std::cout is out

Why iostreams must go

I/O in the standard

C-style I/O

```
    std::FILE* fp = std::fopen(strPath, "wb");
```

C-style I/O

```
std::FILE* fp = std::fopen(strPath, "wb");
if(!fp) { /* error handling, check errno for error message */ }
```

C-style I/O

```
std::FILE* fp = std::fopen(strPath, "wb");
if(!fp) { /* error handling, check errno for error message */ }

unsigned int n = ...;
std::string str = ...;
if(std::fprintf(fp, "Dec: %d Hex: %x String: %s\n", n, n, str.c_str())<0) {
  // error handling
}
```

C++ Streams

C++ Streams

```
    std::basic_ofstream<char> ofs(strPath, std::ios_base::binary | std::ios_base::out);
```

C++ Streams

```
std::basic_ofstream<char> ofs(strPath, std::ios_base::binary | std::ios_base::out);
if(!ofs) { /*error handling*/ }
```

C++ Streams

```cpp
std::basic_ofstream<char> ofs(strPath, std::ios_base::binary | std::ios_base::out);
if(!ofs) { /*error handling*/ }

unsigned int n = ...;
std::string str = ...;

ofs << "Dec: " << n
    << " Hex: " << std::hex << n
    << " String: " << str << std::endl;
```

C++ Streams

```
std::basic_ofstream<char> ofs(strPath, std::ios_base::binary | std::ios_base::out);
if(!ofs) { /*error handling*/ }

unsigned int n = ...;
std::string str = ...;

ofs << "Dec: " << n
    << " Hex: " << std::hex << n << std::unsetf(std::ios_base::hex)
    << " String: " << str << "\n";
```

C++ Streams

```
std::basic_ofstream<char> ofs(strPath, std::ios_base::binary | std::ios_base::out);
if(!ofs) { /*error handling*/ }

unsigned int n = ...;
std::string str = ...;

ofs << "Dec: " << n
    << " Hex: " << std::hex << n << std::unsetf(std::ios_base::hex)
    << " String: " << str << "\n";
if(ofs.fail() || ofs.bad()) {
  // error handling
}
```

So far, so bad

So far, so bad

How do you create a temporary file on Windows?

So far, so bad

How do you create a temporary file on Windows?

```
HANDLE h = CreateFile(
      szFile,
      GENERIC_READ|GENERIC_WRITE,
      FILE_SHARE_READ,
      nullptr, // security descriptor, handle inheritance
      CREATE_NEW,
      FILE_ATTRIBUTE_TEMPORARY, // attributes incl e.g. encryption, hidden flag
      nullptr // template file
);
```

What would we like to have?

What would we like to have?

Better Files

- Platform-independent files, creatable from native handles
- Use type-system to document file capabilities
- With sensible error handling

What would we like to have?

Better Files

- Platform-independent files, creatable from native handles
- Use type-system to document file capabilities
- With sensible error handling

Better Formatting

- Stateless formatting, extensible, usable for different types (Write to `std::string`, `std::vector<char>`, files, `IStream*`!)
- Distinction between binary and text output
- Automatic encoding conversion UTF8 - UTF16

Better Files

Type Safety

Type Safety

```
struct unmanaged_filebase_t {
  // ~unmanaged_filebase_t does not close file, use for stdin etc
  std::FILE* m_file;
};

struct filebase_t : unmanaged_filebase_t {
  ~filebase_t(); // closes file
};
```

Type Safety

```
struct unmanaged_filebase_t {
  // ~unmanaged_filebase_t does not close file, use for stdin etc
  std::FILE* m_file;
};

struct filebase_t : unmanaged_filebase_t {
  ~filebase_t(); // closes file
};

template<typename base>
struct readfile_impl_t : base { /* read operations */ };

template<typename base>
struct appendfile_impl_t : base { /* write operations */ };
```

Type Safety

```
// Append-only, non-owned file, e.g. for writing to stdout
using unmanaged_appendfile = appendfile_impl_t<unmanaged_filebase_t>;

// Append-only, owned file
struct appendfile : appendfile_impl_t<filebase_t> { /* ... */ };

struct readwritefile = readfile_impl_t<appendfile_impl_t<filebase_t>> {
  // Inherits read, write operations on single file handle
};
```

