Modern std::byte stream IO for C++

Document #:	R2146R2
Date:	2020-07-10
Project:	Programming Language C++
Audience:	LEWGI
Reply-to:	Amanda Kornoushenko <amanda@korno.net>

Contents

1 Abstract

This paper proposes fundamental IO concepts, customization points for serialization and deserialization and streams for memory and file IO.

2 Changelog

2.1 R1 -> R2

— Expanded motivation section by adding examples using `std::streambuf`, C API and providing additional rationale.
— Redone design decisions section from scratch.
— Refer to [P1467R4] instead of [P1468R3].
— Put private data members first in RIFF example.
— Updated benchmarks and added `std::filebuf` to comparison.

2.2 R0 -> R1

— Introduced `std::io::offset` and `std::io::position` strong types.
— `get_position` member functions of streams now return `std::io::position`.
— There are now 4 overloads of `seek_position` member functions of streams that take advantage of strong types, provide more `noexcept` and minimize amount of branches in the code.
— Changed wording of `read_some` and `write_some` member functions of streams to take advantage of `std::io::position`.
— Added `std::io::creation::always_new`.

3 Motivation

C++ has text IO for a long time. However, there is no comfortable way to read and write binary data. There are several approaches provided by the standard library.

3.1 High level streams

Beginners who use `std::cout` (and sometimes `std::cin`) a lot quickly learn the API provided by `std::istream` and `std::ostream` and note that they can use unformatted IO like so:

```cpp
int my_value = 42;

{
    std::ofstream stream{"test.bin", std::ios_base::out |
        std::ios_base::binary};
    stream.exceptions(std::ios_base::failbit | std::ios_base::badbit);
    stream.write(reinterpret_cast<const char*>(&my_value), sizeof(my_value));
}

int read_value;

{
    std::ifstream stream{"test.bin", std::ios_base::in | std::ios_base::binary};
    stream.exceptions(std::ios_base::failbit | std::ios_base::badbit);
    stream.read(reinterpret_cast<char*>(&read_value), sizeof(read_value));
}

assert(read_value == my_value);
```

But it has many drawbacks:

- The API still works in terms of `char` so if you use `std::byte` in your code base, you have to `reinterpret_cast` when calling `read` and `write` member functions of streams.
- Streams operate in terms of `std::char_traits` which is not needed when doing binary IO and only complicates the API. In particular, `std::ios::pos_type` is a very painful type to work with but is required in many IO operations.
- Stream objects carry a lot of text formatting flags that are irrelevant when doing binary IO. This leads to wasted memory.
- By default, stream operations don't throw exceptions. This usually means some wrapper code to force exceptions.
- If you want to do IO in memory, you're stuck with string streams that operate using `std::string`. Most binary data is stored in `std::vector<std::byte>` which leads to loss of performance due to unnecessary copies.
- There are other problems that are discussed below.

3.2 `std::streambuf`

More experienced developers note that they can skip `std::istream` and `std::ostream` and go directly for `std::streambuf` API:

```cpp
int my_value = 42;

{
    std::filebuf file;
    auto result = file.open("test.bin", std::ios_base::out | std::ios_base::binary);
    if (result == nullptr)
    {
        /* Handle error */
    }
    auto bytes_written = file.sputn(reinterpret_cast<const char*>(&my_value),
        sizeof(my_value));
    if (bytes_written < sizeof(my_value))
    {
        /* Handle partial write */
    }
}

int read_value;

{
    std::filebuf file;
    auto result = file.open("test.bin", std::ios_base::in | std::ios_base::binary);
    if (result == nullptr)
    {
        /* Handle error */
    }
    auto bytes_read = file.sgetn(reinterpret_cast<char*>(&read_value),
        sizeof(read_value));
    if (bytes_read < sizeof(my_value))
    {
        /* Handle partial read */
    }
}

assert(read_value == my_value);
```

This is a slight improvement but it still has many drawbacks:

— Again, the API still works in terms of `char` and all our `reinterpret_casts` are still there.
— `std::streambuf::pos_type` is still there.
— Now there is no way to enable exceptions. Manually written wrapper classes are necessary.
— There are other problems that are discussed below.

3.3 Going back to C

And, of course, people who remember C standard library can write something like this:

```
int my_value = 42;

auto file = std::fopen("test.bin", "wb");
if (file == nullptr)
{
    /* Handle error */
}
auto result = std::fwrite(my_value, sizeof(my_value), 1, file);
if (result < 1)
{
    /* Handle error */
}
if (std::fclose(file) == EOF)
{
    /* Handle error */
}

int read_value;

file = std::fopen("test.bin", "rb");
if (file == nullptr)
{
    /* Handle error */
}
result = std::fread(read_value, sizeof(read_value), 1, file);
if (result < 1)
{
    /* Handle error */
}
if (std::fclose(file) == EOF)
{
    /* Handle error? */
}

assert(read_value == my_value);
```

This suffers from common drawbacks of any low level code:

— There is no RAII. Such approach would require writing wrapper classes implementing The Rule Of Five.
— There is `void*` in the code but we already need to write wrapper classes anyway.
— ...and we need to throw our own exceptions too.
— ...and there are still other problems that we didn't solve with any approach.

3.4 Unsolved problems

All three approaches suffer from some obvious quality of life deficiencies:

— You have to pass the size manually. There is no "magic" to transform our variables into strongly typed sequences of bytes. This is easy to fix in wrapper classes though.
— All 3 APIs define different "text modes" and "binary modes" and text mode is the default. This requires more code in wrapper classes to fix.

However, there are bigger problems:

— The bytes are copied as-is so you have to arrange them manually, for example, if you want specific endianness. Writing such code in a portable way ranges from non trivial to impossible. Portable endianness conversion of integers can be written using arithmetic shifts but 100% portable conversion of floating point formats is impossible as of C++20 because the standard places almost no guarantees on floating point types.
— There is no agreed standard for customization points for binary IO and serialization. You can convert compatible types to text via `std::basic_ostream::operator<<` but there is no such thing for serialization. This leads to different libraries developing their own private conventions and that makes different libraries incompatible with each other.

This proposal tries to fix all mentioned issues.

4 Prior art

This proposal is based on author's serialization library which was initially written in 2010 targeting C++98 and was gradually updated to C++20. The library is used to work with the following formats:

— Standard MIDI file
— Microsoft RIFF WAVE
— QuakeC bytecode
— WebAssembly module

The following lessons were learned during development:

— Endianness is of utmost importance. Standard MIDI files are always big-endian. Network byte order is big-endian. RIFF can be either big or little. Most newer formats are little-endian. A user must be in control

of endianness at all times. There should be transparent way to do endianness conversion. Endianness may change in the middle of the file in case of container formats such as RIFF MIDI.
— Integers should be two's complement by default. While C++98 through C++17 allowed integers to be stored as ones' complement or sign+magnitude, the author is not aware of any file format that uses those representations. C++20 requires integers to be two's complement. A user working with exotic format can supply user defined type that does bit fiddling manually. Such is the case with WebAssembly that uses LEB128 for integers.
— There should be a way to read and write floating point types in ISO 60559 `binaryN` formats regardless of the native format of floating point types. Most formats store floating point values in ISO 60559 formats. If floating point types provided by the implementation such as `float` and `double` are not ISO 60559, then the implementation is effectively cut off from the rest of the world unless there are conversion functions. Though the recent rise of `bfloat16` format shows that storage of floating point numbers continues to evolve.

The following problems were encountered:

— There was no byte type. This was fixed by `std::byte` in C++17.
— There was no sound way to express a range of bytes. This was fixed by `std::span` in C++20.
— There was no portable way to determine the native endianness, especially since sizes of all fundamental types can be 1 and all fixed-width types are optional. This was fixed by `std::endian` in C++20.
— There was no easy way to convert integers from native representation to two's complement and vice versa. This was fixed by requiring all integers to be two's complement in C++20.
— There is no easy way to convert integers from native endianness to specific endianness and vice versa. There is an `std::byteswap` proposal ([P1272R2]) but it doesn't solve the general case because C++ allows systems that are neither big- nor little-endian.
— There is no easy way to convert floating point number from native represenation to ISO/IEC 60559 and vice versa. This makes makes portable serialization of floating point numbers very hard on non-IEC platforms. [P1467R4] should fix this.

While the author thinks that having endianness and floating point conversion functions available publicly is a good idea, they leave them as implementation details in this paper.

Thoughts on [Boost.Serialization]:

— It uses confusing operator overloading akin to standard text streams which leads to several problems such as unnecessary complexity of `>>` and `<<` returning a reference to the archive.
— It doesn't support portable serialization of floating point values.
— It tries to do too much by adding version number to customization points, performing magic on pointers, arrays, several standard containers and general purpose boost classes.
— Unfortunate macro to split `load` and `save` customization points.
— It still uses standard text streams as archives.

Thoughts on [Cereal]:

— It decided to inherit several Boost problems for the sake of compatibility.
— Strange `operator()` syntax for IO.
— Will not compile if `CHAR_BIT > 8`.
— Undefined behavior when detecting native endianness due to strict aliasing violation.
— Doesn't support portable serialization of floating point values, but gives helpful `static_assert` in case of non-IEC platform.
— Still uses standard text streams as archives.

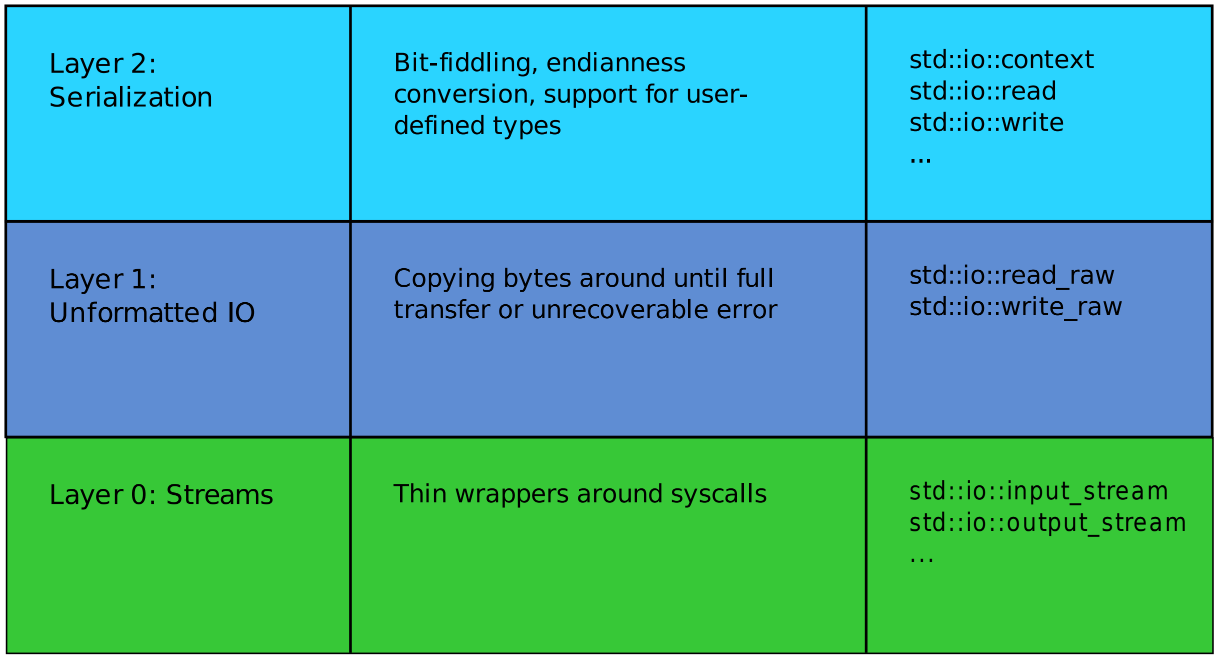
5 Design goals

— Always use `std::byte` instead of `char` when meaning raw bytes. Avoid `char*`, `unsigned char*` and `void*`.
— Do not do any text processing or hold any text-related data inside stream classes, even as template parameters.
— Provide intuitive customization points.

— Support common endiannesses and floating point formats.
— Stream classes should efficiently map to OS API in case of file IO.

6 Design decisions

This proposal consists of 3 layers:



Layer 2: Serialization	Bit-fiddling, endianness conversion, support for user-defined types	std::io::context std::io::read std::io::write ...
Layer 1: Unformatted IO	Copying bytes around until full transfer or unrecoverable error	std::io::read_raw std::io::write_raw
Layer 0: Streams	Thin wrappers around syscalls	std::io::input_stream std::io::output_stream ...

The user is free to choose a layer to work with based on their needs. In author's experience, however, serialization layer is the most commonly used one and the rest act as implementation details.

6.1 Basic definitions

Streams are used to work with sequences of bytes. Each stream has an underlying byte sequence. A stream can support reading, writing or both. A stream that supports reading in called an input stream. A stream that supports writing is called an output stream. A stream that supports both is called an input output stream. Input streams are similar to `std::ranges::input_range` in that they only guarantee to be readable only once. Same applies to output streams.

A stream can be seekable. In that case it is similar to `std::ranges::random_access_range`. Seekable streams have stream position that can be moved similar to random access iterator. Unlike iterators, stream position is represented as a signed integer so it has stronger guarantees when it comes to invalidation. Seekable streams have definite beginning and end. These are also called base positions.

A stream can be buffered. In that case there is a temporary buffer between the underlying byte sequence. When reading, bytes are copied from the underlying byte sequence into the buffer before copying them outside. When writing, bytes are copied into the buffer before writing them to the underlying byte sequence. A buffer can be flushed. In case of an input stream this empties the buffer and forces next read to be from the underlying byte sequence. In case of an output stream this forces the buffer to be written to the underlying byte sequence. In case of an input output stream the semantics depend on the type of the last operation.

6.2 Streams

The design of this proposal stems from one core principle - embracing static polymorphism and `constexpr` to gain the most runtime performance while keeping the code type-safe. Dynamic polymorphism is provided as an optional feature using type erasure. The intention is to have user code be templates that operate with the following concepts:

- `std::io::input_stream` concept matches a stream that supports reading. This requires only one member function - `read_some` that reads zero or more bytes from the stream and returns amount of bytes read.
- `std::io::output_stream` concept matches a stream that supports writing. This requires only one member function - `write_some` that writes one or more bytes to the stream and returns amount of bytes written.
- `std::io::stream` concept matches a stream that supports either reading or writing.
- `std::io::input_output_stream` concept matches a stream that supports both reading and writing.
- `std::io::seekable_stream` concept matches a stream that supports seeking. This requires `std::io::stream` and the following member functions:
  - `get_position` that returns the current position of the stream.
  - `seek_position` that takes `std::io::position` and sets the stream position to the given one.
  - `seek_position` that takes `std::io::offset` and sets the stream position to the given offset relative to the current position of the stream.
  - `seek_position` that takes `std::io::base_position` and sets stream position to the given base position.
  - `seek_position` that takes `std::io::base_position` and `std::io::offset` and sets the stream position to the given offset relative to the given base position.
- `std::io::buffered_stream` concept matches a stream that supports buffering. This requires `std::io::stream` and one another member function - `flush()` that either discards the input buffer or flushes the output buffer.

It is of note that all mentioned functionality has almost direct conceptual counterparts in `std::basic_streambuf` like so:

Stream member function	**`std::basic_streambuf` counterpart**
`read_some`	`sgetn`
`write_some`	`sputn`
`get_position`	Return value of `pubseekoff` and `pubseekpos`
`seek_position`	`pubseekoff` and `pubseekpos`
`flush`	`pubsync`

Of course, just having concepts and no actual streams to work with is not very useful. This proposal specifies 4 groups of streams, each group having an input stream, output stream and input output stream.

The first group is span streams. They are named like that because they use `std::span<[const] std::byte>` as their underlying byte sequence. This way you can read and write bytes to and from fixed-size buffer such as an array allocated on the stack. The size of the span can change at runtime but it is user's responsibility to do it. The following stream classes are provided:

- `std::io::input_span_stream`.
- `std::io::output_span_stream`.
- `std::io::input_output_span_stream`.

The second group is memory streams. These are templates that take sequence container of bytes as their template parameter. They will grow the container automatically (TODO: what about `std::array`?) as you write to them. The following stream class templates are provided:

- `std::io::basic_input_memory_stream`.
- `std::io::basic_output_memory_stream`.
- `std::io::basic_input_output_memory_stream`.

And the following aliases provided that use `std::vector<std::byte>` as the underlying byte sequence:

— `std::io::input_memory_stream`.
— `std::io::output_memory_stream`.
— `std::io::input_output_memory_stream`.

