Modern std::byte stream IO for C++

Document #:	DXXXXR0
Date:	2019-11-19
Project:	Programming Language C++
Audience:	LEWGI

Reply-to:	Lyberta <lyberta@lyberta.net>

Contents

1 Abstract

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

2 Motivation

C++ has text streams for a long time. However, there is no comfortable way to read and write binary data. One can argue that it is possible to [ab]use `char`-based text streams that provide unformatted IO 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 open mode is badly designed and you'd always want to make sure to force it to have `std::ios_base::binary`.

— 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 is no agreed standard for customization points for binary IO.

This proposal tries to fix all mentioned issues.

3 Prior art

This proposal is based on ftz Serialization library which was initially written in 2010 targeting C++98 and was gradually updated to C++20. In particular, 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. [P1468R2] should fix this.

While the author thinks that having endianness and floating point convertion 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.

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

5 Design decisions

— It was chosen to put all new types into separate namespace `std::io`. This follows the model ranges took where they define more modern versions of old facilities inside a new namespace.
— The inheritance heirarchy of legacy text streams has been transformed to concepts that use more flat composition of features than inheritance tree. Legacy base class templates have been loosely transformed into the following concepts:
  — `std::basic_istream` -> `std::io::input_stream`.
  — `std::basic_ostream` -> `std::io::output_stream`.
  — Seeking functionality has been moved to `std::io::seekable_stream`.
— Concrete class templates have been transformed as follows:
  — `std::basic_istringstream` -> `std::io::basic_input_memory_stream`.
  — `std::basic_ostringstream` -> `std::io::basic_output_memory_stream`.
  — `std::basic_stringstream` -> `std::io::basic_memory_stream`.
  — `std::basic_ifstream` -> `std::io::input_file_stream`.
  — `std::basic_ofstream` -> `std::io::output_file_stream`.
  — `std::basic_fstream` -> `std::io::file_stream`.
— The `streambuf` part of legacy text streams has been dropped.
— Fixed size streams have been added:
  — `std::io::input_span_stream`.
  — `std::io::output_span_stream`.
  — `std::io::span_stream`.
— Since the explicit goal of this proposal is to do IO in terms of `std::byte`, `CharT` and `Traits` template parameters have been removed.
— All text formatting flags have been removed. A new class `std::io::format` has been introduced for binary format. The format is no longer a part of stream classes but is constructed on demand during [de]serialization as part of IO context.
— Parts of legacy text streams related to `std::ios_base::iostate` have been removed. It is better to report any specific errors via exceptions and since binary files usually have fixed layout and almost always start chunks of data with size, any kind of IO error is usually unrecoverable.
— `std::ios_base::openmode` has been split into `std::io::mode` and `std::io::creation` that are modeled after the ones from [P1031R2].
— Since there is no more buffering (as of this revision) because of lack of `streambuf` and operating systems only expose a single file position that is used both for reading and writing, the interface has been changed accordingly:
  — `tellg` and `tellp` -> `get_position`.
  — Single argument versions of `seekg` and `seekp` -> `set_position`.
  — Double argument versions of `seekg` and `seekp` -> `seek_position`.
  — `peek`, `putback`, `unget` and `flush` member functions were removed.
— `std::basic_ios::pos_type` has been replaced with `std::streamoff`.
— `std::basic_ios::off_type` has been replaced with `std::streamoff`.
— `std::ios_base::seekdir` has been replaced with `std::io::base_position`.
— `getline` and `ignore` member functions were removed because they don't make sense during binary IO.
— Since it is not always possible to read or write all requested bytes in one system call (especially during networking), the interface has been changed accordingly:
  — `std::io::input_stream` requires `read_some` member function that reads zero or more bytes from the stream and returns amount of bytes read.
  — `std::io::output_stream` requires `write_some` member function that writes one or more bytes to the stream and returns amount of bytes written.
  — `gcount` became the return value of `read_some`.

  - `get`, `read`, `put` and `write` member functions have been replaced with `std::io::read_raw` and `std::io::write_raw` customization points.
- `operator>>` and `operator<<` have been replaced with `std::io::read` and `std::io::write` customization points.

6 Tutorial

6.1 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";
    }
}
```

6.2 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`.

6.3 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.
```

```cpp
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
```

6.4 Example 4: Working with floating point numbers

TODO

6.5 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.

```cpp
#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';
}
```

6.6 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';
}
```

6.7 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.

```
#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';
}
```

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

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

```
#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';
```

```
}
```

6.9 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.