File Creation: POSIX

File Creation: POSIX

```
appendfile(filechar const*  szFile, create_always_tag)
    throw(std::ios_base::failure)
{
  int fd = open(szFile, O_WRONLY | O_APPEND | O_TRUNC | ..., ...); // may fail
  if(-1!=fd) throw std::ios_base::failure("");

  // File handle conversion assumed not to fail
  std::FILE* file = VERIFY(fdopen(fd, "a"));
  ...
}
```

File Creation: Windows

```
appendfile(filechar const*  szFile, create_always_tag)
    throw(std::ios_base::failure)
{
  HANDLE hfile = CreateFile(...); // may fail, error handling omitted
  if(INVALID_HANDLE_VALUE!=hfile) throw std::ios_base::failure("");

  // File handle conversion assumed not to fail
  int fd = _open_osfhandle(reinterpret_cast<intptr_t>(hfile), _O_WRONLY);
  _ASSERT(-1 != fd);
  std::FILE* file = VERIFY(_fdopen(fd, "wb"))
  ...
}
```

Error handling

Error handling

```
template<typename base>
struct readfile_impl_t : base {
  // throws in case of error only:
  std::size_t try_read(void* pv, std::size_t cb) throw(std::ios_base::failure);

  // throws in case of error or if less than cb bytes read:
  void read(void* pv, std::size_t cb) throw(std::ios_base::failure);

  template<typename T>
  T read() throw(std::ios_base::failure);
};
```

Error handling

Reading in chunks e.g. for parsing:

```
auto cbRead = file.try_read(achBuffer, sizeof(achBuffer)); // may return < sizeof(achBuffer)
```

Reading structured data

```
int nLength = file.read<int>(); // must read sizeof(int)
// Now read nLength characters
```

Better Formatting

Better Formatting

```
// Stateless formatting, into arbitrary types, with encoding conversion:
std::basic_string<char16_t> str;
str << "Dec: " << as_dec(1000) << " Hex: " << as_hex(1000);
```

Better Formatting

```
// Stateless formatting, into arbitrary types, with encoding conversion:
std::basic_string<char16_t> str;
str << "Dec: " << as_dec(1000) << " Hex: " << as_hex(1000);

// We need that too (unfortunately):
std::vector<char, some_custom_allocator> vecch;
vecch << "Dec: " << as_dec(1000) << " Hex: " << as_hex(1000);
```

Better Formatting

```
// Stateless formatting, into arbitrary types, with encoding conversion:
std::basic_string<char16_t> str;
str << "Dec: " << as_dec(1000) << " Hex: " << as_hex(1000);

// We need that too (unfortunately):
std::vector<char, some_custom_allocator> vecch;
vecch << "Dec: " << as_dec(1000) << " Hex: " << as_hex(1000);

// And this of course:
appendfile file(...);
make_typed_stream<char16_t>(file) // write UTF16 to file
  << "Dec: " << as_dec(1000) << " Hex: " << as_hex(1000);
```

Better Formatting

```
// Stateless formatting, into arbitrary types, with encoding conversion:
std::basic_string<char16_t> str;
str << "Dec: " << as_dec(1000) << " Hex: " << as_hex(1000);

// We need that too (unfortunately):
std::vector<char, some_custom_allocator> vecch;
vecch << "Dec: " << as_dec(1000) << " Hex: " << as_hex(1000);

// And this of course:
appendfile file(...);
make_typed_stream<char16_t>(file) // write UTF16 to file
  << "Dec: " << as_dec(1000) << " Hex: " << as_hex(1000);

// Let's get crazy
IStream* iStream = ...;
make_typed_stream<char>(iStream)  // write UTF8 to iStream
  << "Dec: " << as_dec(1000) << " Hex: " << as_hex(1000);
```