The third group is file streams. As you have guessed it, they use files are their underlying byte sequence. However, they also match `std::io::buffered_stream` so that file IO is fast by default even if you read or write small chunks. One notable improvement over `std::basic_filebuf` is usage of `std::io::mode` and `std::io::creation` that are modeled after the ones from [P1031R2] instead of `std::ios_base::openmode`. It is also the intention that file handles from [P1031R2] will be usable as file streams. `native_handle_type` is also provided for OS specific code. The following stream classes are provided:

— `std::io::input_file_stream`.
— `std::io::output_file_stream`.
— `std::io::input_output_file_stream`.

And fourth and the final group is polymorphic streams. They use type erasure and concepts to avoid virtual functions in public API. Their primary use case is providing elegant redirection of standard streams that are binary counterparts to `std::cin`, `std::cout` and `std::cerr`. The following stream classes are provided:

— `std::io::any_input_stream`.
— `std::io::any_output_stream`.
— `std::io::any_input_output_stream`.

And the following functions are provided to access standard stream objects:

— `std::io::in`.
— `std::io::out`.
— `std::io::err`.

Other important stream types such as pipes, FIFOs and sockets are not provided in this paper. It is hopeful that compatible pipe streams will be provided by later iteration of [P1750R1] and socket streams by Networking TS.

6.3 Unformatted IO

`read_some` and `write_some` member functions of streams are worded in a such way that they are implementable as thin wrappers over OS syscalls. This is deliberate to allow the users the most control over their code. One downside of them is that they do not guarantee that the IO buffer is transferred fully. In most cases the user code would want to keep calling these functions until the buffer is fully transferred or a hard error is encountered. This proposal defines the following customization points for such use case:

— `std::io::read_raw`
— `std::io::write_raw`

As an additional convenience, these customization points accept single bytes and byte-sized integral types. One important point is that these customization points operate on sequences of bytes with `std::memcpy`-like semantics. They don't care of what is supposed to be inside those bytes.

6.4 Serialization

Of course, in most cases we care about what is inside our bytes. Otherwise, the type system wouldn't be invented. And one of the most important parts of ensuring that the type system is sound is that there are deterministic rules of how our types are transformed into sequences of bytes and back when transferring them across files, processes or networks. This is known as serialization.

From the point of C++, there are 2 major types of values: integers and floating point numbers. Historically, there have been several ways to represent integers in memory. But C++20 (and many other languages and CPU architectures before that) has settled on two's complement. Floating point numbers, however, are still in flux. ISO/IEC/IEEE 60559:2011 specifies 8 interchange formats while there is one non-ISO format in the popular use -

bfloat16. And C++20 specifies almost no constraints on the implementation of floating point types. Another partially orthogonal issue is endianness. Modern CPUs are mostly little-endian. However, big-endian file formats are still in use. And, of course, the number of bits in a byte, also known as `CHAR_BIT` in C++. This is almost always 8 but modern DSPs sometimes use 16 or 32. However, the author is not aware of any file format with non-8-bit-bytes.

When designing a serialization API it is important to balance the exising wording of ISO C++ with the needs of actual hardware and file formats:

What	In ISO C++	In hardware	In file formats
Byte size	`>= 8`	Almost always 8	8
Endianness	`std::endian`	Big and little	Big and little
Integer format	Two's complement	Two's complement	Two's complement and derivatives
Floating point format	Implementation-defined	Many	Many

The following choices were made:

— `CHAR_BIT` issue is sidestepped by operating with object representations of types. Developers compiling for implementations with non-8-bit-bytes are assumed to know what they're doing.
— For integers, two's complement is assumed with support for all 3 values of `std::endian` as stream endianness.
— For floating point types, there is an escape hatch to format them as ISO 60559 `binaryN` formats with specified endianness. Otherwise, byte-wise copy is performed with no endianness conversion. This design choice will be changed if [P1467R4] is accepted.
— Developers are free to wrap their types in user-defined classes that do bit-fiddling manually during [de]serialization.

Serialization is implemented with `std::io::write` customization point. Deserialization is implemented `std::io::read` customization point. `std::io::format` class is used to represent the format of fundamental integral and floating point types when calling these CPOs. It is tied to the stream using an IO context.

IO contexts are (potentially short-lived) objects that hold a reference to the stream and `std::io::format`. The following concepts are provided:

— `std::io::context`
— `std::io::input_context`
— `std::io::output_context`

These can be used in user code that provides custom [de]serialization functions. `std::io::default_context` class template is provided for convenience. Custom [de]serilization code can be provided as either `read` and `write` member functions of user-defined types or `read` and `write` free functions. These can take additional arguments that would need to be provided to serialization CPOs.

`std::io::readable_from` and `std::io::writable_to` concepts can be used to check if types are [de]serializable.

7 Tutorial

7.1 Example 1: Reading and writing raw bytes

In this example we write some bytes to a file, then read them back and check that the bytes match. Here we use `std::io::write_raw` and `std::io::read_raw` customization points. They work with raw bytes and do not try to interpret any data inside those bytes.

```
#include <io>
#include <iostream>
```

```
int main()
{
    // Some bytes we're gonna write to a file.
    std::array<std::byte, 4> initial_bytes{
        std::byte{1},
        std::byte{2},
        std::byte{3},
        std::byte{4}};

    { // Start new RAII block.
        // Open a file for writing.
        std::io::output_file_stream stream{"test.bin"};
        // Write our bytes to the file.
        std::io::write_raw(initial_bytes, stream);
    } // End of RAII block. This will close the stream.

    // Create space for bytes to read from the file.
    std::array<std::byte, 4> read_bytes;

    { // Start new RAII block.
        // Open the file again, but now for reading.
        std::io::input_file_stream stream{"test.bin"};
        // Read the bytes from the file.
        std::io::read_raw(read_bytes, stream);
    } // End of RAII block. This will close the stream.

    // Compare read bytes with initial ones.
    if (read_bytes == initial_bytes)
    {
        std::cout << "Bytes match.\n";
    }
    else
    {
        std::cout << "Bytes don't match.\n";
    }
}
```

7.2 Example 2: Writing integer with default format

Here we write the integer to memory stream and then inspect individual bytes of the stream to see how the integer was serialized. We use high level `std::io::write` customization point that can accept non-byte types and can do bit-fiddling if requested.

```
#include <io>
#include <iostream>

int main()
{
    unsigned int value = 42;

    // Create a stream. This stream will write to dynamically allocated memory.
    std::io::output_memory_stream stream;

    // Create a context. Context contains format of non-byte data that is used
    // to correctly do [de]serialization. If stream answers the question
    // "Where?", context answers the question "How?".
    std::io::default_context context{stream};

    // Write the value to the stream.
    std::io::write(value, context);

    // Get reference to the buffer of the stream.
    const auto& buffer = stream.get_buffer();

    // Print the buffer.
    for (auto byte : buffer)
    {
        std::cout << std::to_integer<int>(byte) << ' ';
    }
    std::cout << '\n';
}
```

The result is implementation defined because by default the bytes of the integer are being copied as-is without any processing. This is the fastest. You don't pay for what you don't use. The output would depend on `CHAR_BIT`, `sizeof(unsigned int)` and `std::endian::native`. On AMD64 this will print:

```
42 0 0 0
```

This is because `CHAR_BIT` is 8, `sizeof(unsigned int)` is 4 and `std::endian::native == std::endian::little`.

7.3 Example 3: Writing integer with specific layout

Of course, in most real world cases you want to ensure the exact bit layout of all the types. For example, most file formats require bytes to be 8 bits wide, so it is good idea to put `static_assert(CHAR_BIT == 8)` in the code to only compile on compatible systems. Second, fundamental types such as `short`, `int` and `long` have implementation defined sizes so using them is also out of question. We need to use fixed-width integer types from `<cstdint>`. Finally, endianness. We need to explicitly specify endianness of the data that we are gonna share with the rest of the world.

```
#include <cstdint>
#include <io>
#include <iostream>

// Do not compile on systems with non-8-bit bytes.
static_assert(CHAR_BIT == 8);

int main()
{
    std::uint32_t value = 42;

    std::io::output_memory_stream stream;

    // Create a context with specific binary format.
    // Here we want our data in the stream to be in big-endian byte order.
    std::io::default_context context{stream, std::endian::big};

    // Write the value to the stream using our format.
    // This will perform endianness conversion on non-big-endian systems.
    std::io::write(value, context);
    const auto& buffer = stream.get_buffer();

    for (auto byte : buffer)
    {
        std::cout << std::to_integer<int>(byte) << ' ';
    }
    std::cout << '\n';
}
```

This will either fail to compile on systems where `CHAR_BIT != 8` or print:

```
0 0 0 42
```

7.4 Example 4: Working with floating point numbers

TODO

7.5 Example 5: User defined type with fixed format, member functions

In a lot of cases you know the format of your data at compile time. Therefore, your types can just provide `read` and `write` member functions that take a reference to stream. Then you just create context on the spot and do [de]serialization.

```
#include <io>
#include <iostream>

struct MyType
{
    int a;
    float b;

    void read(std::io::input_stream auto& stream)
    {
        // We really want only big-endian byte order here.
        std::io::default_context context{stream, std::endian::big};
        std::io::read(a, context);
        std::io::read(b, context);
    }

    void write(std::io::output_stream auto& stream) const
    {
        // We really want only big-endian byte order here.
        std::io::default_context context{stream, std::endian::big};
        std::io::write(a, context);
        std::io::write(b, context);
    }
};

int main()
{
    MyType my_object{1, 2.0f};
    std::io::output_memory_stream stream;

    // std::io::write will automatically pickup "write" member function if it
    // has a valid signature.
    std::io::write(my_object, stream);

    const auto& buffer = stream.get_buffer();

    for (auto byte : buffer)
    {
        std::cout << std::to_integer<int>(byte) << ' ';
    }
    std::cout << '\n';
}
```

7.6 Example 6: User defined type with fixed format, free functions

If for some reason you can't add member functions, you can define `read` and `write` free functions instead.

```
#include <io>
#include <iostream>

struct MyType
{
    int a;
    float b;
};

// Add "read" and "write" as free functions. They will be picked up
// automatically.
void read(MyType& object, std::io::input_stream auto& stream)
{
    std::io::default_context context{stream, std::endian::big};
    std::io::read(object.a, context);
    std::io::read(object.b, context);
}

void write(const MyType& object, std::io::output_stream auto& stream)
{
    std::io::default_context context{stream, std::endian::big};
    std::io::write(object.a, context);
    std::io::write(object.b, context);
}

int main()
{
    MyType my_object{1, 2.0f};
    std::io::output_memory_stream stream;

    std::io::write(my_object, stream);

    const auto& buffer = stream.get_buffer();

    for (auto byte : buffer)
    {
        std::cout << std::to_integer<int>(byte) << ' ';
    }
    std::cout << '\n';
}
```

7.7 Example 7: User defined type with dynamic format, member functions

In more involved cases such as containers the format of the data in inner layers may depend on data in outer layers. One common example is the header of the container specifying endianness of the data inside of the container. In this case you can provide `read` and `write` member functions that take context instead of stream and pass context from outer layers to inner layers, preserving the format recursively.

```cpp
#include <io>
#include <iostream>

struct MyType
{
    int a;
    float b;

    void read(std::io::input_context auto& context)
    {
        // Deserialize data using the context taken from the outside.
        std::io::read(a, context);
        std::io::read(b, context);
    }

    void write(std::io::output_context auto& context) const
    {
        // Serialize data using the context taken from the outside.
        std::io::write(a, context);
        std::io::write(b, context);
    }
};

int main()
{
    MyType my_object{1, 2.0f};
    std::io::output_memory_stream stream;

    // Create context at the top layer that we can pass through to lower layers.
    std::io::default_context context{stream, std::endian::big};

    std::io::write(my_object, context);

    const auto& buffer = stream.get_buffer();

    for (auto byte : buffer)
    {
        std::cout << std::to_integer<int>(byte) << ' ';
    }
    std::cout << '\n';
}
```

7.8 Example 8: User defined type with dynamic format, free functions

And again, you can do the same with free functions.

```cpp
#include <io>
#include <iostream>

struct MyType
{
    int a;
    float b;
};

void read(MyType& object, std::io::input_context auto& context)
{
    std::io::read(object.a, context);
    std::io::read(object.b, context);
}

void write(const MyType& object, std::io::output_context auto& context)
{
    std::io::write(object.a, context);
    std::io::write(object.b, context);
}

int main()
{
    MyType my_object{1, 2.0f};
    std::io::output_memory_stream stream;

    std::io::default_context context{stream, std::endian::big};

    std::io::write(my_object, context);

    const auto& buffer = stream.get_buffer();

    for (auto byte : buffer)
    {
        std::cout << std::to_integer<int>(byte) << ' ';
    }
    std::cout << '\n';
}
```

7.9 Example 9: Working with enums

Enumerations are essentially strong integers. Therefore, serializing them is the same as integers and is done out-of-the-box by `std::io::write`. However, reading is not so simple since there is no language-level mechanism to iterate the valid values. For now you have to write non-member `read` function that will read the integer and manually check if it has a legal value. It is hopeful that the need to write such boilerplate code will be resolved by reflection in the future.

```cpp
enum class MyEnum
{
    Foo,
    Bar
};

void read(MyEnum& my_enum, std::io::input_context auto& context)
```

```
{
    // Create a raw integer that is the same type as underlying type of our
    // enumeration.
    std::underlying_type_t<MyEnum> raw;

    // Read the integer from the stream.
    std::io::read(raw, context);

    // Cast it to our enumeration.
    my_enum = static_cast<MyEnum>(raw);

    // Check the value of enumeration.
    switch (my_enum)
    {
        case MyEnum::Foo:
        case MyEnum::Bar:
        {
            // The value is legal.
            return;
        }
        default:
        {
            // The value is illegal.
            throw /* ... */
        }
    }
}
```

7.10 Example 10: Resource Interchange File Format

There are 2 flavors of RIFF files: little-endian and big-endian. Endianness is determined by the ID of the first chunk. ASCII "RIFF" means little-endian, ASCII "RIFX" means big-endian. We can just read the chunk ID as sequence of bytes, create the context with the correct endianness and read the rest of the file using that context.

```
#include <io>
#include <array>
#include <vector>

namespace RIFF // Put things into separate namespace to save typing long names.
{

// Describes a single RIFF chunk. It starts with 4 byte ID, then size as 32-bit
// unsigned integer followed by the data of the chunk. The size doesn't include
// ID and size fields, only the size of raw data. If size is odd, there is 1
// byte padding so all chunks are aligned at even offsets.
struct Chunk
{
    using ID = std::array<std::byte, 4>;
    using Size = std::uint32_t;

    ID id;
    std::vector<std::byte> data;

    template <std::io::input_context C>
```

```cpp
    requires std::io::seekable_stream<typename C::stream_type>
    Chunk(C& context)
    {
        this->read(context);
    }

    template <std::io::input_context C>
    requires std::io::seekable_stream<typename C::stream_type>
    void read(C& context)
    {
        // Read the ID of the chunk.
        std::io::read(id, context);
        // Read the size of the chunk.
        Size size;
        std::io::read(size, context);
        // Read the data of the chunk.
        data.resize(size);
        std::io::read(data, context);
        // Skip padding.
        if (size % 2 == 1)
        {
            context.get_stream().seek_position(std::io::offset{1});
        }
    }

    void write(std::io::output_context auto& context) const
    {
        // Write the ID of the chunk.
        std::io::write(id, context);
        // Write the size of the chunk.
        Size size = std::size(data); // Production code would make sure there is
        // no overflow here.
        std::io::write(size, context);
        // Write the data of the chunk.
        std::io::write(data, context);
        // Write padding.
        if (size % 2 == 1)
        {
            std::io::write(std::byte{0}, context);
        }
    }

    // Returns the full size of the chunk when serializing.
    Size GetSize() const noexcept
    {
        Size size = 8 + std::size(data);
        if (size % 2 == 1)
        {
            ++size;
        }
        return size;
    }
};
```

```cpp
// C++ doesn't have ASCII literals but we can use UTF-8 literals instead.
constexpr Chunk::ID LittleEndianFile{
    std::byte{u8'R'}, std::byte{u8'I'}, std::byte{u8'F'}, std::byte{u8'F'}};
constexpr Chunk::ID BigEndianFile{
    std::byte{u8'R'}, std::byte{u8'I'}, std::byte{u8'F'}, std::byte{u8'X'}};

class File
{
    std::endian m_endianness;
    ChunkID m_form_type;
    std::vector<Chunk> m_chunks;

public:
    template <std::io::input_stream S>
    requires std::io::seekable_stream<S>
    File(S& stream)
    {
        this->read(stream);
    }

    template <std::io::input_stream S>
    requires std::io::seekable_stream<S>
    void read(S& stream)
    {
        // Read the main chunk ID.
        Chunk::ID chunk_id;
        std::io::read_raw(chunk_id, stream);
        if (chunk_id == LittleEndianFile)
        {
            // We have little endian file.
            m_endianness = std::endian::little;
        }
        else if (chunk_id == BigEndianFile)
        {
            // We have big endian file.
            m_endianness = std::endian::big;
        }
        else
        {
            throw /* ... */
        }
        // Create context with correct endianness.
        std::io::default_context context{stream, m_endianness};
        // We have set correct endianness based on the 1st chunk ID.
        // The rest of the file will be deserialized correctly according to
        // our format.
        Chunk::Size file_size;
        // Read the size of the file.
        std::io::read(file_size, context);
        // Now we can determine where the file ends.
        std::io::position end_position = stream.get_position() +
            std::io::offset{file_size};
        // Read the form type of the file.
        std::io::read(m_form_type, context);
        // Read all the chunks.
        while (stream.get_position() < end_position)
        {
            m_chunks.emplace_back(context);
        }
    }

    void write(std::io::output_stream auto& stream) const
    {
        // Write the ID of the main chunk.
        if (m_endianness == std::endian::little)
        {
            std::io::write_raw(LittleEndianFile, stream);
        }
        else if (m_endianness == std::endian::big)
        {
            std::io::write_raw(BigEndianFile, stream);
        }
        else
        {
            throw /* ... */
        }
        // Create context with correct endianness.
        std::io::default_context context{stream, m_endianness};
        // Calculate the size of the file. For that we need to sum up the size
        // of form type and sizes of all the chunks.
        Chunk::Size file_size = 4;
        for (const auto& chunk : m_chunks)
        {
            file_size += chunk.GetSize();
        }
        // Write the size of the file.
        std::io::write(file_size, context);
        // Write the form type of the file.
        std::io::write(m_form_type, context);
        // Write all the chunks.
        for (const auto& chunk : m_chunks)
        {
            std::io::write(chunk, context);
        }
    }
}

}
```

TODO: More tutorials? More explanations.

