulation of the read and the write positions. However, for the standard streams, the same position is maintained for reading and writing in the same stream buffer. This is important if multiple streams use the same stream buffer (see Section 15.12.2, page 820, for details).

15.9.5. Using File Descriptors

Some implementations provide the possibility of attaching a stream to an already opened I/O channel. To do this, you initialize the file stream with a *file descriptor*.

File descriptors are integers that identify an open I/O channel. In UNIX-like systems, file descriptors are used in the low-level interface to the I/O functions of the operating system. The following file descriptors are predefined:

- `0` for the standard input channel
- `1` for the standard output channel
- `2` for the standard error channel

These channels may be connected to files, the console, other processes, or some other I/O facility.

Unfortunately, the C++ standard library does not provide the possibility of attaching a stream to an I/O channel using file descriptors. The reason is that the language is supposed to be independent of any operating system. In practice, though, the possibility probably still exists. The only drawback is that using it is not portable to all systems. What is missing at this point is a corresponding specification in a standard of operating system interfaces, such as POSIX or X/OPEN. However, such a standard is not yet planned, but at least `posix` is a reserved namespace since C++11.

Nevertheless, it is possible to initialize a stream by a file descriptor. See Section 15.13.3, page 835, for a description and implementation of a possible solution.