**PART I: WHAT ARE STREAMS IN THE FIRST PLACE**

Input and output functionality is not defined as part of the core C++ language, but rather is provided through the C++ standard library (and thus resides in the std namespace). In previous lessons, you included the iostream library header and made use of the std::cin and std::cout objects to do simple I/O. In this lesson, we’ll look at the iostream library in more detail. [Gotten from your lesson on streams]

**WHAT IS STREAM INPUT AND OUTPUT?**

Generally, input and output are the transfer of data between a program and any kind external device. Examples of external devices are files, communication channels, display windows and so on.

The transport to and from an external device can be organized in different ways:

1. Stream I/O: the transferred data can be seen as an unstructured stream of bytes, characters, or any other kind of small units of equal size. Input and output in this case are conceptually seen as a stream of data flowing between the program and the external device.
2. Record or Block I/O: alternatively, a certain structure might be imposed on the transferred data, such as a record, block, or message structure, in that case larger chunks of data, i.e., records, blocks, or messages, are transported. And these chunks may also contain information additional to the actual data.

The main difference between stream I/O and record or block I/O is that input and output for record or block I/O are structured and additional information is transported along with the actual data.

The standard iostreams, as the name already implies, supports stream I/O. This does not mean, however, that the actual external device may not have any structure, only that the concept of iostreams is the of stream I/O, and that the specifics of the actual device are hidden behind the iostreams interface.

**DATA REPRESENTATION USED BY IOSTREAM**

So far, we have talked about how iostreams organizes the transportation of data, but we haven’t talked about what type of data it transports. What is the content of the transferred data? And how is it represented?

To answer the given questions, let us get back to the definition of input and output mentioned above: “*input and output are the transfer of data between a program and any kind external device*”. The representation of data in a program and on an external device may differ. We distinguish between an internal and an external representation like so:

The internal representation of data is a form that is convenient for data processing in a program. Common examples are the binary format of signed integral numbers (current format being the two’s complement), the IEEE representation of floating-point numbers, or the ASCII or Unicode encodings of a string.

The external representations vary depending on the type of device and the intended use of the data. Here are some examples:

* If the data is to be displayed on a screen or printout, the external representation is in a human readable form as a sequence of characters.
* If the data is to be sent to other components in a different system environment, they might be represented in a portable data exchange format. In this case the external data is not in human-readable form but is understandable to the recipient.
* If the data are to be stored on a storage device and it is important to conserve storage space, then a compressed data representation, which is also not in human readable form is appropriate.

Depending on the type of external representation, we distinguish between text I/O and binary I/O.

If the external representation is a sequence of human readable characters, we talk of text I/O but, if it is in any other format, we talk of binary I/O. The main purpose of standard C++ iostreams is to facilitate text I/O, binary I/O is not directly supported.

In sum, the standard C++ iostreams is designed as a means of stream input and output of text.

**STAGES OF STREAM TEXT I/O THAT IOSTREAMS USES**

1. Formatting/parsing
2. Buffering
3. Code Conversion
4. Transport

**Formatting/parsing** is the transformation between a byte sequence representing the internal data and a character sequence. Consider an integral number, for instance. It is represented as a sequence of bytes internally and is converted to a sequence of digit and sign characters for display externally. The diagram below shows an example:

**Formatting**

**Parsing**

158

-259

int {158} //0x0000009e

int x{-259}; //0xfffffefd

Internal/program Data representation External Device representation

Note that the external representations above are a bunch of character sequences that means for example, During an output operation, instead of the byte sequence of integer value 158, used by the program internally, to be outputted, it is instead formatted to a series of character sequences that match the integer value, whereas during an input operation, the external representation of the data value -259, is parsed as a series of byte sequences that match the integer value -259 and given to the variable x of type int.

**Buffering** is the maintenance of character sequences between formatting /parsing and transport to/from the external device. For output, smaller character sequences are compiled into a buffer after formatting, until they are eventually transported to the external device as a larger block of data. Conversely, larger amounts of input are read from an external device, stored in a buffer, and made available for parsing in smaller portions. By default, input and output operations in iostreams are buffered. However, buffering is optional and can be suppressed.

In generic programming, a buffer is a basic term that denotes a computer memory block that acts a temporary placeholder. It is typically a place in storage where data can be kept before processing of the data can be started. In the case of output, it is a temporary placeholder of formatted data to be given to the external device, while in the case of input, it is a temporary placeholder of received data from an external device to be parsed to the program ‘s internal representation.