8 Implementation experience

The reference implementation is here: [cpp-io-impl]

Most of the proposal can be implemented in ISO C++. Endianness conversion of integers can be written in ISO C++ by using arithmetic shifts. Conversion of floating point numbers requires knowledge of their implementation-defined format. File IO requires calling operating system API. The following table provides some examples:

Function	POSIX	Windows	UEFI
Constructor	`open`	`CreateFile`	`EFI_FILE_PROTOCOL.Open`
Destructor	`close`	`CloseHandle`	`EFI_FILE_PROTOCOL.Close`
`get_position`	`lseek`	`SetFilePointerEx`	`EFI_FILE_PROTOCOL.GetPosition`
absolute `seek_position`	`lseek`	`SetFilePointerEx`	`EFI_FILE_PROTOCOL.SetPosition`
relative`seek_position`	`lseek`	`SetFilePointerEx`	No 1:1 mapping
base `seek_position`	`lseek`	`SetFilePointerEx`	`EFI_FILE_PROTOCOL.SetPosition`
`read_some`	`read`	`ReadFile`	`EFI_FILE_PROTOCOL.Read`
`write_some`	`write`	`WriteFile`	`EFI_FILE_PROTOCOL.Write`

8.1 Benchmarks

Hardware:

— CPU: AMD Ryzen 7 2700X running at 3.7 GHz
— RAM: 4 x 8 GiB DDR4 running at 3266 MHz
— Storage: Samsung 970 EVO 500GB (NVMe, PCIe 3.0 x4)

Software:

— OS: Debian Testing (Bullseye)
— Kernel: Linux 5.7.6.
— Compiler: GCC trunk (May 2020)

8.1.1 Reading 10 million of random `std::size_t` values from file sequentially

Type	Time (ms)
`std::FILE`	93.3
`std::filebuf`	76.9
`std::ifstream`	145.4
`std::io::input_file_stream`	75.5

`std::io::input_file_stream` is ~25% faster than `std::FILE`, comparable to `std::filebuf` and almost 2 times faster than `std::ifstream`.

8.1.2 Writing 10 million of random `std::size_t` values to file sequentially

Type	Time (ms)
`std::FILE`	120.4
`std::filebuf`	113.3
`std::ofstream`	210
`std::io::output_file_stream`	77

`std::io::output_file_stream` is ~55% faster than `std::FILE`, ~45% faster than `std::filebuf` and ~170% faster than `std::ofstream`.

9 Future work

It is hopeful that `std::io::format` will be used to handle Unicode encoding schemes during file and network IO so Unicode layer will only need to handle encoding forms.

This proposal is designed to work with [P1031R2] so users have a choice of going "under the hood" with [P1031R2] or staying more high level with streams.

10 Open issues

— Error handling using `throws` + `std::error`.
— `std::filesystem::path_view`
— Remove `std::io::floating_point_format` if [P1467R4] is accepted.
— CPO that returns constructed object for compatibility with constructor initializer lists.
— Synchronization between standard stream objects, `<iostream>` and `<cstdio>`.
— Vectored IO.
— `constexpr` file streams as a generalization of `std::embed`.

11 Wording

All text is relative to [N4849].

Move clauses 29.1 - 29.10 into a new clause 29.2 "Legacy text IO".

Add a new clause 29.1 "Binary IO".

11.1 29.1.? General [io.general]

TODO

11.2 29.1.? Header `<io>` synopsis [io.syn]

```
namespace std
{
namespace io
{

enum class io_errc
{
    bad_file_descriptor = implementation-defined,
    invalid_argument = implementation-defined,
    value_too_large = implementation-defined,
    reached_end_of_file = implementation-defined,
    interrupted = implementation-defined,
    physical_error = implementation-defined,
    file_too_large = implementation-defined
};

}

template <> struct is_error_code_enum<io::io_errc> : public true_type { };

namespace io
{
```

```
// Error handling
error_code make_error_code(io_errc e) noexcept;
error_condition make_error_condition(io_errc e) noexcept;

const error_category& category() noexcept;

class io_error;

// Class offset
class offset;

constexpr bool operator==(offset lhs, offset rhs) noexcept;
constexpr strong_ordering operator<=>(offset lhs, offset rhs) noexcept;

constexpr offset operator+(offset lhs, offset rhs) noexcept;
constexpr offset operator-(offset lhs, offset rhs) noexcept;
constexpr offset operator*(offset lhs, streamoff rhs) noexcept;
constexpr offset operator*(streamoff lhs, offset rhs) noexcept;
constexpr offset operator/(offset lhs, streamoff rhs) noexcept;
constexpr offset operator%(offset lhs, streamoff rhs) noexcept;

// Class position
class position;

constexpr bool operator==(position lhs, position rhs) noexcept;
constexpr strong_ordering operator<=>(position lhs, position rhs) noexcept;

constexpr position operator+(position lhs, offset rhs) noexcept;
constexpr position operator+(offset lhs, position rhs) noexcept;
constexpr position operator-(position lhs, offset rhs) noexcept;
constexpr offset operator-(position lhs, position rhs) noexcept;

enum class base_position
{
    beginning,
    current,
    end
};

// Stream concepts
template <typename T>
concept input_stream = see below;
template <typename T>
concept output_stream = see below;
template <typename T>
concept stream = see below;
template <typename T>
concept input_output_stream = see below;
template <typename T>
concept seekable_stream = see below;
template <typename T>
concept buffered_stream = see below;
```

```
// Customization points for unformatted IO
inline constexpr unspecified read_raw = unspecified;
inline constexpr unspecified write_raw = unspecified;

enum class floating_point_format
{
    iec559,
    native
};

class format;

// Context concepts
template <typename C>
concept context = see below;
template <typename C>
concept input_context = see below;
template <typename C>
concept output_context = see below;

template <stream S>
class default_context;

// Customization points for serialization
inline constexpr unspecified read = unspecified;
inline constexpr unspecified write = unspecified;

// Serialization concepts
template <typename T, typename I, typename... Args>
concept readable_from = see below;
template <typename T, typename O, typename... Args>
concept writable_to = see below;

// Polymorphic stream wrappers
class any_input_stream;
class any_output_stream;
class any_input_output_stream;

// Standard stream objects
any_input_stream& in() noexcept;
any_output_stream& out() noexcept;
any_output_stream& err() noexcept;

// Span streams
class input_span_stream;
class output_span_stream;
class input_output_span_stream;

// Memory streams
template <typename Container>
class basic_input_memory_stream;
template <typename Container>
class basic_output_memory_stream;
```

```
template <typename Container>
class basic_input_output_memory_stream;

using input_memory_stream = basic_input_memory_stream<vector<byte>>;
using output_memory_stream = basic_output_memory_stream<vector<byte>>;
using input_output_memory_stream = basic_memory_stream<vector<byte>>;

// File streams

enum class mode
{
    read,
    write
};

enum class creation
{
    open_existing,
    if_needed,
    truncate_existing,
    always_new
};

class file_stream_base;
class input_file_stream;
class output_file_stream;
class input_output_file_stream;

}
}
```

11.3 29.1.? Error handling [io.errors]

```
const error_category& category() noexcept;
```

Returns: A reference to an object of a type derived from class `error_category`. All calls to this function shall return references to the same object.

Remarks: The object's `default_error_condition` and `equivalent` virtual functions shall behave as specified for the class `error_category`. The object's `name` virtual function shall return a pointer to the string `"io"`.

```
error_code make_error_code(io_errc e) noexcept;
```

Returns: `error_code(static_cast<int>(e), io::category())`.

```
error_condition make_error_condition(io_errc e) noexcept;
```

Returns: `error_condition(static_cast<int>(e), io::category())`.

11.4 29.1.? Class `io_error` [ioerr.ioerr]

```
class io_error : public system_error
{
public:
    io_error(const string& message, error_code ec);
    io_error(const char* message, error_code ec);
};
```

TODO

11.5 29.1.? Class `offset` [io.offset]

```
class offset final
{
public:
    // Constructors
    constexpr offset() = default;
    constexpr explicit offset(streamoff off) noexcept;

    // Observer
    constexpr streamoff value() const noexcept;

    // Arithmetic
    constexpr offset operator+() const noexcept;
    constexpr offset operator-() const noexcept;
    constexpr offset& operator++() noexcept;
    constexpr offset operator++(int) noexcept;
    constexpr offset& operator--() noexcept;
    constexpr offset operator--(int) noexcept;

    constexpr offset& operator+=(offset rhs) noexcept;
    constexpr offset& operator-=(offset rhs) noexcept;

    constexpr offset& operator*=(streamoff rhs) noexcept;
    constexpr offset& operator/=(streamoff rhs) noexcept;
    constexpr offset& operator%=(streamoff rhs) noexcept;
private:
    streamoff off_; // exposition only
};
```

TODO

11.5.1 29.1.?.? Constructors [io.offset.cons]

```
constexpr explicit offset(streamoff off) noexcept;
```

Postconditions: `off_ == off`.

11.5.2 29.1.?.? Observer [io.offset.observer]

```
constexpr streamoff value() const noexcept;
```

Returns: `off_`.

11.5.3 29.1.?.? Arithmetic [io.offset.arithmetic]

```
constexpr offset operator+() const noexcept;
```

Returns: `*this`.

```
constexpr offset operator-() const noexcept;
```

Returns: `offset(-off_)`.

```
constexpr offset& operator++() noexcept;
```

Effects: Equivalent to: `++off_`.

Returns: `*this`.

```
constexpr offset operator++(int) noexcept;
```

Effects: Equivalent to: `return offset(off_++)`.

```
constexpr offset& operator--() noexcept;
```

Effects: Equivalent to: `--off_`.

Returns: `*this`.

```
constexpr offset operator--(int) noexcept;
```

Effects: Equivalent to: `return offset(off_--)`.

```
constexpr offset& operator+=(offset rhs) noexcept;
```

Effects: Equivalent to: `off_ += rhs.value()`.

Returns: `*this`.

```
constexpr offset& operator-=(offset rhs) noexcept;
```

Effects: Equivalent to: `off_ -= rhs.value()`.

Returns: `*this`.

```
constexpr offset& operator*=(streamoff rhs) noexcept;
```

Effects: Equivalent to: `off_ *= rhs`.

Returns: `*this`.

```
constexpr offset& operator/=(streamoff rhs) noexcept;
```

Effects: Equivalent to: `off_ /= rhs`.

Returns: `*this`.

```
constexpr offset& operator%=(streamoff rhs) noexcept;
```

Effects: Equivalent to: `off_ %= rhs`.

Returns: `*this`.

11.5.4 29.1.?.? Comparisons [io.offset.comparisons]

```
constexpr bool operator==(offset lhs, offset rhs) noexcept;
```

Returns: `lhs.value() == rhs.value()`.

```
constexpr strong_ordering operator<=>(offset lhs, offset rhs) noexcept;
```

Returns: `lhs.value() <=> rhs.value()`.

11.5.5 29.1.?.? Non-member arithmetic [io.offset.nonmember]

```
constexpr offset operator+(offset lhs, offset rhs) noexcept;
```

Returns: `lhs += rhs`.

```
constexpr offset operator-(offset lhs, offset rhs) noexcept;
```

Returns: `lhs -= rhs`.

```
constexpr offset operator*(offset lhs, streamoff rhs) noexcept;
```

Returns: `lhs *= rhs`.

```
constexpr offset operator*(streamoff lhs, offset rhs) noexcept;
```

Returns: `rhs *= lhs`.

```
constexpr offset operator/(offset lhs, streamoff rhs) noexcept;
```

Returns: `lhs /= rhs`.

```
constexpr offset operator%(offset lhs, streamoff rhs) noexcept;
```

Returns: `lhs %= rhs`.

11.6 29.1.? Class `position` [io.position]

```
class position final
{
public:
    // Constructors
    constexpr position() = default;
    constexpr explicit position(streamoff pos) noexcept;
    constexpr explicit position(offset off) noexcept;

    // Observer
    constexpr streamoff value() const noexcept;

    // Arithmetic
    constexpr position& operator++() noexcept;
    constexpr position operator++(int) noexcept;
    constexpr position& operator--() noexcept;
    constexpr position operator--(int) noexcept;

    constexpr position& operator+=(offset rhs) noexcept;
    constexpr position& operator-=(offset rhs) noexcept;

    // Special values
    static constexpr position max() noexcept;
private:
    streamoff pos_; // exposition only
};
```

TODO

11.6.1 29.1.?.? Constructors [io.position.cons]

```
constexpr explicit position(streamoff pos) noexcept;
```

Postconditions: `pos_ == pos`.

```
constexpr explicit position(offset off) noexcept;
```

Postconditions: `pos_ == off.value()`.

11.6.2 29.1.?.? Observer [io.position.observer]

```
constexpr streamoff value() const noexcept;
```

Returns: `pos_`.

11.6.3 29.1.?.? Arithmetic [io.position.arithmetic]

```
constexpr position& operator++() noexcept;
```

Effects: Equivalent to: `++pos_`.

Returns: `*this`.

```
constexpr position operator++(int) noexcept;
```

Effects: Equivalent to: `return position(pos_++)`.

```
constexpr position& operator--() noexcept;
```

Effects: Equivalent to: `--pos_`.

Returns: `*this`.

```
constexpr position operator--(int) noexcept;
```

Effects: Equivalent to: `return position(pos_--)`.

```
constexpr position& operator+=(position rhs) noexcept;
```

Effects: Equivalent to: `pos_ += rhs.value()`.

Returns: `*this`.

```
constexpr position& operator-=(position rhs) noexcept;
```

Effects: Equivalent to: `pos_ -= rhs.value()`.

Returns: `*this`.

11.6.4 29.1.?.? Special values [io.position.special]

```
static constexpr position max() noexcept;
```

Returns: `numeric_limits<streamoff>::max()`.

11.6.5 29.1.?.? Comparisons [io.position.comparisons]

```
constexpr bool operator==(position lhs, position rhs) noexcept;
```

Returns: `lhs.value() == rhs.value()`.

```
constexpr strong_ordering operator<=>(position lhs, position rhs) noexcept;
```

Returns: `lhs.value() <=> rhs.value()`.

