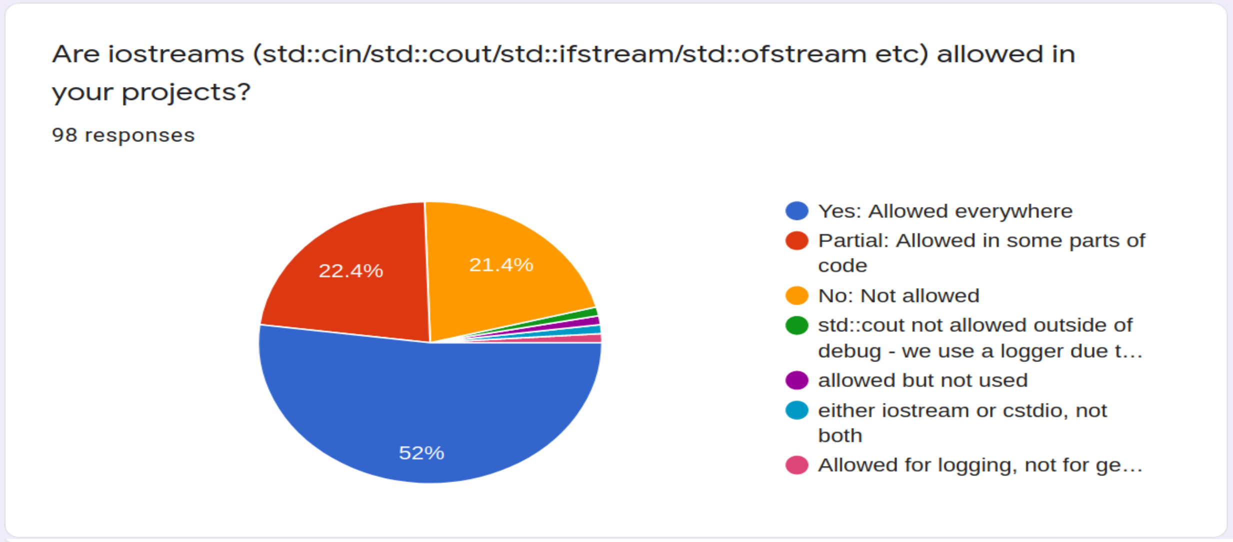
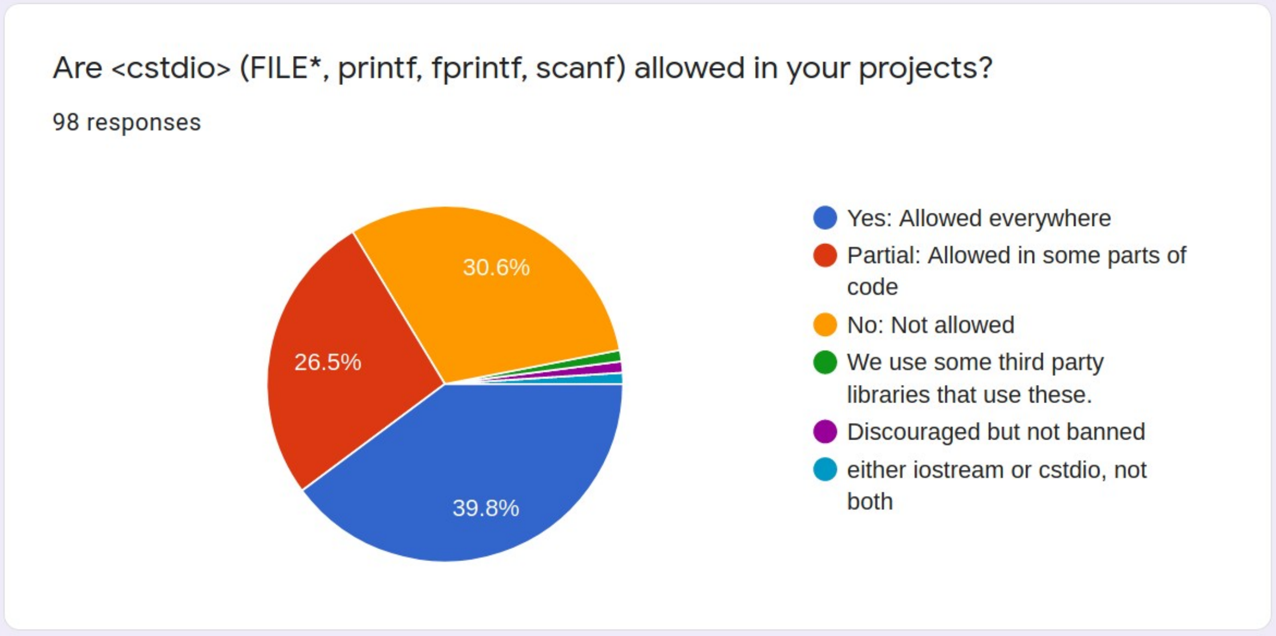
Document number: N6829R0
Date: 2020-02-20
Project: N+1 Problem of C++ I/O facilities
Reply-to: Shengdun Wang <swang206@binghamton.edu>
Audience: All

This paper is a not constructive proposal. It is probably not even a proposal since I am not going to solve this in this proposal. However, I see this as the worst issue in C++. Though I have my library of attempting to solve the problem and it is a success for me, smart people like Herb Sutter, Niall Douglas, Titus Winters, etc. may have better solutions than mine. I think the direction of future C++ is wrong. I think we need to delay further proposals like Network TS, Executors, Process, and LLFIO since they have an integration issue, and they will probably be another failure just like charconv.