Better Formatting

```
// Stateless formatting, into arbitrary types, with encoding conversion:
std::basic_string<char16_t> str;
str << "Dec: " << as_dec(1000) << " Hex: " << as_hex(1000);

// We need that too (unfortunately):
std::vector<char, some_custom_allocator> vecch;
vecch << "Dec: " << as_dec(1000) << " Hex: " << as_hex(1000);

// And this of course:
appendfile file(...);
make_typed_stream<char16_t>(file) // write UTF16 to file
  << "Dec: " << as_dec(1000) << " Hex: " << as_hex(1000);

// Let's get crazy
IStream* iStream = ...;
make_typed_stream<char>(iStream)  // write UTF8 to iStream
  << "Dec: " << as_dec(1000) << " Hex: " << as_hex(1000);
```

10 lines of code needed to adapt `IStream*` to formatting library.

Make Everything Streamable

Make Everything Streamable

How to pipe characters into anything?

Make Everything Streamable

How to pipe characters into anything?

```
template<typename Sink>
typename std::enable_if<
   is_streamable<Sink>::value,
  Sink&
>::type
operator<<( Sink& os, stream_traits<Sink>::value_type ch) {
   stream_traits<Sink>::append(os, ch);
  return os;
}
```

Make Everything Streamable

How to pipe characters into anything?

```
template<typename Sink>
typename std::enable_if<
   is_streamable<Sink>::value,
  Sink&
>::type
operator<<( Sink& os, stream_traits<Sink>::value_type ch) {
   stream_traits<Sink>::append(os, ch);
  return os;
}
```

We need traits of course!

Type Traits the Classic Way

Type Traits the Classic Way

```
template<typename T>
struct stream_traits { // fall-back implementation
  using value_type = void; // with non-sensical value_type
};

template<typename Char>
struct stream_traits<std::vector<Char>> { // specialization for std::vector
  using value_type = Char; // provide needed typedefs and methods (omitted)
};
```

Type Traits the Classic Way

```
template<typename T>
struct stream_traits { // fall-back implementation
  using value_type = void; // with non-sensical value_type
};

template<typename Char>
struct stream_traits<std::vector<Char>> { // specialization for std::vector
  using value_type = Char; // provide needed typedefs and methods (omitted)
};

template<typename T> // Define is_streamable<T> by looking up stream_traits<T>
using is_streamable = std::integral_constant<
  bool,
   !std::is_same<void, typename stream_traits<T>::value_type>::value
>;
```

Type Traits using Function Lookup

```
struct not_streamable {};
not_streamable tc_stream_traits(...); // Fallback implemenation, always least preferred overload

template<typename Char>;
struct vector_stream_traits {
  using value_type = Char; // Again, required trait methods are omitted
};
template<typename Char> // Overload tc_stream_traits for std::vector<Char>
vector_stream_traits<Char> tc_stream_traits(std::vector<Char>&);
```

Type Traits using Function Lookup

```
struct not_streamable {};
not_streamable tc_stream_traits(...); // Fallback implemenation, always least preferred overload

template<typename Char>;
struct vector_stream_traits {
  using value_type = Char; // Again, required trait methods are omitted
};
template<typename Char> // Overload tc_stream_traits for std::vector<Char>
vector_stream_traits<Char> tc_stream_traits(std::vector<Char>&);

template<typename T>
using stream_traits = // Lookup return type of tc_stream_traits(T&) (with ADL and overloads!)
  decltype( tc_stream_traits( std::declval<std::decay_t<T>&>() ) );
```

Type Traits using Function Lookup

```
struct not_streamable {};
not_streamable tc_stream_traits(...); // Fallback implemenation, always least preferred overload

template<typename Char>;
struct vector_stream_traits {
  using value_type = Char; // Again, required trait methods are omitted
};
template<typename Char> // Overload tc_stream_traits for std::vector<Char>
vector_stream_traits<Char> tc_stream_traits(std::vector<Char>&);

template<typename T>
using stream_traits = // Lookup return type of tc_stream_traits(T&) (with ADL and overloads!)
  decltype( tc_stream_traits( std::declval<std::decay_t<T>&>() ) );

template<typename T> // Again, is_streamable<T> is defined via stream_traits<T>
using is_streamable = std::integral_constant<
      bool,
      !std::is_same<not_streamable, stream_traits<T>>::value
    >;
```