11.6.6 29.1.?.? Non-member arithmetic [io.position.nonmember]

```
constexpr position operator+(position lhs, offset rhs) noexcept;
```

Returns: `lhs += rhs`.

```
constexpr position operator+(offset lhs, position rhs) noexcept;
```

Returns: `rhs += lhs`.

```
constexpr position operator-(position lhs, offset rhs) noexcept;
```

Returns: `lhs -= rhs`.

```
constexpr offset operator-(position lhs, position rhs) noexcept;
```

Returns: `offset(lhs.value() += rhs.value())`.

11.7 29.1.? Stream concepts [stream.concepts]

11.7.1 29.1.?.? Concept `input_stream` [stream.concept.input]

```
template <typename T>
concept input_stream = requires(T s, span<byte> out_buffer)
    {
        {s.read_some(out_buffer);} -> same_as<streamsize>;
    };
```

TODO

11.7.1.1 29.1.?.?.? Reading [input.stream.read]

```
streamsize read_some(span<byte> out_buffer);
```

Effects:

— If `ranges::empty(out_buffer)`, returns `0`.
— Otherwise reads zero or more bytes from the stream and advances the position by the amount of bytes read.

Returns: The amount of bytes read.

Throws: `io_error` in case of error.

Error conditions:

— `value_too_large` - if starting position is equal or greater than maximum value supported by the implementation.
— `interrupted` - if reading was iterrupted due to the receipt of a signal.
— `physical_error` - if physical I/O error has occured.

11.7.2 29.1.?.? Concept `output_stream` [stream.concept.output]

```
template <typename T>
concept output_stream = requires(T s, span<const byte> in_buffer)
    {
        {s.write_some(in_buffer);} -> same_as<streamsize>;
    };
```

TODO

11.7.2.1 29.1.?.?.? Writing [output.stream.write]

```
streamsize write_some(span<const byte> in_buffer);
```

Effects:

— If `ranges::empty(in_buffer)`, returns `0`.
— Otherwise writes one or more bytes to the stream and advances the position by the amount of bytes written.

Returns: The amount of bytes written.

Throws: `io_error` in case of error.

Error conditions:

— `file_too_large` - tried to write past the maximum size supported by the stream.
— `interrupted` - if writing was iterrupted due to the receipt of a signal.
— `physical_error` - if physical I/O error has occured.

11.7.2.2 29.1.?.? Concept `stream` [stream.concept.stream]

```
template <typename T>
concept stream = input_stream<T> || output_stream<T>;
```

TODO

11.7.2.3 29.1.?.? Concept `input_output_stream` [stream.concept.io]

```
template <typename T>
concept input_output_stream = input_stream<T> && output_stream<T>;
```

TODO

11.7.3 29.1.?.? Concept `seekable_stream` [stream.concept.seekable]

```
template <typename T>
concept seekable_stream = stream<T> && requires(const T s)
    {
        {s.get_position()} -> same_as<position>;
    } && requires(T s, position pos, offset off, base_position base)
    {
        s.seek_position(pos);
        s.seek_position(off);
        s.seek_position(base);
        s.seek_position(base, off);
    };
```

TODO

11.7.3.1 29.1.?.?.? Position [seekable.stream.position]

```
position get_position();
```

Returns: Current position of the stream.

```
void seek_position(position pos);
```

Effects: Sets the position of the stream to the given value.

Throws: `io_error` in case of error.

Error conditions:

- `invalid_argument` - if position is negative and the stream doesn't support that.
- `value_too_large` - if position is greater than the maximum size supported by the stream.

```
void seek_position(offset off);
```

Effects: Sets the position of the stream to the given offset relative to the current position of the stream.

Throws: `io_error` in case of error.

Error conditions:

- `invalid_argument` - if resulting position is negative and the stream doesn't support that.
- `value_too_large` - if resulting position cannot be represented as type `streamoff` or is greater than the maximum size supported by the stream.

```
void seek_position(base_position base);
```

Effects:

- If `base == base_position::beginning`, sets the position of the stream to the beginning of the stream.
- If `base == base_position::current`, does nothing.
- If `base == base_position::end`, sets the position of the stream to the end of the stream.

```
void seek_position(base_position base, offset off);
```

Effects:

- If `base == base_position::beginning`, calls `seek_position(position(off))`.
- If `base == base_position::current`, calls `seek_position(off)`.
- If `base == base_position::end`, sets the position of the stream to the given offset relative to the end of the stream.

Throws: `io_error` in case of error.

Error conditions:

- `invalid_argument` - if resulting position is negative and the stream doesn't support that.
- `value_too_large` - if resulting position cannot be represented as type `streamoff` or is greater than the maximum size supported by the stream.

```
template <typename T>
constexpr position move_position(T pos, offset off); // exposition only
```

Returns: `position(pos + off.value())`.

Throws: `io_error` in case of error.

Error conditions:

- `value_too_large` - if `pos + off.value()` would overflow or cannot be represented as type `streamoff`.

11.7.4 29.1.?.? Concept `buffered_stream` [stream.concept.buffered]

```
template <typename T>
concept buffered_stream = stream<T> && requires(T s)
    {
        s.flush();
    };
```

TODO

11.7.4.1 29.1.?.?.? Buffering [buffered.stream.buffer]

```
void flush();
```

Effects: If the last operation on the stream was input, resets the internal buffer. If the last operation on the stream was output, writes the contents of the internal buffer to the stream and then resets the internal buffer.

Throws: TODO

11.8 29.1.? Customization points for unformatted IO [io.raw]

11.8.1 29.1.?.1 `io::read_raw` [io.read.raw]

The name `read_raw` denotes a customization point object. The expression `io::read_raw(E, S)` for some subexpression `E` with type `T` and subexpression `S` with type `U` has the following effects:

— If `U` is not `input_stream`, `io::read_raw(E, S)` is ill-formed.
— If `T` is `byte`, reads one byte from the stream and assigns it to `E`.
— If `T` is `ranges::output_range<byte>`, for every iterator in the range reads a byte from the stream and assigns it to the said iterator.
— If `T` is `integral` and `sizeof(T) == 1`, reads one byte from the stream and assigns its object representation to `E`.

11.8.2 29.1.?.2 `io::write_raw` [io.write.raw]

The name `write_raw` denotes a customization point object. The expression `io::write_raw(E, S)` for some subexpression `E` with type `T` and subexpression `S` with type `U` has the following effects:

— If `U` is not `output_stream`, `io::write_raw(E, S)` is ill-formed.
— If `T` is `byte`, writes it to the stream.
— If `T` is `ranges::input_range` and `same_as<ranges::range_value_t<T>, byte>`, for every iterator in the range writes the iterator's value to the stream.
— If `T` is `integral` and `sizeof(T) == 1`, writes the object representation of `E` to the stream.

11.9 29.1.? Class `format` [io.format]

```
class format final
{
public:
    // Constructor
    constexpr format(endian endianness = endian::native,
        floating_point_format float_format = floating_point_format::native)
        noexcept;

    // Member functions
    constexpr endian get_endianness() const noexcept;
    constexpr void set_endianness(endian new_endianness) noexcept;
    constexpr floating_point_format get_floating_point_format() const noexcept;
    constexpr void set_floating_point_format(floating_point_format new_format)
        noexcept;

    // Equality
    friend constexpr bool operator==(const format& lhs, const format& rhs)
        noexcept = default;
private:
    endian endianness_; // exposition only
    floating_point_format float_format_; // exposition only
};
```

TODO

11.9.1 29.1.?.? Constructor [io.format.cons]

```
constexpr format(endian endianness = endian::native,
    floating_point_format float_format = floating_point_format::native)
    noexcept;
```

Postconditions: `endianness_ == endianness` and `float_format_ == float_format`.

11.9.2 29.1.?.? Member functions [io.format.members]

```
constexpr endian get_endianness() const noexcept;
```

Returns: `endianness_`.

```
constexpr void set_endianness(endian new_endianness) noexcept;
```

Postconditions: `endianness_ == new_endianness`.

```
constexpr floating_point_format get_floating_point_format() const noexcept;
```

Returns: `float_format_`.

```
constexpr void set_floating_point_format(floating_point_format new_format)
    noexcept;
```

Postconditions: `float_format_ == new_format`.

11.10 29.1.? Context concepts [io.context.concepts]

11.10.1 29.1.?.? Concept `context` [io.context]

```
template <typename C>
concept context =
    stream<typename C::stream_type> &&
    requires(const C ctx)
    {
        {ctx.get_stream()} -> same_as<const typename C::stream_type&>;
        {ctx.get_format()} -> same_as<format>;
    } && requires(C ctx, format f)
    {
        {ctx.get_stream()} -> same_as<typename C::stream_type&>;
        ctx.set_format(f);
    };
```

TODO

11.10.2 29.1.?.? Concept `input_context` [input.context]

```
template <typename C>
concept input_context = context<C> && input_stream<typename C::stream_type>;
```

TODO

11.10.3 29.1.?.? Concept `output_context` [output.context]

```
template <typename C>
concept output_context = context<C> && output_stream<typename C::stream_type>;
```

TODO

11.11 29.1.? Class template `default_context` [io.default.context]

```
template <stream S>
class default_context final
{
public:
    using stream_type = S;

    // Constructor
    constexpr default_context(S& s, format f = {}) noexcept;

    // Stream
    constexpr S& get_stream() noexcept;
    constexpr const S& get_stream() const noexcept;

    // Format
    constexpr format get_format() const noexcept;
    constexpr void set_format(format f) noexcept;
private:
    S& stream_; // exposition only
    format format_; // exposition only
};
```

TODO

11.11.1 29.1.?.? Constructor [io.default.context.cons]

```
constexpr default_context(S& s, format f = {}) noexcept;
```

Effects: Initializes `stream_` with `s`.

Postconditions: `format_ == f`.

11.11.2 29.1.?.? Stream [io.default.context.stream]

```
constexpr S& get_stream() noexcept;
```

Returns: `stream_`.

```
constexpr const S& get_stream() const noexcept;
```

Returns: `stream_`.

11.11.3 29.1.?.? Format [io.default.context.format]

```
constexpr format get_format() const noexcept;
```

Returns: `format_`.

```
constexpr void set_format(format f) noexcept;
```

Postconditions: `format_ == f`.

11.12 29.1.? Customization points for serialization [io.serialization]

11.12.1 29.1.?.? Helper concepts

```
template <typename T, typename I, typename... Args>
concept customly-readable-from = // exposition only
    (input_stream<I> || input_context<I>) &&
    requires(T object, I& i, Args&&... args)
    {
        object.read(i, forward<Args>(args)...);
    };
```

```
template <typename T, typename O, typename... Args>
concept customly-writable-to = // exposition only
    (output_stream<O> || output_context<O>) &&
    requires(const T object, O& o, Args&&... args)
    {
        object.write(o, forward<Args>(args)...);
    };
```

11.12.2 29.1.?.? `io::read` [io.read]

The name `read` denotes a customization point object. The expression `io::read(E, I, args...)` for some subexpression `E` with type `T`, subexpression `I` with type `U` and `args` with template parameter pack `Args` has the following effects:

- If `U` is not `input_stream` or `input_context`, `io::read(E, I, args...)` is ill-formed.
- If `T`,`U` and `Args` satisfy *`customly-readable-from`*`<T, U, Args...>`, calls `E.read(I, forward<Args>(args)...)`.
- Otherwise, if `sizeof...(Args) != 0`, `io::read(E, I, args...)` is ill-formed.
- If `U` is `input_stream` and:
    - If `T` is `byte` or `ranges::output_range<byte>`, calls `io::read_raw(E, I)`.
    - If `T` is `integral` and `sizeof(T) == 1`, calls `io::read_raw(E, I)`.
- If `U` is `input_context` and:
    - If `T` is `byte` or `ranges::output_range<byte>`, calls `io::read(E, I.get_stream())`.
    - If `T` is `bool`, reads 1 byte from the stream, contextually converts its value to `bool` and assigns the result to `E`.
    - If `T` is `integral`, reads `sizeof(T)` bytes from the stream, performs conversion of bytes from context endianness to native endianness and assigns the result to object representation of `E`.
    - If `T` is `floating_point`, reads `sizeof(T)` bytes from the stream and:
        - If context floating point format is `native`, assigns the bytes to the object representation of `E`.
        - If context floating point format is `iec559`, performs conversion of bytes treated as an ISO/IEC/IEEE 60559 floating point representation in context endianness to native format and assigns the result to the object representation of `E`.

11.12.3 29.1.?.? `io::write` [io.write]

The name `write` denotes a customization point object. The expression `io::write(E, O, args...)` for some subexpression `E` with type `T`, subexpression `O` with type `U` and `args` with template parameter pack `Args` has the following effects:

- If `U` is not `output_stream` or `output_context`, `io::write(E, O, args...)` is ill-formed.
- If `T`,`U` and `Args` satisfy *`customly-writable-to`*`<T, U, Args...>`, calls `E.write(O, forward<Args>(args)...)`.
- Otherwise, if `sizeof...(Args) != 0`, `io::write(E, O, args...)` is ill-formed.
- If `U` is `output_stream` and:
  - If `T` is `byte` or `ranges::input_range` and `same_as<ranges::range_value_t<T>, byte>`, calls `io::write_raw(E, O)`.
  - If `T` is `integral` or an enumeration type and `sizeof(T) == 1`, calls `io::write_raw(static_cast<byte>(E), O)`.
- If `U` is `output_context` and:
  - If `T` is `byte` or `ranges::input_range` and `same_as<ranges::range_value_t<T>, byte>`, calls `io::write(E, O.get_stream())`.
  - If `T` is `bool`, writes a single byte whose value is the result of integral promotion of `E` to the stream.
  - If `T` is `integral` or an enumeration type, performs conversion of object representation of `E` from native endianness to context endianness and writes the result to the stream.
  - If `T` is `floating_point` and:
    - If context floating point format is `native`, writes the object representation of `E` to the stream.
    - If context floating point format is `iec559`, performs conversion of object representation of `E` from native format to ISO/IEC/IEEE 60559 format in context endianness and writes the result to the stream.

11.13 29.1.? Serialization concepts [serialization.concepts]

11.13.1 29.1.?.? Concept `readable_from` [io.concept.readable]

```
template <typename T, typename I, typename... Args>
concept readable_from =
    (input_stream<I> || input_context<I>) &&
    requires(T& object, I& i, Args&&... args)
    {
        io::read(object, i, forward<Args>(args)...);
    };
```

TODO

11.13.2 29.1.?.? Concept `writable_to` [io.concept.writable]

```
template <typename T, typename O, typename... Args>
concept writable_to =
    (output_stream<O> || output_context<O>) &&
    requires(const T& object, O& o, Args&&... args)
    {
        io::write(object, o, forward<Args>(args)...);
    };
```

TODO

11.14 29.1.? Polymorphic stream wrappers [any.streams]

11.14.1 29.1.?.1 Class `any_input_stream` [any.input.stream]

```
class any_input_stream final
{
public:
    // Constructors and destructor
    constexpr any_input_stream() noexcept;
    template <input_stream S>
    constexpr any_input_stream(S&& s);
    template <input_stream S, typename... Args>
    requires constructible_from<S, Args...>
    constexpr explicit any_input_stream(in_place_type_t<S>, Args&&... args);
    constexpr any_input_stream(const any_input_stream& other);
    constexpr any_input_stream(any_input_stream&& other) noexcept;
    constexpr ~any_input_stream();

    // Assignment
    constexpr any_input_stream& operator=(const any_input_stream& other);
    constexpr any_input_stream& operator=(any_input_stream&& other) noexcept;
    template <input_stream S>
    constexpr any_input_stream& operator=(S&& s);

    // Observers
    constexpr bool has_value() const noexcept;
    constexpr const type_info& type() const noexcept;
    template <input_stream S>
    constexpr const S& get() const;
    template <input_stream S>
    constexpr S& get();

    // Modifiers
    template <input_stream S, typename... Args>
    requires constructible_from<S, Args...>
    constexpr void emplace(Args&&... args);
    template <input_stream S>
    requires movable<S>
    constexpr S release();
    constexpr void reset() noexcept;
    constexpr void swap(any_input_stream& other) noexcept;

    // Position
    constexpr position get_position() const;
    constexpr void seek_position(position pos);
    constexpr void seek_position(offset off);
    constexpr void seek_position(base_position base);
    constexpr void seek_position(base_position base, offset off);

    // Buffering
    constexpr void flush();

    // Reading
    constexpr streamsize read_some(span<byte> out_buffer);
};
```

TODO

11.14.1.1 29.1.?.?.? Constructors and destructor [any.input.stream.cons]

```
constexpr any_input_stream() noexcept;
```

Postconditions: `has_value() == false`.

```
template <input_stream S>
constexpr any_input_stream(S&& s);
```

Let `VS` be `decay_t<S>`.