```
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 /* ... */
        }
    }
}
```

6.10 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>
```

```cpp
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>
    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::base_position::current,
                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;
    }
};

// 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
{
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::streamoff end_position = stream.get_position() + 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);
```

```
        }
    }
private:
    std::endian m_endianness;
    ChunkID m_form_type;
    std::vector<Chunk> m_chunks;
}

}
```

TODO: More tutorials? More explanations.

7 Implementation experience

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

Most of the proposal can be implemented in ISO C++. Low level conversions inside `std::io::read` and `std::io::write` require knowledge of implementation defined format of integers and floating point numbers. 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`
`set_position`	`lseek`	`SetFilePointerEx`	`EFI_FILE_PROTOCOL.SetPosition`
`seek_position`	`lseek`	`SetFilePointerEx`	No 1:1 mapping
`read_some`	`read`	`ReadFile`	`EFI_FILE_PROTOCOL.Read`
`write_some`	`write`	`WriteFile`	`EFI_FILE_PROTOCOL.Write`

8 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 doesn't rule out more low-level library that exposes complex details of modern operating systems. However, the design of this library has been intentionally kept as simple as possible to be novice-friendly.

9 Open issues

— Error handling using `throws` + `std::error`.
— `std::filesystem::path_view`
— Remove `std::io::floating_point_format` if [P1468R2] is accepted.
— Buffering for file IO.
— Binary versions of `std::cin`, `std::cout` and `std::cerr`.
— Vectored IO.
— `constexpr` file streams as a generalization of `std::embed`.

10 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”.

10.1 29.1.? General [io.general]

TODO

10.2 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;

// 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;

enum class base_position
{
    beginning,
    current,
    end
};
```

```
template <typename T>
concept seekable_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;

// Span streams
class input_span_stream;
class output_span_stream;
class 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_memory_stream;

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

// File streams
```

```
enum class mode
{
    read,
    write
};

enum class creation
{
    open_existing,
    if_needed,
    truncate_existing
};

class file_stream_base;
class input_file_stream;
class output_file_stream;
class file_stream;

}
}
```

10.3 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())`.

10.4 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

10.5 29.1.? Stream concepts [stream.concepts]

10.5.1 29.1.?.? Concept `input_stream` [stream.concept.input]

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

TODO

10.5.1.1 29.1.?.?.? Reading [input.stream.read]

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

Effects: If `empty(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.

10.5.2 29.1.?.? Concept `output_stream` [stream.concept.output]

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

TODO

10.5.2.1 29.1.?.?.? Writing [output.stream.write]

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

Effects: If `empty(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.

10.5.2.2 29.1.?.? Concept `stream` [stream.concept.stream]

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

TODO

10.5.3 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<streamoff>;
    } && requires(T s, streamoff position, base_position base)
    {
        s.set_position(position);
        s.seek_position(base, position);
    };
```

TODO

10.5.3.1 29.1.?.?.? Position [seekable.stream.position]

```
streamoff get_position();
```

Returns: Current position of the stream.

```
void set_position(streamoff position);
```

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(base_position base, streamoff offset);
```

Effects: TODO

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.

10.6 29.1.? Customization points for unformatted IO [io.raw]

10.6.1 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`.

10.6.2 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.

10.7 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

10.7.1 29.1.?.? Constructor [io.format.cons]

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

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

10.7.2 29.1.?.? Member functions [io.format.members]

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

Returns: `endianness_`.

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

Ensures: `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;
```

Ensures: `float_format_ == new_format`.

10.8 29.1.? Context concepts [io.context.concepts]

10.8.1 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

10.8.2 29.1.?.? Concept `input_context` [input.context]

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

TODO

10.8.3 29.1.?.? Concept `output_context` [output.context]

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

TODO

10.9 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

10.9.1 29.1.?.? Constructor [io.default.context.cons]

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

Effects: Initializes `stream_` with `s`.

Ensures: `format_ == f`.

10.9.2 29.1.?.? Stream [io.default.context.stream]

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

Returns: `stream_`.

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

Returns: `stream_`.

10.9.3 29.1.?.? Format [io.default.context.format]

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

Returns: `format_`.

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

Ensures: `format_ == f`.

10.10 29.1.? Customization points for serialization [io.serialization]

10.10.1 29.1.?.? Helper concepts

```
template <typename T, typename I, typename... Args>
concept customly-readable-from =
    (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 =
    (output_stream<O> || output_context<O>) &&
    requires(const T object, O& o, Args&&... args)
    {
```

```
        object.write(o, forward<Args>(args)...);
    };
```

10.10.2 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`.

