15.2. Fundamental Stream Classes and Objects

15.2.1. Classes and Class Hierarchy

The stream classes of the IOStream library form a hierarchy, as shown in Figure 15.1. For class templates, the upper row shows the name of the class template, and the lower row presents the names of the instantiations for the character types `char` and `wchar_t`.

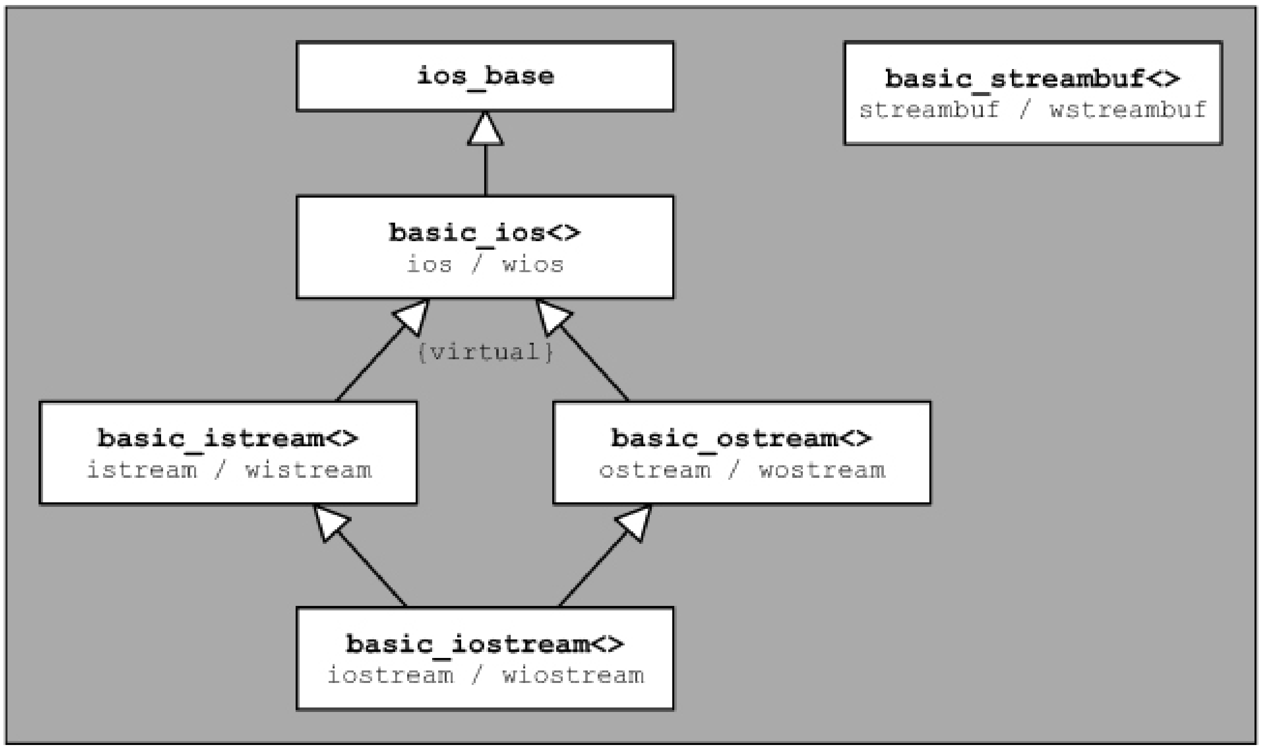

Figure 15.1. Class Hierarchy of the Fundamental Stream Classes

The classes in this class hierarchy play the following roles:

- The base class `ios_base` defines the properties of all stream classes independent of the character type and the corresponding character traits. Most of this class consists of components and functions for state and format flags.
- The class template `basic_ios<>` is derived from `ios_base` and defines the common properties of all stream classes that depend on the character types and the corresponding character traits. These properties include the definition of the buffer used by the stream. The buffer is an object of a class derived from the template class `basic_streambuf<>` with the corresponding template instantiation. It performs the actual reading and/or writing.
- The class templates `basic_istream<>` and `basic_ostream<>` derive virtually from `basic_ios<>` and define objects that can be used for reading or writing, respectively. Like `basic_ios<>`, these classes are templates that are parametrized with a character type and its traits. When internationalization does not matter, the corresponding instantiations for the character type `char` — `istream` and `ostream` — are used.
- The class template `basic_iostream<>` derives from both `basic_istream<>` and `basic_ostream<>`. This class template defines objects that can be used for both reading and writing.
- The class template `basic_streambuf<>` is the heart of the IOStream library. This class defines the interface to all representations that can be written to or read from by streams and is used by the other stream classes to perform the reading and writing of characters. For access to some external representation, classes are derived from `basic_streambuf<>`. See the following subsection for details.