Effects: Constructs an object of type `any_input_stream` that contains an object of type `VS` direct-initialized with `forward<S>(s)`.

Throws: Any exception thrown by the selected constructor of `VS`.

```
template <input_stream S, typename... Args>
requires constructible_from<S, Args...>
constexpr explicit any_input_stream(in_place_type_t<S>, Args&&... args);
```

Let `VS` be `decay_t<S>`.

Effects: Initializes the contained stream as if direct-non-list-initializing an object of type `VS` with the arguments `forward<Args>(args)...`.

Postconditions: `*this` contains a stream of type `VS`.

Throws: Any exception thrown by the selected constructor of `VS`.

```
constexpr any_input_stream(const any_input_stream& other);
```

Effects: If `other.has_value() == false`, constructs an object that has no stream. Otherwise, if contained stream of `other` is not copyable, throws exception. Otherwise, equivalent to `any_input_stream(in_place_type_t<S>, other.get())` where `S` is the type of the contained stream.

Throws: `io_error` if contained stream of `other` is not copyable. Otherwise, any exception thrown by the selected constructor of `S`.

Error conditions:

— `bad_file_descriptor` - if contained stream of `other` is not copyable.

```
constexpr any_input_stream(any_input_stream&& other) noexcept;
```

Effects: If `other.has_value() == false`, constructs an object that has no stream. Otherwise, constructs an object of type `any_input_stream` that contains the contained stream of `other`.

Postconditions: `other.has_value() == false`.

```
constexpr ~any_input_stream();
```

Effects: As if by `reset()`.

11.14.1.2 29.1.?.?.? Assignment [any.input.stream.assign]

```
constexpr any_input_stream& operator=(const any_input_stream& other);
```

Effects: As if by `any_input_stream(other).swap(*this)`. No effects if an exception is thrown.

Returns: `*this`.

Throws: Any exceptions arising from the copy constructor for the contained stream.

```
constexpr any_input_stream& operator=(any_input_stream&& other) noexcept;
```

Effects: As if by `any_input_stream(move(other)).swap(*this)`.

Returns: `*this`.

Postconditions: The state of `*this` is equivalent to the original state of `other`.

```
template <input_stream S>
constexpr any_input_stream& operator=(S&& s);
```

Let `VS` be `decay_t<S>`.

Effects: Constructs an object `tmp` of type `any_input_stream` that contains a stream of type `VS` direct-initialized with `std::forward<S>(s)`, and `tmp.swap(*this)`. No effects if an exception is thrown.

Returns: `*this`.

Throws: Any exception thrown by the selected constructor of `VS`.

11.14.1.3 29.1.?.?.? Observers [any.input.stream.observers]

```
constexpr bool has_value() const noexcept;
```

Returns: `true` if `*this` contains a stream, otherwise `false`.

```
constexpr const type_info& type() const noexcept;
```

Returns: `typeid(S)` if `*this` contains a stream of type `S`, otherwise `typeid(void)`.

```
template <input_stream S>
constexpr const S& get() const;
template <input_stream S>
constexpr S& get();
```

Let `VS` be `decay_t<S>`.

Returns: Reference to the contained stream.

Throws: `bad_any_cast` if `type() != typeid(VS)`.

11.14.1.4 29.1.?.?.? Modifiers [any.input.stream.modifiers]

```
template <input_stream S, typename... Args>
requires constructible_from<S, Args...>
constexpr void emplace(Args&&... args);
```

Let `VS` be `decay_t<S>`.

Effects: Calls `reset()`. Then initializes the contained stream as if direct-non-list-initializing an object of type `VS` with the arguments `forward<Args>(args)...`.

Postconditions: `*this` contains a stream of type `VS`.

Throws: Any exception thrown by the selected constructor of `VS`.

Remarks: If an exception is thrown during the call to `VS`'s constructor, `*this` does not contain a stream, and any previously contained stream has been destroyed.

```
template <input_stream S>
requires movable<S>
constexpr S release();
```

Let `VS` be `decay_t<S>`.

Postconditions: `has_value() == false`.

Returns: The stream of type `S` constructed from the contained stream considering that contained stream as an rvalue.

Throws: `bad_any_cast` if `type() != typeid(VS)`.

```
constexpr void reset() noexcept;
```

Effects: If `has_value() == true`, destroys the contained stream.

Postconditions: `has_value() == false`.

```
constexpr void swap(any_input_stream& other) noexcept;
```

Effects: Exchanges the states of `*this` and `other`.

11.14.1.5 29.1.?.?.? Position [any.input.stream.position]

```
constexpr position get_position() const;
```

Let `s` be the contained stream of `*this` and `S` be `decltype(s)`.

Returns: `s.get_position()`.

Throws: `io_error` if `has_value() == false` or `!seekable_stream<S>`. Otherwise, any exception thrown by `s`.

Error conditions:

— `bad_file_descriptor` - if `has_value() == false` or `!seekable_stream<S>`.

```
constexpr void seek_position(position pos);
```

Let `s` be the contained stream of `*this` and `S` be `decltype(s)`.

Effects: Calls `s.seek_position(pos)`.

Throws: `io_error` if `has_value() == false` or `!seekable_stream<S>`. Otherwise, any exception thrown by `s`.

Error conditions:

— `bad_file_descriptor` - if `has_value() == false` or `!seekable_stream<S>`.

```
constexpr void seek_position(offset off);
```

Let `s` be the contained stream of `*this` and `S` be `decltype(s)`.

Effects: Calls `s.seek_position(off)`.

Throws: `io_error` if `has_value() == false` or `!seekable_stream<S>`. Otherwise, any exception thrown by `s`.

Error conditions:

— `bad_file_descriptor` - if `has_value() == false` or `!seekable_stream<S>`.

```
constexpr void seek_position(base_position base);
```

Let `s` be the contained stream of `*this` and `S` be `decltype(s)`.

Effects: Calls `s.seek_position(base)`.

Throws: `io_error` if `has_value() == false` or `!seekable_stream<S>`. Otherwise, any exception thrown by `s`.

Error conditions:

— `bad_file_descriptor` - if `has_value() == false` or `!seekable_stream<S>`.

```
constexpr void seek_position(base_position base, offset off);
```

Let `s` be the contained stream of `*this` and `S` be `decltype(s)`.

Effects: Calls `s.seek_position(base, off)`.

Throws: `io_error` if `has_value() == false` or `!seekable_stream<S>`. Otherwise, any exception thrown by `s`.

Error conditions:

— `bad_file_descriptor` - if `has_value() == false` or `!seekable_stream<S>`.

11.14.1.6 29.1.?.?.? Buffering [any.input.stream.buffer]

```
constexpr void flush();
```

Let `s` be the contained stream of `*this` and `S` be `decltype(s)`.

Effects: If `!buffered_stream<S>`, does nothing. Otherwise, calls `s.flush()`.

Throws: `io_error` if `has_value() == false`. Otherwise, any exception thrown by `s`.

Error conditions:

— `bad_file_descriptor` - if `has_value() == false`.

11.14.1.7 29.1.?.?.? Reading [any.input.stream.read]

```
constexpr streamsize read_some(span<byte> out_buffer);
```

Let `s` be the contained stream of `*this`.

Returns: `s.read_some(out_buffer)`.

Throws: `io_error` if `has_value() == false`. Otherwise, any exception thrown by `s`.

Error conditions:

— `bad_file_descriptor` - if `has_value() == false`.

11.14.2 29.1.?.2 Class `any_output_stream` [any.output.stream]

```
class any_output_stream final
{
public:
    // Constructors and destructor
    constexpr any_output_stream() noexcept;
    template <output_stream S>
    constexpr any_output_stream(S&& s);
    template <output_stream S, typename... Args>
    requires constructible_from<S, Args...>
    constexpr explicit any_output_stream(in_place_type_t<S>, Args&&... args);
    constexpr any_output_stream(const any_output_stream& other);
    constexpr any_output_stream(any_output_stream&& other) noexcept;
    constexpr ~any_output_stream();

    // Assignment
    constexpr any_output_stream& operator=(const any_output_stream& other);
    constexpr any_output_stream& operator=(any_output_stream&& other) noexcept;
    template <output_stream S>
    constexpr any_output_stream& operator=(S&& s);
    // Observers
    constexpr bool has_value() const noexcept;
    constexpr const type_info& type() const noexcept;
    template <output_stream S>
    constexpr const S& get() const;
    template <output_stream S>
    constexpr S& get();

    // Modifiers
    template <output_stream S, typename... Args>
    requires constructible_from<S, Args...>
    constexpr void emplace(Args&&... args);
    template <output_stream S>
    requires movable<S>
    constexpr S release();
    constexpr void reset() noexcept;
    constexpr void swap(any_output_stream& other) noexcept;

    // Position
    constexpr position get_position() const;
    constexpr void seek_position(position pos);
    constexpr void seek_position(offset off);
    constexpr void seek_position(base_position base);
    constexpr void seek_position(base_position base, offset off);

    // Buffering
    constexpr void flush();

    // Writing
    constexpr streamsize write_some(span<const byte> in_buffer);
};
```

TODO

11.14.2.1 29.1.?.?.? Constructors and destructor [any.output.stream.cons]

```
constexpr any_output_stream() noexcept;
```

Postconditions: `has_value() == false`.

```
template <output_stream S>
constexpr any_output_stream(S&& s);
```

Let `VS` be `decay_t<S>`.

Effects: Constructs an object of type `any_output_stream` that contains an object of type `VS` direct-initialized with `forward<S>(s)`.

Throws: Any exception thrown by the selected constructor of `VS`.

```
template <output_stream S, typename... Args>
requires constructible_from<S, Args...>
constexpr explicit any_output_stream(in_place_type_t<S>, Args&&... args);
```

Effects: Initializes the contained stream as if direct-non-list-initializing an object of type `VS` with the arguments `forward<Args>(args)...`.

Postconditions: `*this` contains a stream of type `VS`.

Throws: Any exception thrown by the selected constructor of `VS`.

```
constexpr any_output_stream(const any_output_stream& other);
```

Effects: If `other.has_value() == false`, constructs an object that has no stream. Otherwise, if contained stream of `other` is not copyable, throws exception. Otherwise, equivalent to `any_output_stream(in_place_type_t<S>, other.get())` where `S` is the type of the contained stream.

Throws: `io_error` if contained stream of `other` is not copyable. Otherwise, any exception thrown by the selected constructor of `S`.

Error conditions:

— `bad_file_descriptor` - if contained stream of `other` is not copyable.

```
constexpr any_output_stream(any_output_stream&& other) noexcept;
```

Effects: If `other.has_value() == false`, constructs an object that has no stream. Otherwise, constructs an object of type `any_output_stream` that contains the contained stream of `other`.

Postconditions: `other.has_value() == false`.

```
constexpr ~any_output_stream();
```

Effects: As if by `reset()`.

11.14.2.2 29.1.?.?.? Assignment [any.output.stream.assign]

```
constexpr any_output_stream& operator=(const any_output_stream& other);
```

Effects: As if by `any_output_stream(other).swap(*this)`. No effects if an exception is thrown.

Returns: `*this`.

Throws: Any exceptions arising from the copy constructor for the contained stream.

```
constexpr any_output_stream& operator=(any_output_stream&& other) noexcept;
```

Effects: As if by `any_output_stream(move(other)).swap(*this)`.

Returns: `*this`.

Postconditions: The state of `*this` is equivalent to the original state of `other`.

```
template <output_stream S>
constexpr any_output_stream& operator=(S&& s);
```

Let `VS` be `decay_t<S>`.

Effects: Constructs an object `tmp` of type `any_output_stream` that contains a stream of type `VS` direct-initialized with `std::forward<S>(s)`, and `tmp.swap(*this)`. No effects if an exception is thrown.

Returns: `*this`.

Throws: Any exception thrown by the selected constructor of `VS`.

11.14.2.3 29.1.?.?.? Observers [any.output.stream.observers]

```
constexpr bool has_value() const noexcept;
```

Returns: `true` if `*this` contains a stream, otherwise `false`.

```
constexpr const type_info& type() const noexcept;
```

Returns: `typeid(S)` if `*this` contains a stream of type `S`, otherwise `typeid(void)`.

```
template <output_stream S>
constexpr const S& get() const;
template <output_stream S>
constexpr S& get();
```

Let `VS` be `decay_t<S>`.

Returns: Reference to the contained stream.

Throws: `bad_any_cast` if `type() != typeid(VS)`.

11.14.2.4 29.1.?.?.? Modifiers [any.output.stream.modifiers]

```
template <output_stream S, typename... Args>
requires constructible_from<S, Args...>
constexpr void emplace(Args&&... args);
```

Let `VS` be `decay_t<S>`.

Effects: Calls `reset()`. Then initializes the contained stream as if direct-non-list-initializing an object of type `VS` with the arguments `forward<Args>(args)...`.

Postconditions: `*this` contains a stream of type `VS`.

Throws: Any exception thrown by the selected constructor of `VS`.

Remarks: If an exception is thrown during the call to `VS`'s constructor, `*this` does not contain a stream, and any previously contained stream has been destroyed.

```
template <output_stream S>
requires movable<S>
constexpr S release();
```

Let `VS` be `decay_t<S>`.

Postconditions: `has_value() == false`.

Returns: The stream of type `S` constructed from the contained stream considering that contained stream as an rvalue.

Throws: `bad_any_cast` if `type() != typeid(VS)`.

```
constexpr void reset() noexcept;
```

Effects: If `has_value() == true`, destroys the contained stream.

Postconditions: `has_value() == false`.

```
constexpr void swap(any_output_stream& other) noexcept;
```

Effects: Exchanges the states of `*this` and `other`.

11.14.2.5 29.1.?.?.? Position [any.output.stream.position]

```
constexpr position get_position() const;
```

Let `s` be the contained stream of `*this` and `S` be `decltype(s)`.

Returns: `s.get_position()`.

Throws: io_error if has_value() == false or !seekable_stream<S>. Otherwise, any exception thrown by s.

Error conditions:

— bad_file_descriptor - if has_value() == false or !seekable_stream<S>.

```
constexpr void seek_position(position pos);
```

Let s be the contained stream of *this and S be decltype(s).

Effects: Calls s.seek_position(pos).

Throws: io_error if has_value() == false or !seekable_stream<S>. Otherwise, any exception thrown by s.

Error conditions:

— bad_file_descriptor - if has_value() == false or !seekable_stream<S>.

```
constexpr void seek_position(offset off);
```

Let s be the contained stream of *this and S be decltype(s).

Effects: Calls s.seek_position(off).

Throws: io_error if has_value() == false or !seekable_stream<S>. Otherwise, any exception thrown by s.

Error conditions:

— bad_file_descriptor - if has_value() == false or !seekable_stream<S>.

```
constexpr void seek_position(base_position base);
```

Let s be the contained stream of *this and S be decltype(s).

Effects: Calls s.seek_position(base).

Throws: io_error if has_value() == false or !seekable_stream<S>. Otherwise, any exception thrown by s.

Error conditions:

— bad_file_descriptor - if has_value() == false or !seekable_stream<S>.

```
constexpr void seek_position(base_position base, offset off);
```

Let s be the contained stream of *this and S be decltype(s).

Effects: Calls s.seek_position(base, off).

Throws: io_error if has_value() == false or !seekable_stream<S>. Otherwise, any exception thrown by s.

Error conditions:

— bad_file_descriptor - if has_value() == false or !seekable_stream<S>.

11.14.2.6 29.1.?.?.? Buffering [any.output.stream.buffer]

```
constexpr void flush();
```

Let s be the contained stream of *this and S be decltype(s).

Effects: If !buffered_stream<S>, does nothing. Otherwise, calls s.flush().

Throws: io_error if has_value() == false. Otherwise, any exception thrown by s.

Error conditions:

— bad_file_descriptor - if has_value() == false.

11.14.2.7 29.1.?.?.? Writing [any.output.stream.write]

```
constexpr streamsize write_some(span<const byte> in_buffer);
```

Let `s` be the contained stream of `*this`.

Returns: `s.write_some(in_buffer)`.

Throws: `io_error` if `has_value() == false`. Otherwise, any exception thrown by `s`.

Error conditions:

- `bad_file_descriptor` - if `has_value() == false`.