A Complete Traits Implementation

```
struct istream_stream_traits {
  using value_type = unsigned char; // binary device, like a file

  // Write a character
  static void append(ISequentialStream* iStream, unsigned char ch) noexcept {
    HRERR(iStream->Write(std::addressof(ch), 1, nullptr));
  }

  // Write a range of characters
  static void append_range(ISequentialStream* iStream, Rng const& rng) noexcept {
    HRERR(iStream->Write(boost::begin(rng), size(rng), nullptr));
  }

  // Flush device
  static void flush(ISequentialStream* iStream) noexcept {}
};
istream_stream_traits tc_stream_traits(ISequentialStream*);
```

`IStream*` is convertible to `ISequentialStream*`

Converting Encodings

```
template<typename Sink, typename Rng>
typename std::enable_if<
   is_streamable<Sink>::value && is_char_range<Rng>::value,
  Sink&
>::type
operator<<( Sink& os, Rng&& rng) {
  ConvertEnc<char_t of Rng, char_t of Sink>::Append(os, std::forward<Rng>(rng));
  return os;
}
```

Converting Encodings

```
template<typename Sink, typename Rng>
typename std::enable_if<
    is_streamable<Sink>::value && is_char_range<Rng>::value,
  Sink&
>::type
operator<<( Sink& os, Rng&& rng) {
  ConvertEnc<char_t of Rng, char_t of Sink>::Append(os, std::forward<Rng>(rng));
  return os;
}
```

- `ConvertEnc<Char, Char>::Append(os, rng)` simply calls `stream_traits<Sink>::append_range`
- Specializations `ConvertEnc<char, char16_t>::Append` and `ConvertEnc<char16_t, char>::Append` convert encodings
- Compile-time error when no suitable conversion exists, e.g. `ConvertEnc<char16_t, unsigned char>`

Converting Encodings

Convert UTF-8 to UTF-16:

```
std::basic_string<char16_t> str;
str << "Hello World\n"; // UTF8 to UTF16
```

Converting Encodings

Convert UTF-8 to UTF-16:

```
std::basic_string<char16_t> str;
str << "Hello World\n"; // UTF8 to UTF16
```

Files are binary and have no encoding. This does not compile:

```
appendfile file = ...;
file << "Hello World\n";
```

Converting Encodings

Convert UTF-8 to UTF-16:

```
std::basic_string<char16_t> str;
str << "Hello World\n"; // UTF8 to UTF16
```

Files are binary and have no encoding. This does not compile:

```
appendfile file = ...;
file << "Hello World\n";
```

Declare file to accept UTF-8 encoding:

```
appendfile file = ...;
make_typed_stream<char>(file) << "Hello World\n";
```

Streaming Custom Types

Streaming Custom Types

Overload global `operator<<` for custom types:

```
template<typename Sink>
typename std::enable_if<
   is_streamable<Sink>::value,
  Sink&
>::type
operator<<( Sink& os,boost::uuids::uuid const& uuid) {
   os << boost::uuids::to_string(uuid);
  return os;
}
```

Formatting Numbers

```
template<T>
struct as_dec_impl {
  T m_f;
  ...
  // Decimal separator, precision, padding etc

  template<typename Sink>
  friend typename std::enable_if<
     is_streamable<Sink>::value,
    Sink&
  >::type
  operator<<( Sink& os, as_dec_impl<T,Char> const& n) {
         // Convert m_f to characters and stream those to os
        return os;
  }
};
```

Formatting Numbers

```
template<T>
struct as_dec_impl {
  T m_f;
  ...
  // Decimal separator, precision, padding etc

  template<typename Sink>
  friend typename std::enable_if<
     is_streamable<Sink>::value,
    Sink&
  >::type
  operator<<( Sink& os, as_dec_impl<T,Char> const& n) {
         // Convert m_f to characters and stream those to os
        return os;
  }
};
```

Formatters we have:

- `as_dec(FloatingPointType f, CharType chDecimal)`
- `as_padded_lc_hex(IntegralType const& t)`