10.10.3 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(O.get_stream(), E)`.
  - 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.

10.11 29.1.? Serialization concepts [serialization.concepts]

10.11.1 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

10.11.2 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

10.12 29.1.? Span streams [span.streams]

10.12.1 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 streamoff get_position() const noexcept;
    constexpr void set_position(streamoff position);
    constexpr void seek_position(base_position base, streamoff offset);

    // Reading
    constexpr streamsize read_some(span<byte> 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 position_; // exposition only
};
```

TODO

10.12.1.1 29.1.?.?.? Constructors [input.span.stream.cons]

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

Ensures:

- `empty(buffer_) == true`,
- `position_ == 0`.

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

Ensures:

- `data(buffer_) == data(buffer)`,
- `size(buffer_) == size(buffer)`,
- `position_ == 0`.

10.12.1.2 29.1.?.?.? Position [input.span.stream.position]

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

Returns: `position_`.

```
constexpr void set_position(streamoff position);
```

Ensures: `position_ == position`.

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(base_position base, streamoff offset);
```

Effects: TODO

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`.

10.12.1.3 29.1.?.?.? Reading [input.span.stream.read]

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

Effects: If `empty(buffer)`, returns `0`. If `position_ >= ssize(buffer_)`, returns `0`. If `position_ == numeric_limits<stream` throws exception. Otherwise determines the amount of bytes to read so that it satisfies the following constrains:

- Must be less than or equal to `ssize(buffer)`.
- Must be representable as `streamsize`.
- Position after the read must be less than or equal to `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 !empty(buffer) and position_ == numeric_limits<streamoff>::max().

10.12.1.4 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;
```

Ensures:

— data(buffer_) == data(new_buffer),
— size(buffer_) == size(new_buffer),
— position_ == 0.

10.12.2 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 streamoff get_position() const noexcept;
    constexpr void set_position(streamoff position);
    constexpr void seek_position(base_position base, streamoff offset);

    // Writing
    constexpr streamsize write_some(span<const byte> 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 position_; // exposition only
};
```

TODO

10.12.2.1 29.1.?.?.? Constructors [output.span.stream.cons]

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

Ensures:

— empty(buffer_) == true,
— position_ == 0.

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

Ensures:

— data(buffer_) == data(buffer),
— size(buffer_) == size(buffer),

— `position_ == 0`.

10.12.2.2 29.1.?.?.? Position [output.span.stream.position]

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

Returns: `position_`.

```
constexpr void set_position(streamoff position);
```

Ensures: `position_ == position`.

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(base_position base, streamoff offset);
```

Effects: TODO

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`.

10.12.2.3 29.1.?.?.? Writing [output.span.stream.write]

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

Effects: If `empty(buffer)`, returns `0`. If `position_ >= ssize(buffer_)` or `position_ == numeric_limits<streamoff>::max` throws exception. Otherwise determines the amount of bytes to write so that it satisfies the following constrains:

— Must be less than or equal to `ssize(buffer)`.
— Must be representable as `streamsize`.
— Position after the write must be less than or equal to `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 `!empty(buffer) && ((position_ == ssize(buffer_)) || (position_ == numeric_limits<st`

10.12.2.4 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;
```

Ensures:

— `data(buffer_) == data(new_buffer)`,
— `size(buffer_) == size(new_buffer)`,

— position_ == 0.

10.12.3 29.1.?.3 Class span_stream [span.stream]

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

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

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

    // Writing
    constexpr streamsize write_some(span<const byte> 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 position_; // exposition only
};
```

TODO

10.12.3.1 29.1.?.?.? Constructors [span.stream.cons]

```
constexpr span_stream() noexcept;
```

Ensures:

— empty(buffer_) == true,
— position_ == 0.

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

Ensures:

— data(buffer_) == data(buffer),
— size(buffer_) == size(buffer),
— position_ == 0.

10.12.3.2 29.1.?.?.? Position [output.span.stream.position]

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

Returns: position_.

```
constexpr void set_position(streamoff position);
```

Ensures: position_ == position.

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(base_position base, streamoff offset);
```

Effects: TODO

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`.

10.12.3.3 29.1.?.?.? Reading [span.stream.read]

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

Effects: If `empty(buffer)`, returns `0`. If `position_ >= ssize(buffer_)`, returns `0`. If `position_ == numeric_limits<stream` throws exception. Otherwise determines the amount of bytes to read so that it satisfies the following constrains:

— Must be less than or equal to `ssize(buffer)`.
— Must be representable as `streamsize`.
— Position after the read must be less than or equal to `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 `!empty(buffer)` and `position_ == numeric_limits<streamoff>::max()`.