11.14.3 29.1.?.3 Class `any_input_output_stream` [any.io.stream]

```
class any_input_output_stream final
{
public:
    // Constructors and destructor
    constexpr any_input_output_stream() noexcept;
    template <input_output_stream S>
    constexpr any_input_output_stream(S&& s);
    template <input_output_stream S, typename... Args>
    requires constructible_from<S, Args...>
    constexpr explicit any_input_output_stream(in_place_type_t<S>,
        Args&&... args);
    constexpr any_input_output_stream(const any_input_output_stream& other);
    constexpr any_input_output_stream(any_input_output_stream&& other) noexcept;
    constexpr ~any_input_output_stream();

    // Assignment
    constexpr any_input_output_stream& operator=(
        const any_input_output_stream& other);
    constexpr any_input_output_stream& operator=(
        any_input_output_stream&& other) noexcept;
    template <input_output_stream S>
    constexpr any_input_output_stream& operator=(S&& s);

    // Observers
    constexpr bool has_value() const noexcept;
    constexpr const type_info& type() const noexcept;
    template <input_output_stream S>
    constexpr const S& get() const;
    template <input_output_stream S>
    constexpr S& get();

    // Modifiers
    template <input_output_stream S, typename... Args>
    requires constructible_from<S, Args...>
    constexpr void emplace(Args&&... args);
    template <input_output_stream S>
    requires movable<S>
    constexpr S release();
    constexpr void reset() noexcept;
    constexpr void swap(any_input_output_stream& other) noexcept;
    // Position
    constexpr position get_position() const;
    constexpr void seek_position(position pos);
    constexpr void seek_position(offset off);
    constexpr void seek_position(base_position base);
    constexpr void seek_position(base_position base, offset off);

    // Buffering
    constexpr void flush();

    // Reading
    constexpr streamsize read_some(span<byte> out_buffer);

    // Writing
    constexpr streamsize write_some(span<const byte> in_buffer);
};
```

TODO

11.14.3.1 29.1.?.?.? Constructors and destructor [any.io.stream.cons]

```
constexpr any_input_output_stream() noexcept;
```

Postconditions: `has_value() == false`.

```
template <input_output_stream S>
constexpr any_input_output_stream(S&& s);
```

Let `VS` be `decay_t<S>`.

Effects: Constructs an object of type `any_input_output_stream` that contains an object of type `VS` direct-initialized with `forward<S>(s)`.

Throws: Any exception thrown by the selected constructor of `VS`.

```
template <input_output_stream S, typename... Args>
requires constructible_from<S, Args...>
constexpr explicit any_input_output_stream(in_place_type_t<S>, Args&&... args);
```

Effects: Initializes the contained stream as if direct-non-list-initializing an object of type `VS` with the arguments `forward<Args>(args)...`.

Postconditions: `*this` contains a stream of type `VS`.

Throws: Any exception thrown by the selected constructor of `VS`.

```
constexpr any_input_output_stream(const any_input_output_stream& other);
```

Effects: If `other.has_value() == false`, constructs an object that has no stream. Otherwise, if contained stream of `other` is not copyable, throws exception. Otherwise, equivalent to `any_input_output_stream(in_place_type_t<S>, other.` where `S` is the type of the contained stream.

Throws: `io_error` if contained stream of `other` is not copyable. Otherwise, any exception thrown by the selected constructor of `S`.

Error conditions:

- `bad_file_descriptor` - if contained stream of `other` is not copyable.

```
constexpr any_input_output_stream(any_input_output_stream&& other) noexcept;
```

Effects: If `other.has_value() == false`, constructs an object that has no stream. Otherwise, constructs an object of type `any_input_output_stream` that contains the contained stream of `other`.

Postconditions: `other.has_value() == false`.

```
constexpr ~any_input_output_stream();
```

Effects: As if by `reset()`.

11.14.3.2 29.1.?.?.? Assignment [any.io.stream.assign]

```
constexpr any_input_output_stream& operator=(
    const any_input_output_stream& other);
```

Effects: As if by `any_input_output_stream(other).swap(*this)`. No effects if an exception is thrown.

Returns: `*this`.

Throws: Any exceptions arising from the copy constructor for the contained stream.

```
constexpr any_input_output_stream& operator=(any_input_output_stream&& other)
    noexcept;
```

Effects: As if by `any_input_output_stream(move(other)).swap(*this)`.

Returns: `*this`.

Postconditions: The state of `*this` is equivalent to the original state of `other`.

```
template <input_output_stream S>
constexpr any_input_output_stream& operator=(S&& s);
```

Let `VS` be `decay_t<S>`.

Effects: Constructs an object `tmp` of type `any_input_output_stream` that contains a stream of type `VS` direct-initialized with `std::forward<S>(s)`, and `tmp.swap(*this)`. No effects if an exception is thrown.

Returns: `*this`.

Throws: Any exception thrown by the selected constructor of `VS`.

11.14.3.3 29.1.?.?.? Observers [any.io.stream.observers]

```
constexpr bool has_value() const noexcept;
```

Returns: `true` if `*this` contains a stream, otherwise `false`.

```
constexpr const type_info& type() const noexcept;
```

Returns: `typeid(S)` if `*this` contains a stream of type `S`, otherwise `typeid(void)`.

```
template <input_output_stream S>
constexpr const S& get() const;
template <input_output_stream S>
constexpr S& get();
```

Let `VS` be `decay_t<S>`.

Returns: Reference to the contained stream.

Throws: `bad_any_cast` if `type() != typeid(VS)`.

11.14.3.4 29.1.?.?.? Modifiers [any.io.stream.modifiers]

```
template <input_output_stream S, typename... Args>
requires constructible_from<S, Args...>
constexpr void emplace(Args&&... args);
```

Let `VS` be `decay_t<S>`.

Effects: Calls `reset()`. Then initializes the contained stream as if direct-non-list-initializing an object of type `VS` with the arguments `forward<Args>(args)...`.

Postconditions: `*this` contains a stream of type `VS`.

Throws: Any exception thrown by the selected constructor of `VS`.

Remarks: If an exception is thrown during the call to `VS`'s constructor, `*this` does not contain a stream, and any previously contained stream has been destroyed.

```
template <input_output_stream S>
requires movable<S>
constexpr S release();
```

Let `VS` be `decay_t<S>`.

Postconditions: `has_value() == false`.

Returns: The stream of type `S` constructed from the contained stream considering that contained stream as an rvalue.

Throws: `bad_any_cast` if `type() != typeid(VS)`.

```
constexpr void reset() noexcept;
```

Effects: If `has_value() == true`, destroys the contained stream.

Postconditions: `has_value() == false`.

```
constexpr void swap(any_input_output_stream& other) noexcept;
```

Effects: Exchanges the states of `*this` and `other`.

11.14.3.5 29.1.?.?.? Position [any.io.stream.position]

```
constexpr position get_position() const;
```

Let `s` be the contained stream of `*this` and `S` be `decltype(s)`.

Returns: `s.get_position()`.

Throws: `io_error` if `has_value() == false` or `!seekable_stream<S>`. Otherwise, any exception thrown by `s`.

Error conditions:

— `bad_file_descriptor` - if `has_value() == false` or `!seekable_stream<S>`.

```
constexpr void seek_position(position pos);
```

Let `s` be the contained stream of `*this` and `S` be `decltype(s)`.

Effects: Calls `s.seek_position(pos)`.

Throws: `io_error` if `has_value() == false` or `!seekable_stream<S>`. Otherwise, any exception thrown by `s`.

Error conditions:

— `bad_file_descriptor` - if `has_value() == false` or `!seekable_stream<S>`.

```
constexpr void seek_position(offset off);
```

Let `s` be the contained stream of `*this` and `S` be `decltype(s)`.

Effects: Calls `s.seek_position(off)`.

Throws: `io_error` if `has_value() == false` or `!seekable_stream<S>`. Otherwise, any exception thrown by `s`.

Error conditions:

— `bad_file_descriptor` - if `has_value() == false` or `!seekable_stream<S>`.

```
constexpr void seek_position(base_position base);
```

Let `s` be the contained stream of `*this` and `S` be `decltype(s)`.

Effects: Calls `s.seek_position(base)`.

Throws: `io_error` if `has_value() == false` or `!seekable_stream<S>`. Otherwise, any exception thrown by `s`.

Error conditions:

— `bad_file_descriptor` - if `has_value() == false` or `!seekable_stream<S>`.

```
constexpr void seek_position(base_position base, offset off);
```

Let `s` be the contained stream of `*this` and `S` be `decltype(s)`.

Effects: Calls `s.seek_position(base, off)`.

Throws: `io_error` if `has_value() == false` or `!seekable_stream<S>`. Otherwise, any exception thrown by `s`.

Error conditions:

— `bad_file_descriptor` - if `has_value() == false` or `!seekable_stream<S>`.

11.14.3.6 29.1.?.?.? Buffering [any.io.stream.buffer]

```
constexpr void flush();
```

Let `s` be the contained stream of `*this` and `S` be `decltype(s)`.

Effects: If `!buffered_stream<S>`, does nothing. Otherwise, calls `s.flush()`.

Throws: `io_error` if `has_value() == false`. Otherwise, any exception thrown by `s`.

Error conditions:

— `bad_file_descriptor` - if `has_value() == false`.

11.14.3.7 29.1.?.?.? Reading [any.io.stream.read]

```
constexpr streamsize read_some(span<byte> out_buffer);
```

Let `s` be the contained stream of `*this`.

Returns: `s.read_some(out_buffer)`.

Throws: `io_error` if `has_value() == false`. Otherwise, any exception thrown by `s`.

Error conditions:

— `bad_file_descriptor` - if `has_value() == false`.

11.14.3.8 29.1.?.?.? Writing [any.io.stream.write]

```
constexpr streamsize write_some(span<const byte> in_buffer);
```

Let `s` be the contained stream of `*this`.

Returns: `s.write_some(in_buffer)`.

Throws: `io_error` if `has_value() == false`. Otherwise, any exception thrown by `s`.

Error conditions:

— `bad_file_descriptor` - if `has_value() == false`.

11.15 29.1.? Standard stream objects [stream.objects]

```
any_input_stream& in() noexcept;
any_output_stream& out() noexcept;
any_output_stream& err() noexcept;
```

11.15.1 29.1.?.? Overview [stream.objects.overview]

Standard stream objects defined in this clause have static storage duration and are initialized during the first call to the function that returns a reference to them. Types of default contained streams are implementation-defined. Concurrent access to default contained streams by multiple threads does not result in a data race.

11.15.2 29.1.?.? Functions [stream.objects.functions]

```
any_input_stream& in() noexcept;
```

Returns: Reference to the standard stream object initialized with the standard input stream.

```
any_output_stream& out() noexcept;
```

Returns: Reference to the standard stream object initialized with the standard output stream.

```
any_output_stream& err() noexcept;
```

Returns: Reference to the standard stream object initialized with the standard error stream.

11.16 29.1.? Span streams [span.streams]

11.16.1 29.1.?.1 Class `input_span_stream` [input.span.stream]

```
class input_span_stream final
{
public:
    // Constructors
    constexpr input_span_stream() noexcept;
    constexpr input_span_stream(span<const byte> buffer) noexcept;

    // Position
    constexpr position get_position() const noexcept;
    constexpr void seek_position(position pos);
    constexpr void seek_position(offset off);
    constexpr void seek_position(base_position base) noexcept;
    constexpr void seek_position(base_position base, offset off);
    // Reading
    constexpr streamsize read_some(span<byte> out_buffer);

    // Buffer management
    constexpr span<const byte> get_buffer() const noexcept;
    constexpr void set_buffer(span<const byte> new_buffer) noexcept;
private:
    span<const byte> buffer_; // exposition only
    ptrdiff_t pos_; // exposition only
};
```

TODO

11.16.1.1 29.1.?.?.? Constructors [input.span.stream.cons]

```
constexpr input_span_stream() noexcept;
```

Postconditions:

- ranges::empty(buffer_) == true,
- pos_ == 0.

```
constexpr input_span_stream(span<const byte> buffer) noexcept;
```

Postconditions:

- ranges::data(buffer_) == ranges::data(buffer),
- ranges::ssize(buffer_) == ranges::ssize(buffer),
- pos_ == 0.

11.16.1.2 29.1.?.?.? Position [input.span.stream.position]

```
constexpr position get_position() const noexcept;
```

Returns: `position(pos_)`.

```
constexpr void seek_position(position pos);
```

Postconditions: `pos_ == pos.value()`.

Throws: `io_error` in case of error.

Error conditions:

- `invalid_argument` - if position is negative.
- `value_too_large` - if position cannot be represented as type `ptrdiff_t`.

```
constexpr void seek_position(offset off);
```

Effects: Calls `seek_position(`*`move_position`*`(pos_, off))`

Throws: `io_error` in case of error.

Error conditions:

- `invalid_argument` - if resulting position is negative.
- `value_too_large` - if resulting position cannot be represented as type `streamoff` or `ptrdiff_t`.

```
constexpr void seek_position(base_position base) noexcept;
```

Effects:

- If `base == base_position::beginning`, equivalent to `pos_ = 0`.

— If `base == base_position::current`, does nothing.
— If `base == base_position::end`, equivalent to `pos_ = min(ranges::ssize(buffer_), position::max().value())`.

```
constexpr void seek_position(base_position base, offset off);
```

Effects:

— If `base == base_position::beginning`, calls `seek_position(position(off))`.
— If `base == base_position::current`, calls `seek_position(off)`.
— If `base == base_position::end`, calls `seek_position(`*`move_position`*`(ranges::ssize(buffer_), off))`.

Throws: `io_error` in case of error.

Error conditions:

— `invalid_argument` - if resulting position is negative.
— `value_too_large` - if resulting position cannot be represented as type `streamoff` or `ptrdiff_t`.

11.16.1.3 29.1.?.?.? Reading [input.span.stream.read]

```
constexpr streamsize read_some(span<byte> out_buffer);
```

Effects:

— If `ranges::empty(out_buffer)`, returns `0`.
— If `pos_ >= ranges::ssize(buffer_)`, returns `0`.
— If `pos_ == position::max().value()`, throws exception.
— Otherwise determines the amount of bytes to read so that it satisfies the following constrains:
  — Must be less than or equal to `ranges::ssize(out_buffer)`.
  — Must be representable as `streamsize`.
  — Position after the read must be less than or equal to `ranges::ssize(buffer_)`.
  — Position after the read must be representable as `streamoff`.
— After that reads that amount of bytes from the stream to the given buffer and advances stream position by the amount of bytes read.

Returns: The amount of bytes read.

Throws: `io_error` in case of error.

Error conditions:

— `value_too_large` - if `!ranges::empty(out_buffer)` and `pos_ == position::max().value()`.

11.16.1.4 29.1.?.?.? Buffer management [input.span.stream.buffer]

```
constexpr span<const byte> get_buffer() const noexcept;
```

Returns: `buffer_`.

```
constexpr void set_buffer(span<const byte> new_buffer) noexcept;
```

Postconditions:

— `ranges::data(buffer_) == ranges::data(new_buffer)`,
— `ranges::ssize(buffer_) == ranges::ssize(new_buffer)`,
— `pos_ == 0`.

11.16.2 29.1.?.2 Class `output_span_stream` [output.span.stream]

```
class output_span_stream final
{
public:
    // Constructors
    constexpr output_span_stream() noexcept;
    constexpr output_span_stream(span<byte> buffer) noexcept;

    // Position
    constexpr position get_position() const noexcept;
    constexpr void seek_position(position pos);
    constexpr void seek_position(offset off);
    constexpr void seek_position(base_position base) noexcept;
    constexpr void seek_position(base_position base, offset off);

    // Writing
    constexpr streamsize write_some(span<const byte> in_buffer);

    // Buffer management
    constexpr span<byte> get_buffer() const noexcept;
    constexpr void set_buffer(span<byte> new_buffer) noexcept;
private:
    span<byte> buffer_; // exposition only
    ptrdiff_t pos_; // exposition only
};
```

TODO

11.16.2.1 29.1.?.?.? Constructors [output.span.stream.cons]

```
constexpr output_span_stream() noexcept;
```

Postconditions:

- `ranges::empty(buffer_) == true`,
- `pos_ == 0`.

```
constexpr output_span_stream(span<byte> buffer) noexcept;
```

Postconditions:

- `ranges::data(buffer_) == ranges::data(buffer)`,
- `ranges::ssize(buffer_) == ranges::ssize(buffer)`,
- `pos_ == 0`.

11.16.2.2 29.1.?.?.? Position [output.span.stream.position]

```
constexpr position get_position() const noexcept;
```

Returns: `position(pos_)`.

```
constexpr void seek_position(position pos);
```

Postconditions: `pos_ == pos.value()`.