Thank You!

```
// Stateless formatting, into arbitrary types, with encoding conversion:
std::basic_string<char16_t> str;
str << "Dec: " << as_dec(1000) << " Hex: " << as_hex(1000);

// We need that too (unfortunately):
std::vector<char, some_custom_allocator> vecch;
vecch << "Dec: " << as_dec(1000) << " Hex: " << as_hex(1000);

// And this of course:
appendfile file(...);
make_typed_stream<char16_t>(file) // write UTF16 to file
  << "Dec: " << as_dec(1000) << " Hex: " << as_hex(1000);

// Let's get crazy
IStream* iStream = ...;
make_typed_stream<char>(iStream)  // write UTF8 to iStream
  << "Dec: " << as_dec(1000) << " Hex: " << as_hex(1000);
```

Slides & Benchmark

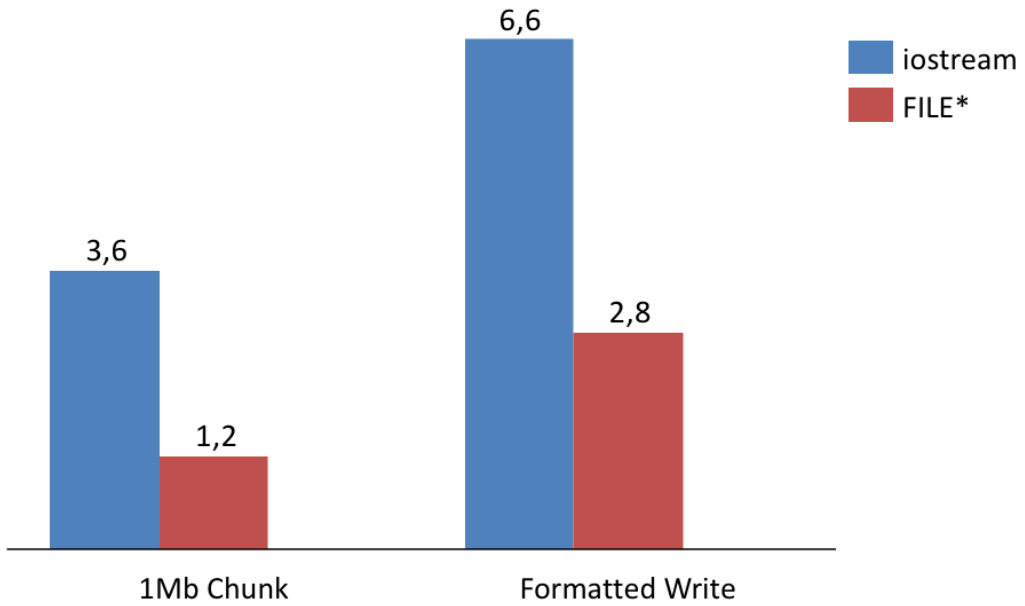
https://github.com/stheophil/iostreams_talk

https://github.com/CppUGBerlin/Talks

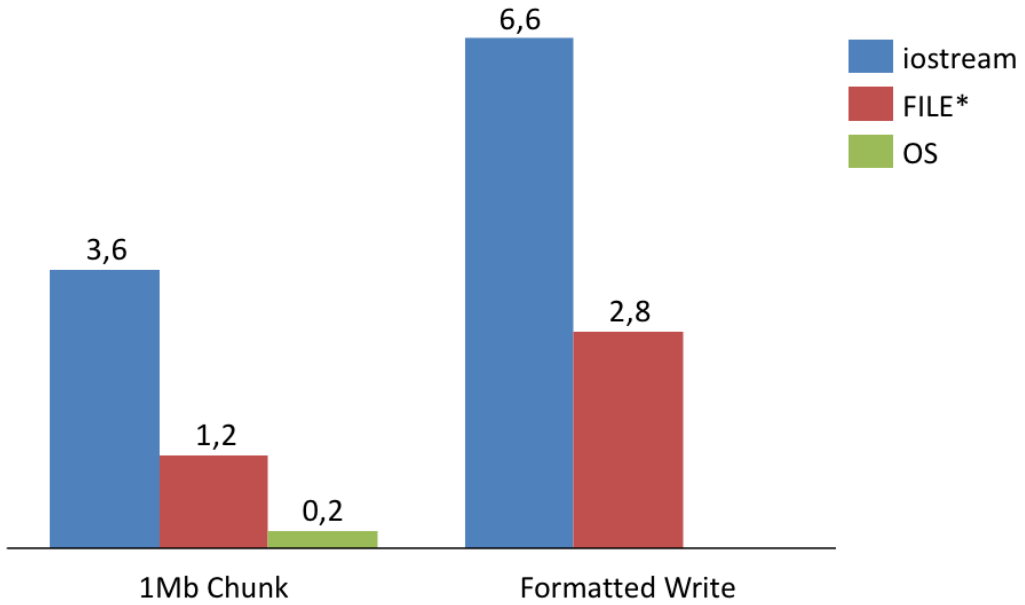
"iostreams are super slow!"