10.12.3.4 29.1.?.?.? Writing [span.stream.write]

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

Effects: If `empty(buffer)`, returns `0`. If `position_ >= ssize(buffer_)` or `position_ == numeric_limits<streamoff>::max` throws exception. Otherwise determines the amount of bytes to write so that it satisfies the following constrains:

— Must be less than or equal to `ssize(buffer)`.
— Must be representable as `streamsize`.
— Position after the write must be less than or equal to `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 `!empty(buffer) && ((position_ == ssize(buffer_)) || (position_ == numeric_limits<st`

10.12.3.5 29.1.?.?.? Buffer management [span.stream.buffer]

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

Returns: buffer_.

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

Ensures:

- data(buffer_) == data(new_buffer),
- size(buffer_) == size(new_buffer),
- position_ == 0.

10.13 29.1.? Memory streams [memory.streams]

10.13.1 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 streamoff get_position() const noexcept;
    constexpr void set_position(streamoff position);
    constexpr void seek_position(base_position base, streamoff offset);

    // Reading
    constexpr streamsize read_some(span<byte> 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 position_; // exposition only
};
```

TODO

10.13.1.1 29.1.?.?.? Constructors [input.memory.stream.cons]

```
constexpr basic_input_memory_stream();
```

Ensures:

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

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

Effects: Initializes `buffer_` with `c`.

Ensures: `position_ == 0`.

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

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

Ensures: `position_ == 0`.

10.13.1.2 29.1.?.?.? Position [input.memory.stream.position]

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

Returns: `position_`.

```
constexpr void set_position(streamoff position);
```

Ensures: `position_ == position`.

Throws: `io_error` in case of error.

Error conditions:

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

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

Effects: TODO

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`

10.13.1.3 29.1.?.?.? Reading [input.memory.stream.read]

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

Effects: If `empty(buffer)`, returns `0`. If `position_ >= ssize(buffer_)`, returns `0`. If `position_ == numeric_limits<stream` throws exception. Otherwise determines the amount of bytes to read so that it satisfies the following constrains:

— Must be less than or equal to `ssize(buffer)`.
— Must be representable as `streamsize`.
— Position after the read must be less than or equal to `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 `!empty(buffer)` and `position_ == numeric_limits<streamoff>::max()`.

10.13.1.4 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);
```

Ensures:

— `buffer_ == new_buffer`.
— `position_ == 0`.

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

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

Ensures: `position_ == 0`.

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

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

Ensures: `position_ == 0`.

10.13.2 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 streamoff get_position() const noexcept;
    constexpr void set_position(streamoff position);
    constexpr void seek_position(base_position base, streamoff offset);

    // Writing
    constexpr streamsize write_some(span<const byte> 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 position_; // exposition only
};
```

TODO

10.13.2.1 29.1.?.?.? Constructors [output.memory.stream.cons]

```
constexpr basic_output_memory_stream();
```

Ensures:

— `buffer_ == Container{}`,
— `position_ == 0`.

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

Effects: Initializes `buffer_` with `c`.

Ensures: `position_ == 0`.

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

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

Ensures: `position_ == 0`.

10.13.2.2 29.1.?.?.? Position [output.memory.stream.position]

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

Returns: `position_`.

```
constexpr void set_position(streamoff position);
```

Ensures: `position_ == position`.

Throws: `io_error` in case of error.

Error conditions:

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

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

Effects: TODO

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`

10.13.2.3 29.1.?.?.? Writing [output.memory.stream.write]

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

Effects: If `empty(buffer)`, returns `0`. If `position_ >= buffer_.max_size()` or `position_ == numeric_limits<streamoff>:` throws exception. If `position_ < ssize(buffer_)`:

— Determines the amount of bytes to write so that it satisfies the following constrains:
  — Must be less than or equal to `ssize(buffer)`.
  — Must be representable as `streamsize`.
  — Position after the write must be less than or equal to `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 `ssize(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 `!empty(buffer) && ((position_ == buffer_.max_size()) || (position_ == numeric_limit`

10.13.2.4 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);
```

Ensures:

— `buffer_ == new_buffer`.
— `position_ == 0`.

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

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

Ensures: `position_ == 0`.

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

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

Ensures: `position_ == 0`.