Throws: `io_error` in case of error.

Error conditions:

- `invalid_argument` - if position is negative.
- `value_too_large` - if position cannot be represented as type `ptrdiff_t`.

```
constexpr void seek_position(offset off);
```

Effects: Calls `seek_position(move_position(pos_, off))`

Throws: `io_error` in case of error.

Error conditions:

— `invalid_argument` - if resulting position is negative.
— `value_too_large` - if resulting position cannot be represented as type `streamoff` or `ptrdiff_t`.

```
constexpr void seek_position(base_position base) noexcept;
```

Effects:

— If `base == base_position::beginning`, equivalent to `pos_ = 0`.
— If `base == base_position::current`, does nothing.
— If `base == base_position::end`, equivalent to `pos_ = min(ranges::ssize(buffer_), position::max().value())`.

```
constexpr void seek_position(base_position base, offset off);
```

Effects:

— If `base == base_position::beginning`, calls `seek_position(position(off))`.
— If `base == base_position::current`, calls `seek_position(off)`.
— If `base == base_position::end`, calls `seek_position(move_position(ranges::ssize(buffer_), off))`.

Throws: `io_error` in case of error.

Error conditions:

— `invalid_argument` - if resulting position is negative.
— `value_too_large` - if resulting position cannot be represented as type `streamoff` or `ptrdiff_t`.

11.16.2.3 29.1.?.?.? Writing [output.span.stream.write]

```
constexpr streamsize write_some(span<const byte> in_buffer);
```

Effects:

— If `ranges::empty(in_buffer)`, returns `0`.
— If `pos_ >= ranges::ssize(buffer_)` or `pos_ == position::max().value()`, throws exception.
— Otherwise determines the amount of bytes to write so that it satisfies the following constrains:
  — Must be less than or equal to `ranges::ssize(in_buffer)`.
  — Must be representable as `streamsize`.
  — Position after the write must be less than or equal to `ranges::ssize(buffer_)`.
  — Position after the write must be representable as `streamoff`.
— After that writes that amount of bytes from the given buffer to the stream and advances stream position by the amount of bytes written.

Returns: The amount of bytes written.

Throws: `io_error` in case of error.

Error conditions:

— `file_too_large` - if `!ranges::empty(in_buffer) && ((pos_ == ranges::ssize(buffer_)) || (pos_ == position`

11.16.2.4 29.1.?.?.? Buffer management [output.span.stream.buffer]

```
constexpr span<byte> get_buffer() const noexcept;
```

Returns: `buffer_`.

```
constexpr void set_buffer(span<byte> new_buffer) noexcept;
```

Postconditions:

— `ranges::data(buffer_) == ranges::data(new_buffer)`,
— `ranges::ssize(buffer_) == ranges::ssize(new_buffer)`,
— `pos_ == 0`.

11.16.3 29.1.?.3 Class `input_output_span_stream` [io.span.stream]

```
class input_output_span_stream final
{
public:
    // Constructors
    constexpr input_output_span_stream() noexcept;
    constexpr input_output_span_stream(span<byte> buffer) noexcept;

    // Position
    constexpr position get_position() const noexcept;
    constexpr void seek_position(position pos);
    constexpr void seek_position(offset off);
    constexpr void seek_position(base_position base) noexcept;
    constexpr void seek_position(base_position base, offset off);

    // Reading
    constexpr streamsize read_some(span<byte> out_buffer);

    // Writing
    constexpr streamsize write_some(span<const byte> in_buffer);

    // Buffer management
    constexpr span<byte> get_buffer() const noexcept;
    constexpr void set_buffer(span<byte> new_buffer) noexcept;
private:
    span<byte> buffer_; // exposition only
    ptrdiff_t pos_; // exposition only
};
```

TODO

11.16.3.1 29.1.?.?.? Constructors [io.span.stream.cons]

```
constexpr input_output_span_stream() noexcept;
```

Postconditions:

— `ranges::empty(buffer_) == true`,
— `pos_ == 0`.

```
constexpr input_output_span_stream(span<byte> buffer) noexcept;
```

Postconditions:

— `ranges::data(buffer_) == ranges::data(buffer)`,
— `ranges::ssize(buffer_) == ranges::ssize(buffer)`,
— `pos_ == 0`.

11.16.3.2 29.1.?.?.? Position [io.span.stream.position]

```
constexpr position get_position() const noexcept;
```

Returns: `position(pos_)`.

```
constexpr void seek_position(position pos);
```

Postconditions: `pos_ == pos.value()`.

Throws: `io_error` in case of error.

Error conditions:

- `invalid_argument` - if position is negative.
- `value_too_large` - if position cannot be represented as type `ptrdiff_t`.

```
constexpr void seek_position(offset off);
```

Effects: Calls `seek_position(`*`move_position`*`(pos_, off))`

Throws: `io_error` in case of error.

Error conditions:

- `invalid_argument` - if resulting position is negative.
- `value_too_large` - if resulting position cannot be represented as type `streamoff` or `ptrdiff_t`.

```
constexpr void seek_position(base_position base) noexcept;
```

Effects:

- If `base == base_position::beginning`, equivalent to `pos_ = 0`.
- If `base == base_position::current`, does nothing.
- If `base == base_position::end`, equivalent to `pos_ = min(ranges::ssize(buffer_), position::max().value())`.

```
constexpr void seek_position(base_position base, offset off);
```

Effects:

- If `base == base_position::beginning`, calls `seek_position(position(off))`.
- If `base == base_position::current`, calls `seek_position(off)`.
- If `base == base_position::end`, calls `seek_position(`*`move_position`*`(ranges::ssize(buffer_), off))`.

Throws: `io_error` in case of error.

Error conditions:

- `invalid_argument` - if resulting position is negative.
- `value_too_large` - if resulting position cannot be represented as type `streamoff` or `ptrdiff_t`.

11.16.3.3 29.1.?.?.? Reading [io.span.stream.read]

```
constexpr streamsize read_some(span<byte> out_buffer);
```

Effects:

- If `ranges::empty(out_buffer)`, returns `0`.
- If `pos_ >= ranges::ssize(buffer_)`, returns `0`.
- If `pos_ == position::max().value()`, throws exception.
- Otherwise determines the amount of bytes to read so that it satisfies the following constrains:
  - Must be less than or equal to `ranges::ssize(out_buffer)`.
  - Must be representable as `streamsize`.
  - Position after the read must be less than or equal to `ranges::ssize(buffer_)`.

  - Position after the read must be representable as `streamoff`.
- After that reads that amount of bytes from the stream to the given buffer and advances stream position by the amount of bytes read.

Returns: The amount of bytes read.

Throws: `io_error` in case of error.

Error conditions:

- `value_too_large` - if `!ranges::empty(out_buffer)` and `pos_ == position::max().value()`.

11.16.3.4 29.1.?.?.? Writing [io.span.stream.write]

```
constexpr streamsize write_some(span<const byte> in_buffer);
```

Effects:

- If `ranges::empty(in_buffer)`, returns `0`.
- If `pos_ >= ranges::ssize(buffer_)` or `pos_ == position::max().value()`, throws exception.
- Otherwise determines the amount of bytes to write so that it satisfies the following constrains:
  - Must be less than or equal to `ranges::ssize(in_buffer)`.
  - Must be representable as `streamsize`.
  - Position after the write must be less than or equal to `ranges::ssize(buffer_)`.
  - Position after the write must be representable as `streamoff`.
- After that writes that amount of bytes from the given buffer to the stream and advances stream position by the amount of bytes written.

Returns: The amount of bytes written.

Throws: `io_error` in case of error.

Error conditions:

- `file_too_large` - if `!ranges::empty(in_buffer) && ((pos_ == ranges::ssize(buffer_)) || (pos_ == position`

11.16.3.5 29.1.?.?.? Buffer management [io.span.stream.buffer]

```
constexpr span<byte> get_buffer() const noexcept;
```

Returns: `buffer_`.

```
constexpr void set_buffer(span<byte> new_buffer) noexcept;
```

Postconditions:

- `ranges::data(buffer_) == ranges::data(new_buffer)`,
- `ranges::ssize(buffer_) == ranges::ssize(new_buffer)`,
- `pos_ == 0`.

11.17 29.1.? Memory streams [memory.streams]

11.17.1 29.1.?.1 Class template `basic_input_memory_stream` [input.memory.stream]

```
template <typename Container>
class basic_input_memory_stream final
{
public:
    // Constructors
    constexpr basic_input_memory_stream();
    constexpr basic_input_memory_stream(const Container& c);
    constexpr basic_input_memory_stream(Container&& c);

    // Position
    constexpr position get_position() const noexcept;
    constexpr void seek_position(position pos);
    constexpr void seek_position(offset off);
    constexpr void seek_position(base_position base) noexcept;
    constexpr void seek_position(base_position base, offset off);

    // Reading
    constexpr streamsize read_some(span<byte> out_buffer);

    // Buffer management
    constexpr const Container& get_buffer() const & noexcept;
    constexpr Container get_buffer() && noexcept;
    constexpr void set_buffer(const Container& new_buffer);
    constexpr void set_buffer(Container&& new_buffer);
    constexpr void reset_buffer() noexcept;
private:
    Container buffer_; // exposition only
    typename Container::difference_type pos_; // exposition only
};
```

TODO

11.17.1.1 29.1.?.?.? Constructors [input.memory.stream.cons]

```
constexpr basic_input_memory_stream();
```

Postconditions:

- `buffer_ == Container{}`,
- `pos_ == 0`.

```
constexpr basic_input_memory_stream(const Container& c);
```

Effects: Initializes `buffer_` with `c`.

Postconditions: `pos_ == 0`.

```
constexpr basic_input_memory_stream(Container&& c);
```

Effects: Initializes `buffer_` with `move(c)`.

Postconditions: `pos_ == 0`.

11.17.1.2 29.1.?.?.? Position [input.memory.stream.position]

```
constexpr position get_position() const noexcept;
```

Returns: `position(pos_)`.

```
constexpr void seek_position(position pos);
```

Postconditions: `pos_ == pos.value()`.

Throws: `io_error` in case of error.

Error conditions:

— `invalid_argument` - if position is negative.
— `value_too_large` - if position cannot be represented as type `typename Container::difference_type`.

```
constexpr void seek_position(offset off);
```

Effects: Calls `seek_position(`*`move_position`*`(pos_, off))`

Throws: `io_error` in case of error.

Error conditions:

— `invalid_argument` - if resulting position is negative.
— `value_too_large` - if resulting position cannot be represented as type `streamoff` or `typename Container::difference_t`

```
constexpr void seek_position(base_position base) noexcept;
```

Effects:

— If `base == base_position::beginning`, equivalent to `pos_ = 0`.
— If `base == base_position::current`, does nothing.
— If `base == base_position::end`, equivalent to `pos_ = min(ranges::ssize(buffer_), position::max().value())`.

```
constexpr void seek_position(base_position base, offset off);
```

Effects:

— If `base == base_position::beginning`, calls `seek_position(position(off))`.
— If `base == base_position::current`, calls `seek_position(off)`.
— If `base == base_position::end`, calls `seek_position(`*`move_position`*`(ranges::ssize(buffer_), off))`.

Throws: `io_error` in case of error.

Error conditions:

— `invalid_argument` - if resulting position is negative.
— `value_too_large` - if resulting position cannot be represented as type `streamoff` or `typename Container::difference_t`

11.17.1.3 29.1.?.?.? Reading [input.memory.stream.read]

```
constexpr streamsize read_some(span<byte> out_buffer);
```

Effects:

— If `ranges::empty(out_buffer)`, returns 0.
— If `pos_ >= ranges::ssize(buffer_)`, returns 0.
— If `pos_ == position::max().value()`, throws exception.
— Otherwise determines the amount of bytes to read so that it satisfies the following constrains:
  — Must be less than or equal to `ranges::ssize(out_buffer)`.
  — Must be representable as `streamsize`.
  — Position after the read must be less than or equal to `ranges::ssize(buffer_)`.
  — Position after the read must be representable as `streamoff`.
— After that reads that amount of bytes from the stream to the given buffer and advances stream position by the amount of bytes read.

Returns: The amount of bytes read.

Throws: `io_error` in case of error.

Error conditions:

— `value_too_large` - if `!ranges::empty(out_buffer)` and `pos_ == position::max().value()`.

11.17.1.4 29.1.?.?.? Buffer management [input.memory.stream.buffer]

```
constexpr const Container& get_buffer() const & noexcept;
```

Returns: `buffer_`.

```
constexpr Container get_buffer() && noexcept;
```

Returns: `move(buffer_)`.

```
constexpr void set_buffer(const Container& new_buffer);
```

Postconditions:

— `buffer_ == new_buffer`.
— `pos_ == 0`.

```
constexpr void set_buffer(Container&& new_buffer);
```

Effects: Move assigns `new_buffer` to `buffer_`.

Postconditions: `pos_ == 0`.

```
constexpr void reset_buffer() noexcept;
```

Effects: Equivalent to `buffer_.clear()`.

Postconditions: `pos_ == 0`.

11.17.2 29.1.?.2 Class template `basic_output_memory_stream` [output.memory.stream]

```
template <typename Container>
class basic_output_memory_stream final
{
public:
    // Constructors
    constexpr basic_output_memory_stream();
    constexpr basic_output_memory_stream(const Container& c);
    constexpr basic_output_memory_stream(Container&& c);

    // Position
    constexpr position get_position() const noexcept;
    constexpr void seek_position(position pos);
    constexpr void seek_position(offset off);
    constexpr void seek_position(base_position base) noexcept;
    constexpr void seek_position(base_position base, offset off);

    // Writing
    constexpr streamsize write_some(span<const byte> in_buffer);

    // Buffer management
    constexpr const Container& get_buffer() const & noexcept;
    constexpr Container get_buffer() && noexcept;
    constexpr void set_buffer(const Container& new_buffer);
    constexpr void set_buffer(Container&& new_buffer);
    constexpr void reset_buffer() noexcept;
private:
    Container buffer_; // exposition only
    typename Container::difference_type pos_; // exposition only
};
```

TODO

11.17.2.1 29.1.?.?.? Constructors [output.memory.stream.cons]

```
constexpr basic_output_memory_stream();
```

Postconditions:

- buffer_ == Container{},
- pos_ == 0.

```
constexpr basic_output_memory_stream(const Container& c);
```

Effects: Initializes buffer_ with c.

Postconditions: pos_ == 0.

```
constexpr basic_output_memory_stream(Container&& c);
```

Effects: Initializes buffer_ with move(c).

Postconditions: pos_ == 0.

11.17.2.2 29.1.?.?.? Position [output.memory.stream.position]

```
constexpr position get_position() const noexcept;
```

Returns: position(pos_).

```
constexpr void seek_position(position pos);
```

Postconditions: pos_ == pos.value().

Throws: io_error in case of error.

Error conditions:

- invalid_argument - if position is negative.
- value_too_large - if position cannot be represented as type typename Container::difference_type.

```
constexpr void seek_position(offset off);
```

Effects: Calls seek_position(*move_position*(pos_, off))

Throws: io_error in case of error.

Error conditions:

- invalid_argument - if resulting position is negative.
- value_too_large - if resulting position cannot be represented as type streamoff or typename Container::difference_t

```
constexpr void seek_position(base_position base) noexcept;
```

Effects:

- If base == base_position::beginning, equivalent to pos_ = 0.
- If base == base_position::current, does nothing.
- If base == base_position::end, equivalent to pos_ = min(ranges::ssize(buffer_), position::max().value()).

```
constexpr void seek_position(base_position base, offset off);
```

Effects:

- If `base == base_position::beginning`, calls `seek_position(position(off))`.
- If `base == base_position::current`, calls `seek_position(off)`.
- If `base == base_position::end`, calls `seek_position(`*`move_position`*`(ranges::ssize(buffer_), off))`.

Throws: `io_error` in case of error.