**Code Conversion** is the process of translating one character representation into another. Code conversion is necessary if the character representation that results from formatting is different from the external character representation, or if the external representation differs from the representation that can be parsed by iostreams. So in short, for output, it means if the external character representation is different from the one internally used by a program, it is converted in this stage to the appropriate character encoding that the external representation uses after formatting, whereas for input, if the encoding of a character in an external representation is different from the internal character representation of a program, it is then converted in this stage to match that of the internal character representation, which would then be parsed accurately by iostreams.

Note, the code conversion stage is done for output after formatting has already been done, while for input, it is done before parsing of the character sequences even begins.

**Transport** involves access to the external device for reading and writing data. It manages the physical transfer of character sequences to the device after formatting, buffering, and code conversion, as well as extracting data from the device and making the received data available as a sequence of characters for subsequent code conversion, buffering, and parsing.

**CLASSES IN IOSTREAMS**

When you include the iostream header, you gain access to a whole hierarchy of classes responsible for providing I/O functionality (including one class that is named iostream). You can find a class hierarchy diagram for the non-file I/O classes [here](https://en.cppreference.com/w/cpp/io).

The first thing you may notice about this hierarchy is that it uses multiple inheritance (that thing we told you to avoid if possible). However, the iostream library has been designed and extensively tested to avoid any of the typical multiple inheritance problems, so you can use it freely without worrying. [Gotten from your lesson on streams]

Here is a brief overview of the stream classes in iostreams. There are two stream base classes that encapsulate information and functionality common to all stream classes. Class std::ios\_base encapsulates all information that is independent of the character type handled by a stream. Class std::basic\_ios<CharT, Traits> is a class template taking the character type as a template argument. It contains character-type dependent information common to all stream classes. Then there are the general input and output stream classes, i.e. std::basic\_istream<CharT, Traits>, std::basic\_ostream<CharT, Traits> and std::basic\_iostream<CharT, Traits>, that implement the concepts of input, output, and bidirectional I/O. They provide the entire functionality for parsing of input and formatting for output. However, they do not contain any information that is specific to the external device associated with the stream.

Additional classes are derived from the iostream classes that facilitate input and output to files and strings in memory (commonly called in-memory I/O).

There are several typedefs of the iostream classes, all living the std namespace, that are worth mentioning:

**Narrow Characters Iostream typedefs**

 typedef basic\_ios<char>       ios;

 typedef basic\_istream<char>       istream;

 typedef basic\_ostream<char>       ostream;

 typedef basic\_iostream<char>      iostream;

 typedef basic\_istringstream<char>     istringstream;

 typedef basic\_ostringstream<char>     ostringstream;

 typedef basic\_stringstream<char>  stringstream;

 typedef basic\_ifstream<char>      ifstream;

 typedef basic\_ofstream<char>      ofstream;

 typedef basic\_fstream<char>       fstream;

**Wide Character Iostream typedefs**

 typedef basic\_ios<wchar\_t>        wios;

 typedef basic\_istream<wchar\_t>    wistream;

 typedef basic\_ostream<wchar\_t>    wostream;

 typedef basic\_iostream<wchar\_t>   wiostream;

 typedef basic\_istringstream<wchar\_t>  wistringstream;

 typedef basic\_ostringstream<wchar\_t>  wostringstream;

 typedef basic\_stringstream<wchar\_t>   wstringstream;

 typedef basic\_ifstream<wchar\_t>   wifstream;

 typedef basic\_ofstream<wchar\_t>   wofstream;

 typedef basic\_fstream<wchar\_t>    wfstream;

**THE PREDEFINED GLOBAL STREAMS**

There are eight predefined global stream objects, which are automatically created and initialized at program start:

|  |  |  |
| --- | --- | --- |
| **Narrow character stream** | **Wide Character stream** | **I/O** |
| std::cin | std::wcin | I |
| std::cout | std::wcout | O |
| std::cerr | std::wcerr | O |
| std::clog | std::wclog | O |

std::cin - an istream object tied to the standard input (typically the keyboard).

std::cout - an ostream object tied to the standard output (typically the monitor).

std::cerr - an ostream object tied to the standard error (typically the monitor), providing unbuffered output.

std::clog - an ostream object tied to the standard error (typically the monitor), providing buffered output.

The wide character counterpart all have the same setting, just uses a different character representation.

[Gotten from your stream lesson]

**PART II: FORMATTED INPUT AND OUPUT**