"iostreams are super slow!"

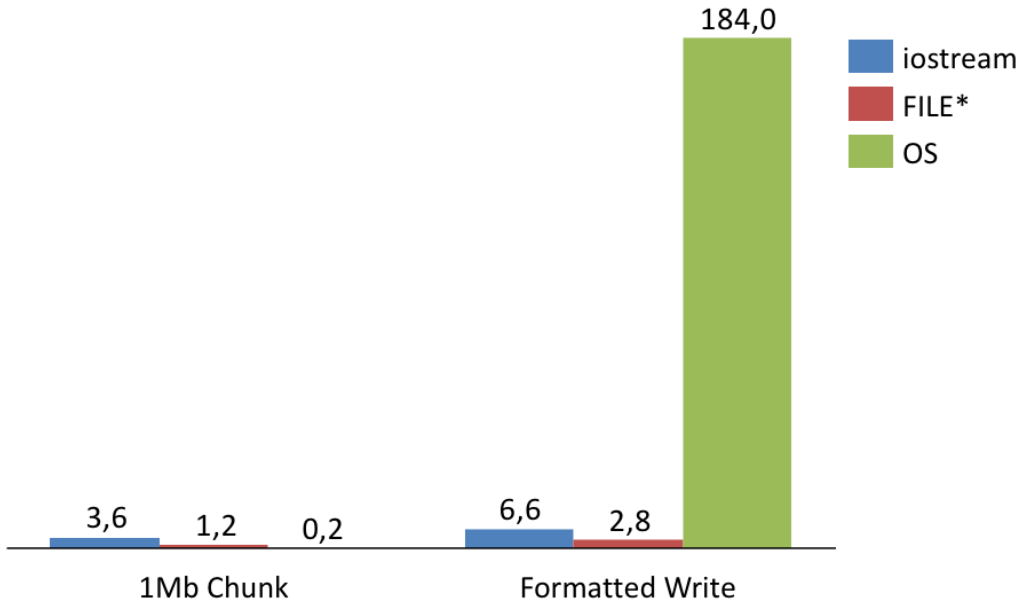
... but are they?



[Times in ms]



[Times in ms]


184,0
iostream
FILE*
OS
3,6
1,2
0,2
6,6
2,8
1Mb Chunk
Formatted Write

[Times in ms]

Remarks for Implementors

1. The number of `std::FILE*` that can be returned is limited on POSIX and Windows systems.
    - When using the Windows CRT, call `_setmaxstdio(8192)` to increase the limit to its maximum permitted value
    - On POSIX systems, call `setrlimit(RLIMIT_NOFILE, ...)` before you do any standard I/O call
2. According to the standard, when a `std::FILE*` is opened both for reading and writing, you must call `fseek` or `fflush` when you switch from reading to writing or vice versa.

Other File Implementations

- **Chromium:** https://chromium.googlesource.com/chromium/src.git/+/master/base/files/file.h
- **QT:** http://doc.qt.io/qt-5/qfile.html
- **Juce:** http://www.juce.com/doc/classFile

Formatting

- **cppformat:** printf-style formatting but type-safe
  http://cppformat.github.io/latest/reference.html