Error conditions:

- `invalid_argument` - if resulting position is negative.
- `value_too_large` - if resulting position cannot be represented as type `streamoff` or `typename Container::difference_t`

11.17.2.3 29.1.?.?.? Writing [output.memory.stream.write]

```
constexpr streamsize write_some(span<const byte> in_buffer);
```

Effects:

- If `ranges::empty(in_buffer)`, returns `0`.
- If `pos_ >= buffer_.max_size()` or `pos_ == position::max().value()`, throws exception.
- If `pos_ < ranges::ssize(buffer_)`:
  - Determines the amount of bytes to write so that it satisfies the following constrains:
    - Must be less than or equal to `ranges::ssize(in_buffer)`.
    - Must be representable as `streamsize`.
    - Position after the write must be less than or equal to `ranges::ssize(buffer_)`.
    - Position after the write must be representable as `streamoff`.
  - Writes that amount of bytes from the given buffer to the stream and advances stream position by the amount of bytes written.
- Otherwise:
  - Determines the amount of bytes to write so that it satisfies the following constrains:
    - Must be less than or equal to `ranges::ssize(in_buffer)`.
    - Must be representable as `streamsize`.
    - Position after the write must be less than or equal to `buffer_.max_size()`.
    - Position after the write must be representable as `streamoff`.
  - Resizes the stream buffer so it has enough space to write the chosen amount of bytes. If any exceptions are thrown during resizing of stream buffer, they are propagated outside.
  - Writes chosen amount of bytes from the given buffer to the stream and advances stream position by the amount of bytes written.

Returns: The amount of bytes written.

Throws: `io_error` in case of error.

Error conditions:

- `file_too_large` - if `!ranges::empty(in_buffer) && ((pos_ == buffer_.max_size()) || (pos_ == position::ma`

11.17.2.4 29.1.?.?.? Buffer management [output.memory.stream.buffer]

```
constexpr const Container& get_buffer() const & noexcept;
```

Returns: `buffer_`.

```
constexpr Container get_buffer() && noexcept;
```

Returns: `move(buffer_)`.

```
constexpr void set_buffer(const Container& new_buffer);
```

Postconditions:

- `buffer_ == new_buffer`.

— `pos_ == 0`.

```
constexpr void set_buffer(Container&& new_buffer);
```

Effects: Move assigns `new_buffer` to `buffer_`.

Postconditions: `pos_ == 0`.

```
constexpr void reset_buffer() noexcept;
```

Effects: Equivalent to `buffer_.clear()`.

Postconditions: `pos_ == 0`.

11.17.3 29.1.?.3 Class template `basic_input_output_memory_stream` [io.memory.stream]

```
template <typename Container>
class basic_input_output_memory_stream final
{
public:
    // Constructors
    constexpr basic_input_output_memory_stream();
    constexpr basic_input_output_memory_stream(const Container& c);
    constexpr basic_input_output_memory_stream(Container&& c);

    // Position
    constexpr position get_position() const noexcept;
    constexpr void seek_position(position pos);
    constexpr void seek_position(offset off);
    constexpr void seek_position(base_position base) noexcept;
    constexpr void seek_position(base_position base, offset off);

    // Reading
    constexpr streamsize read_some(span<byte> out_buffer);

    // Writing
    constexpr streamsize write_some(span<const byte> in_buffer);

    // Buffer management
    constexpr const Container& get_buffer() const & noexcept;
    constexpr Container get_buffer() && noexcept;
    constexpr void set_buffer(const Container& new_buffer);
    constexpr void set_buffer(Container&& new_buffer);
    constexpr void reset_buffer() noexcept;
private:
    Container buffer_; // exposition only
    typename Container::difference_type pos_; // exposition only
};
```

TODO

11.17.3.1 29.1.?.?.? Constructors [io.memory.stream.cons]

```
constexpr basic_input_output_memory_stream();
```

Postconditions:

— `buffer_ == Container{}`,

— `pos_ == 0`.

```
constexpr basic_input_output_memory_stream(const Container& c);
```

Effects: Initializes `buffer_` with `c`.

Postconditions: `pos_ == 0`.

```
constexpr basic_input_output_memory_stream(Container&& c);
```

Effects: Initializes `buffer_` with `move(c)`.

Postconditions: `pos_ == 0`.

11.17.3.2 29.1.?.?.? Position [io.memory.stream.position]

```
constexpr position get_position() const noexcept;
```

Returns: `position(pos_)`.

```
constexpr void seek_position(position pos);
```

Postconditions: `pos_ == pos.value()`.

Throws: `io_error` in case of error.

Error conditions:

— `invalid_argument` - if position is negative.
— `value_too_large` - if position cannot be represented as type `typename Container::difference_type`.

```
constexpr void seek_position(offset off);
```

Effects: Calls `seek_position(`*`move_position`*`(pos_, off))`

Throws: `io_error` in case of error.

Error conditions:

— `invalid_argument` - if resulting position is negative.
— `value_too_large` - if resulting position cannot be represented as type `streamoff` or `typename Container::difference_t`

```
constexpr void seek_position(base_position base) noexcept;
```

Effects:

— If `base == base_position::beginning`, equivalent to `pos_ = 0`.
— If `base == base_position::current`, does nothing.
— If `base == base_position::end`, equivalent to `pos_ = min(ranges::ssize(buffer_), position::max().value())`.

```
constexpr void seek_position(base_position base, offset off);
```

Effects:

— If `base == base_position::beginning`, calls `seek_position(position(off))`.
— If `base == base_position::current`, calls `seek_position(off)`.
— If `base == base_position::end`, calls `seek_position(`*`move_position`*`(ranges::ssize(buffer_), off))`.

Throws: `io_error` in case of error.

Error conditions:

— `invalid_argument` - if resulting position is negative.
— `value_too_large` - if resulting position cannot be represented as type `streamoff` or `typename Container::difference_t`

11.17.3.3 29.1.?.?.? Reading [io.memory.stream.read]

```
constexpr streamsize read_some(span<byte> out_buffer);
```

Effects:

- If `ranges::empty(out_buffer)`, returns `0`.
- If `pos_ >= ranges::ssize(buffer_)`, returns `0`.
- If `pos_ == position::max().value()`, throws exception.
- Otherwise determines the amount of bytes to read so that it satisfies the following constrains:
    - Must be less than or equal to `ranges::ssize(out_buffer)`.
    - Must be representable as `streamsize`.
    - Position after the read must be less than or equal to `ranges::ssize(buffer_)`.
    - Position after the read must be representable as `streamoff`.
- After that reads that amount of bytes from the stream to the given buffer and advances stream position by the amount of bytes read.

Returns: The amount of bytes read.

Throws: `io_error` in case of error.

Error conditions:

- `value_too_large` - if `!ranges::empty(out_buffer)` and `pos_ == position::max().value()`.

11.17.3.4 29.1.?.?.? Writing [io.memory.stream.write]

```
constexpr streamsize write_some(span<const byte> in_buffer);
```

Effects:

- If `ranges::empty(in_buffer)`, returns `0`.
- If `pos_ >= buffer_.max_size()` or `pos_ == position::max().value()`, throws exception.
- If `pos_ < ranges::ssize(buffer_)`:
    - Determines the amount of bytes to write so that it satisfies the following constrains:
        - Must be less than or equal to `ranges::ssize(in_buffer)`.
        - Must be representable as `streamsize`.
        - Position after the write must be less than or equal to `ranges::ssize(buffer_)`.
        - Position after the write must be representable as `streamoff`.
    - Writes that amount of bytes from the given buffer to the stream and advances stream position by the amount of bytes written.
- Otherwise:
    - Determines the amount of bytes to write so that it satisfies the following constrains:
        - Must be less than or equal to `ranges::ssize(in_buffer)`.
        - Must be representable as `streamsize`.
        - Position after the write must be less than or equal to `buffer_.max_size()`.
        - Position after the write must be representable as `streamoff`.
    - Resizes the stream buffer so it has enough space to write the chosen amount of bytes. If any exceptions are thrown during resizing of stream buffer, they are propagated outside.
    - Writes chosen amount of bytes from the given buffer to the stream and advances stream position by the amount of bytes written.

Returns: The amount of bytes written.

Throws: `io_error` in case of error.

Error conditions:

- `file_too_large` - if `!ranges::empty(in_buffer) && ((pos_ == buffer_.max_size()) || (pos_ == position::ma`

11.17.3.5 29.1.?.?.? Buffer management [io.memory.stream.buffer]

```
constexpr const Container& get_buffer() const & noexcept;
```

Returns: `buffer_`.

```
constexpr Container get_buffer() && noexcept;
```

Returns: `move(buffer_)`.

```
constexpr void set_buffer(const Container& new_buffer);
```

Postconditions:

- `buffer_ == new_buffer`.
- `pos_ == 0`.

```
constexpr void set_buffer(Container&& new_buffer);
```

Effects: Move assigns `new_buffer` to `buffer_`.

Postconditions: `pos_ == 0`.

```
constexpr void reset_buffer() noexcept;
```

Effects: Equivalent to `buffer_.clear()`.

Postconditions: `pos_ == 0`.

11.18 29.1.? File streams [file.streams???] (naming conflict)

11.18.1 29.1.?.? Native handles [file.streams.native]

TODO

11.18.2 29.1.?.? Class `file_stream_base` [file.stream.base]

```
class file_stream_base
{
public:
    using native_handle_type = implementation-defined;

    // Position
    position get_position() const;
    void seek_position(position pos);
    void seek_position(offset off);
    void seek_position(base_position base);
    void seek_position(base_position base, offset off);

    // Buffering
    void flush();

    // Native handle management
    native_handle_type native_handle();
    void assign(native_handle_type handle);
    native_handle_type release();
protected:
    // Construct/copy/destroy
    file_stream_base() noexcept;
    file_stream_base(const filesystem::path& file_name, mode mode, creation c);
    file_stream_base(native_handle_type handle);
    file_stream_base(const file_stream_base&) = delete;
    file_stream_base(file_stream_base&&);
    ~file_stream_base();
    file_stream_base& operator=(const file_stream_base&) = delete;
    file_stream_base& operator=(file_stream_base&&);

    input_output_span_stream buffer_; // exposition only
```

TODO

11.18.2.1 29.1.?.?.? Constructors [file.stream.base.cons]

```
file_stream_base() noexcept;
```

Effects: TODO

```
file_stream_base(const filesystem::path& file_name, mode mode, creation c);
```

Effects: TODO

Throws: TODO

```
file_stream_base(native_handle_type handle);
```

Effects: TODO

Throws: TODO

11.18.2.2 29.1.?.?.? Position [file.stream.base.position]

```
position get_position() const;
```

Returns: Current position of the stream.

Throws: TODO

```
void seek_position(position pos);
```

Effects: Calls `flush()` and then sets the position of the stream to the given value.

Throws: TODO

```
void seek_position(offset off);
```

Effects: Calls `flush()` and then seeks the position of the stream to the given offset relative to the current position of the stream.

Throws: TODO

```
void seek_position(base_position base);
```

Effects: Calls `flush()` and then seeks the position of the stream to the given base position.

Throws: TODO

```
void seek_position(base_position base, offset off);
```

Effects: Calls `flush()` and then seeks the position of the stream according to the given base position and offset.

Throws: TODO

11.18.2.3 29.1.?.?.? Buffering [file.stream.base.buffer]

```
void flush();
```

Effects: If the last operation on the stream was input, resets the internal buffer. If the last operation on the stream was output, writes the contents of the internal buffer to the file and then resets the internal buffer.

Throws: TODO

11.18.3 29.1.?.? Class `input_file_stream` [input.file.stream]

```
class input_file_stream final : public file_stream_base
{
public:
    // Construct/copy/destroy
    input_file_stream() noexcept = default;
    input_file_stream(const filesystem::path& file_name);
    input_file_stream(native_handle_type handle);

    // Reading
    streamsize read_some(span<byte> out_buffer);
};
```

TODO

11.18.3.1 29.1.?.?.? Constructors [input.file.stream.cons]

```
input_file_stream(const filesystem::path& file_name);
```

Effects: Initializes the base class with `file_stream_base(file_name, mode::read, creation::open_existing)`.

```
input_file_stream(native_handle_type handle);
```

Effects: Initializes the base class with `file_stream_base(handle)`.

11.18.3.2 29.1.?.?.? Reading [input.file.stream.read]

```
streamsize read_some(span<byte> out_buffer);
```

Effects:

- If `ranges::empty(out_buffer)`, returns `0`.
- Otherwise:
    - If the internal buffer is empty, reads zero or more bytes from the file into the internal buffer.
    - Calls `buffer_.read_some(out_buffer)`.

Returns: The amount of bytes read from the internal buffer.

Throws: TODO

11.18.4 29.1.?.? Class `output_file_stream` [output.file.stream]

```
class output_file_stream final : public file_stream_base
{
public:
    // Construct/copy/destroy
    output_file_stream() noexcept = default;
    output_file_stream(const filesystem::path& file_name,
        creation c = creation::if_needed);
    output_file_stream(native_handle_type handle);

    // Writing
    streamsize write_some(span<const byte> in_buffer);
};
```

TODO

11.18.4.1 29.1.?.?.? Constructors [output.file.stream.cons]

```
output_file_stream(const filesystem::path& file_name,
    creation c = creation::if_needed);
```

Effects: Initializes the base class with `file_stream_base(file_name, mode::write, c)`.

```
output_file_stream(native_handle_type handle);
```

Effects: Initializes the base class with `file_stream_base(handle)`.

11.18.4.2 29.1.?.?.? Writing [output.file.stream.write]

```
streamsize write_some(span<const byte> in_buffer);
```

Effects:

- If `ranges::empty(in_buffer)`, returns `0`.
- Otherwise:
  - If the internal buffer is full, calls `flush()`.
  - Calls `buffer_.write_some(in_buffer)`.

Returns: The amount of bytes written to the internal buffer.

Throws: TODO

11.18.5 29.1.?.? Class `input_output_file_stream` [io.file.stream]

```
class input_output_file_stream final : public file_stream_base
{
public:
    // Construct/copy/destroy
    input_output_file_stream() noexcept = default;
    input_output_file_stream(const filesystem::path& file_name,
        creation c = creation::if_needed);
    input_output_file_stream(native_handle_type handle);

    // Reading
    streamsize read_some(span<byte> out_buffer);

    // Writing
    streamsize write_some(span<const byte> in_buffer);
};
```

TODO

11.18.5.1 29.1.?.?.? Constructors [io.file.stream.cons]

```
input_output_file_stream(const filesystem::path& file_name,
    creation c = creation::if_needed);
```

Effects: Initializes the base class with `file_stream_base(file_name, mode::write, c)`.

```
input_output_file_stream(native_handle_type handle);
```

Effects: Initializes the base class with `file_stream_base(handle)`.

11.18.5.2 29.1.?.?.? Reading [io.file.stream.read]

```
streamsize read_some(span<byte> out_buffer);
```

Effects:

- If `ranges::empty(out_buffer)`, returns `0`.
- Otherwise:
  - If the last operation on the stream was output:
    - Calls `flush()`.
    - Reads zero or more bytes from the file to the internal buffer.
    - Calls `buffer_.read_some(out_buffer)`.
  - If the last operation on the stream was input:
    - If the internal buffer is empty, reads zero or more bytes from the file to the internal buffer.
    - Calls `buffer_.read_some(out_buffer)`.

Returns: The amount of bytes read from the internal buffer.

Throws: TODO

11.18.5.3 29.1.?.?.? Writing [io.file.stream.write]

```
streamsize write_some(span<const byte> in_buffer);
```

Effects:

- If `ranges::empty(in_buffer)`, returns `0`.
- Otherwise:
  - If the last operation on the stream was input:
    - Resets the internal buffer.
    - Calls `buffer_.write_some(in_buffer)`.
  - If the last operation on the stream was output:
    - If the internal buffer is full, calls `flush()`.
    - Calls `buffer_.write_some(in_buffer)`.

Returns: The amount of bytes written to the internal buffer.

Throws: TODO

12 References

[Boost.Serialization] Boost.Serialization.
https://www.boost.org/doc/libs/1_69_0/libs/serialization/doc/index.html

[Cereal] Cereal.
https://uscilab.github.io/cereal/index.html

[cpp-io-impl] Implementation of modern std::byte stream IO for C++.
https://github.com/Lyberta/cpp-io-impl

[N4849] Richard Smith. 2020. Working Draft, Standard for Programming Language C++.
https://wg21.link/n4849

[P1031R2] Niall Douglas. 2019. Low level file i/o library.
https://wg21.link/p1031r2

[P1272R2] Isabella Muerte. 2019. Byteswapping for fun&&nuf.
https://wg21.link/p1272r2

[P1467R4] David Olsen, Michał Dominiak. 2020. Extended floating-point types and standard names.
https://wg21.link/p1467r4

[P1468R3] Michał Dominiak, David Olsen, Boris Fomitchev, Sergei Nikolaev. 2020. Fixed-layout floating-point type aliases.
https://wg21.link/p1468r3

[P1750R1] Klemans Morgenstern, Jeff Garland, Elias Kosunen, Fatih Bakir. 2019. A Proposal to Add Process Management to the C++ Standard Library.
https://wg21.link/p1750r1