Purpose of the Stream Buffer Classes

The IOStream library is designed with a rigid separation of responsibilities. The classes derived from `basic_ios` handle only *formatting* of the data.[2] The reading and writing of characters is performed by the stream buffers maintained by the `basic_ios` subobjects. The stream buffers supply character buffers for reading and writing. In addition, an abstraction from the external representation, such as files or strings, is formed by the stream buffers.

[2] In fact, they don't even do the formatting! The formatting is delegated to corresponding facets in the locale library. See Section 16.2.2, page 864, and Section 16.4, page 869, for details on facets.

Thus, stream buffers play an important role when performing I/O with new external representations (such as sockets or graphical user interface components), redirecting streams, or combining streams to form pipelines (for example, to compress output before writing to another stream). Also, the stream buffer synchronizes the I/O when doing simultaneous I/O on the same external representation. The details about these techniques are explained in Section 15.12, page 819.

By using stream buffers, it is quite easy to define access to a new "external representation," such as a new storage device. All that has to be done is to derive a new stream buffer class from `basic_streambuf<>` or an appropriate specialization and to define functions for reading and/or writing characters for this new external representation. All options for formatted I/O are available automatically if a stream object is initialized to use an object of the new stream buffer class. See Section 15.13, page 826, for details of stream buffers and Section 15.13.3, page 832, for examples of how to define new stream buffers for access to special storage devices.

Detailed Class Definitions

Like all class templates in the IOStream library, the class template `basic_ios<>` is parametrized by two arguments and is defined as follows:

```
namespace std {
    template <typename charT,
              typename traits = char_traits<charT> >
             class  basic_ios;
}
```

The template arguments are the character type used by the stream classes and a class describing the traits of the character type that are used by the stream classes.

Examples of traits defined in the traits class are the value used to represent end-of-file[3] and the instructions for how to copy or move a sequence of characters. Normally, the traits for a character type are coupled with the character type, thereby making it reasonable to define a class template that is specialized for specific character types. Hence, the traits class defaults to `char_traits<charT>` if `charT` is the character type argument. The C++ standard library provides specializations of the class `char_traits` for the character types `char` , `char16_t` , `char32_t` , and `wchar_t` .[4] For more details about character traits, see Section 16.1.4, page 853.

[3] I use theterm *end-of-file* for the "end of input data." This corresponds with the constant `EOF` in C.

[4] Character traits for `char16_t` and `char32_t` are provided since C++11.

There are two instantiations of the class `basic_ios<>` for the two character types used most often:

```
namespace std {
    typedef basic_ios<char>    ios;
    typedef basic_ios<wchar_t> wios;
}
```

The type `ios` corresponds to the base class of the "old-fashioned" IOStream library from AT&T and can be used for compatibility in older C++ programs.

The stream buffer class used by `basic_ios` is defined similarly:

```
namespace std {
    template <typename charT,
              typename traits = char_traits<charT> >
    class basic_streambuf;
    typedef basic_streambuf<char>    streambuf;
    typedef basic_streambuf<wchar_t> wstreambuf;
}
```

Of course, the class templates `basic_istream<>` , `basic_ostream<>` , and `basic_iostream<>` are also parametrized with the character type and a traits class:

```
namespace std {
    template <typename charT,
              typename traits = char_traits<charT> >
             class  basic_istream;
    template <typename charT,
              typename traits = char_traits<charT> >
             class  basic_ostream;
    template <typename charT,
              typename traits = char_traits<charT> >
             class  basic_iostream;
}
```

As for the other classes, there are also type definitions for the instantiations of the two most important character types:

```
namespace std {
    typedef basic_istream<char>    istream;
    typedef basic_istream<wchar_t> wistream;
```

```
      typedef basic_ostream<char>    ostream;
      typedef basic_ostream<wchar_t> wostream;

      typedef basic_iostream<char>    iostream;
      typedef basic_iostream<wchar_t> wiostream;
  }
```

The types `istream` and `ostream` are the types normally used in the Western Hemisphere, where 8-bit character sets are enough.[5] `wchar_t` allow to use character sets with more than 8 bits (see Section 16.1, page 850). Note that for types `char16_t` and `char32_t`, no corresponding instantiations are provided by the C++ standard library.