10.13.3 29.1.?.3 Class template `basic_memory_stream` [memory.stream]

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

```
    // Position
    constexpr streamoff get_position() const noexcept;
    constexpr void set_position(streamoff position);
    constexpr void seek_position(base_position base, streamoff offset);

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

    // Writing
    constexpr streamsize write_some(span<const byte> 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 position_; // exposition only
};
```

TODO

10.13.3.1 29.1.?.?.? Constructors [memory.stream.cons]

```
constexpr basic_memory_stream();
```

Ensures:

— `buffer_ == Container{}`,
— `position_ == 0`.

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

Effects: Initializes `buffer_` with `c`.

Ensures: `position_ == 0`.

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

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

Ensures: `position_ == 0`.

10.13.3.2 29.1.?.?.? Position [memory.stream.position]

```
constexpr streamoff get_position();
```

Returns: `position_`.

```
constexpr void set_position(streamoff position);
```

Ensures: `position_ == position`.

Throws: `io_error` in case of error.

Error conditions:

— `invalid_argument` - if position is negative.

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

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

Effects: TODO

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`

10.13.3.3 29.1.?.?.? Reading [memory.stream.read]

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

Effects: If `empty(buffer)`, returns `0`. If `position_ >= ssize(buffer_)`, returns `0`. If `position_ == numeric_limits<stream` throws exception. Otherwise determines the amount of bytes to read so that it satisfies the following constrains:

— Must be less than or equal to `ssize(buffer)`.
— Must be representable as `streamsize`.
— Position after the read must be less than or equal to `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 `!empty(buffer)` and `position_ == numeric_limits<streamoff>::max()`.

10.13.3.4 29.1.?.?.? Writing [memory.stream.write]

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

Effects: If `empty(buffer)`, returns `0`. If `position_ >= buffer_.max_size()` or `position_ == numeric_limits<streamoff>:` throws exception. If `position_ < ssize(buffer_)`:

— Determines the amount of bytes to write so that it satisfies the following constrains:
  — Must be less than or equal to `ssize(buffer)`.
  — Must be representable as `streamsize`.
  — Position after the write must be less than or equal to `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 `ssize(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 `!empty(buffer) && ((position_ == buffer_.max_size()) || (position_ == numeric_limit`

10.13.3.5 29.1.?.?.? Buffer management [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);
```

Ensures:

— `buffer_ == new_buffer`.
— `position_ == 0`.

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

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

Ensures: `position_ == 0`.

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

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

Ensures: `position_ == 0`.

10.14 29.1.? File streams [file.streams???] (naming conflict)

10.14.1 29.1.?.? Native handles [file.streams.native]

TODO

10.14.2 29.1.?.? Class `file_stream_base` [file.stream.base]

```
class file_stream_base
{
public:
    using native_handle_type = implementation-defined;

    // Position
    streamoff get_position() const;
    void set_position(streamoff position);
    void seek_position(base_position base, streamoff offset);

    // 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&&);
```

TODO

10.14.2.1 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

10.14.2.2 29.1.?.?.? Position [file.stream.base.position]

```
streamoff get_position() const;
```

Returns: Current position of the stream.

Throws: TODO

```
void set_position(streamoff position);
```

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

Throws: TODO

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

Effects: TODO

Throws: TODO

10.14.3 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> buffer);
};
```

TODO

10.14.3.1 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)`.

10.14.3.2 29.1.?.?.? Reading [input.file.stream.read]

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

Effects: If `empty(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: TODO

10.14.4 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> buffer);
};
```

TODO

10.14.4.1 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)`.

10.14.4.2 29.1.?.?.? Writing [output.file.stream.write]

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

Effects: If `empty(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: TODO

10.14.5 29.1.?.? Class `file_stream` [file.stream]

```
class file_stream final : public file_stream_base
{
public:
    // Construct/copy/destroy
    file_stream() noexcept = default;
    file_stream(const filesystem::path& file_name,
        creation c = creation::if_needed);
    file_stream(native_handle_type handle);

    // Reading
    streamsize read_some(span<byte> buffer);

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

TODO

10.14.5.1 29.1.?.?.? Constructors [file.stream.cons]

```
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)`.

```
file_stream(native_handle_type handle);
```

Effects: Initializes the base class with `file_stream_base(handle)`.

10.14.5.2 29.1.?.?.? Reading [file.stream.read]

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

Effects: If `empty(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: TODO

10.14.5.3 29.1.?.?.? Writing [file.stream.write]

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

Effects: If `empty(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: TODO

11 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

[P1468R2] Michał Dominiak, David Olsen, Boris Fomitchev, Sergei Nikolaev. 2019. Fixed-layout floating-point type aliases.
https://wg21.link/p1468r2