[5] The classes `istream_withassign`, `ostream_withassign`, and `iostream_withassign`, which are present in some older stream libraries (derived from `istream`, `ostream`, and `iostream`, respectively), are not supported by the standard. The corresponding functionality is achieved differently (see Section 15.12.3, page 822).

The C++ standard library provides additional stream classes for formatted I/O with files (see Section 15.9, page 791) and strings (see Section 15.10, page 802).

15.2.2. Global Stream Objects

Several global stream objects are defined for the stream classes. These objects are for access to the standard I/O channels mentioned previously for streams, with `char` as the character type and a set of corresponding objects for the streams using `wchar_t` as the character type (see Table 15.2).

Table 15.2. Global Stream Objects

Type	Name	Purpose
`istream`	`cin`	Reads input from the standard input channel
`ostream`	`cout`	Writes "normal" output to the standard output channel
`ostream`	`cerr`	Writes error messages to the standard error channel
`ostream`	`clog`	Writes log messages to the standard logging channel
`wistream`	`wcin`	Reads wide-character input from the standard input channel
`wostream`	`wcout`	Writes "normal" wide-character output to the standard output channel
`wostream`	`wcerr`	Writes wide-character error messages to the standard error channel
`wostream`	`wclog`	Writes wide-character log messages to the standard logging channel

By default, these standard streams are synchronized with the standard streams of C. That is, the C++ standard library ensures that the order of mixed output with C++ streams and C streams is preserved. Before it writes data, any buffer of standard C++ streams flushes the buffer of the corresponding C streams, and vice versa. Of course, this synchronization takes some time. If it isn't necessary, you can turn it off by calling `sync_with_stdio(false)` before any input or output is done (see Section 15.14.1, page 845).

Since C++11, some guarantees regarding concurrency are given for these stream objects: When synchronized with the standard streams of C, using them in multiple parallel threads does *not* cause undefined behavior. Thus, you can write from or read into multiple threads. Note, however, that this might result in interleaved characters, or the thread that gets a character read is undefined. For any other stream object or if these objects are not synchronized with C streams, concurrent reads or writes result in undefined behavior.

15.2.3. Header Files

The definitions of the stream classes are scattered among several header files:

- **<iosfwd>** contains forward declarations for the stream classes. This header file is necessary because it is no longer permissible to use a simple forward declaration, such as `class ostream`.
- **<streambuf>** contains the definitions for the stream buffer base class (`basic_streambuf<>`).
- **<istream>** contains the definitions for the classes that support input only (`basic_istream<>`) and for the classes that support both input and output (`basic_iostream<>`).[6]

[6] At first, `<istream>` might not appear to be a logical choice for declaration of the classes for input *and* output. However, because there may be some initialization overhead at start-up for every translation unit that includes `<iostream>` (see the following paragraph for details), the declarations for input *and* output were put into `<istream>`.

- **<ostream>** contains the definitions for the output stream class (`basic_ostream<>`).
- **<iostream>** contains declarations of the global stream objects, such as `cin` and `cout`.

Most of the headers exist for the internal organization of the C++ standard library. For the application programmer, it should be sufficient to include `<iosfwd>` for the declaration of the stream classes and `<istream>` or `<ostream>` when using the input or output functions, respectively. The header `<iostream>` should be included only if the standard stream objects are to be used. For some implementations, some code is executed at start-up for each translation unit including this header. The code being executed is not that expensive, but it requires loading the corresponding pages of the executable, which might be expensive. In general, only those headers defining necessary “stuff” should be included. In particular, header files should include only `<iosfwd>`, and the corresponding implementation files should then include the header with the complete definition.

For special stream features, such as parametrized manipulators, file streams, or string streams, there are additional headers: `<iomanip>`, `<fstream>`, `<sstream>`, and `<strstream>`. The details about these headers are provided in the sections that introduce these special features.