Recently I did a simple survey on google forms. My statistics show 48% of C++ projects banned iostream in part or whole of their project. The statistics look very similar to Herb Sutter’ s Survey on C++ exceptions. Around half of C++ projects banned iostream.
I know my sample size too small and very likely to be biased. I would like to see Herb Sutter do the same survey in ISO C++ survey.
https://docs.google.com/forms/d/1z1iwXSFn_gBl2I8Um8Vix_CTScMgMzfgDf1RyxOIkQs/viewanalytics


Are iostreams (std::cin/std::cout/std::ifstream/std::ofstream etc) allowed in your projects?
98 responses


Are <cstdio> (FILE*, printf, fprintf, scanf) allowed in your projects?
98 responses

iostream and cstdio are largely banned in C++ projects. Most programmers in the world know them even they are not using C or C++, but they are also largely banned and also part of reasons why a lot of programmers quit C and C++ on the first day of their programming course.

We keep reinventing stuff for doing exactly the same thing what cstdio and iostream were doing.

std::filesystem, charconv, fmt, C++ sync stream, boost iostreams, asio Networking TS, LLFIO, Process, Executors (cppio). How many more do we have????? N+1???

They are doing the same thing with minor differences from an operating system perspective. Unfortunately, they are all designed for a particular purpose and never intended to talk with each other efficiently. Often, using them will still invoke calls on iostreams. iostreams are mostly implemented with FILE* in the mainstream implementation. The performance of stdio and iostream is terrible that defeats the purpose of using any of band fixes. Even process ts and networking ts rely on iostream, and I think it is a huge issue.

I see this as the same pattern of Herb Sutter’s previous deterministic exception proposal. Probably worse than the situation of exceptions and RTTI are. And also, I/O is the number one producer of exceptions. If we do not solve the issue of C++ I/O system, we will also never be able to fix the exception model in C++.

I do not use iostream either since I have recreated my I/O library, and it is at least ten times faster and it can do charconv, fmt, networking ts, process ts, iostream, stdio.h combined. I am one of the C++ community who hates iostream, and I know I am not alone. I tried to search for all the feedbacks on the internet to see the reasons people why people banned or hated C++’s iostream. I also read a lot of security papers about the entire I/O. Here are legitimate reasons why people do not like it.

1. C++ iostream is not zero overhead, including binary size, dependencies, performance.

However, unlike Herb Sutter, I do not blame everything on why people are banning one feature in C++ because it violates the zero-overhead principle. I think people ban one feature for a lot of reasons. Some ban one feature because they need to support legacy toolchains.

Here is an example in Linux Kernel C programmers ban // comments and blaming on us.
https://github.com/torvalds/linux/commit/df616d7a442b90798d63fbf4447154bbbb9040b1

However, it is a legitimate reason why people are banning it.
Zero overhead means two things.
a. You do not pay for what you do not use.
Even I do not use iostream, iostream will still be loaded into my memory with C++ runtime (libstdc++, libc++, MSVC abis) and bloat my binary size when I statically compile it. A common misunderstanding of dynamic linking is that people think it is free. It is not. It will incur runtime overhead, and that is why new languages like Rust do not use dynamic linking.

A standard security exploit is what we called Return-oriented Programming. Return-to-libc is one of the worst examples of ROP. C++’s stream and the entire C++ runtime rely on libc. iostream/fstream rely on FILE*, and they are also dynamic linking, which create a huge amount of attack space in C++ and severely harms the entire security in fact of C++. The modern operating system uses Address Space Layout Randomization (ASLR) to prevent ROP and SOP. However, it is not mandatory. In GCC, it requires -fPIC and -fPIE compilation toggle to enable it and a lot of C++ programs are not using it. It is neither free. ASLR, as a kind of abstraction, incurs runtime cost, which further harms the performance of C++.

b. If you do use it, you cannot reasonably do it better by your own.

Here is a benchmark of stdio.h, iostream, charconv compared to my I/O library for outputting integers from 0 to 9999999 to file.
On Linux, the benchmark looks like this:

```
./output_10M_size_t
std::FILE*:	0.612624519s		//fprintf(“%zu\n”,i);
std::ofstream:	0.5514320030000001s	//fout<<i<<’\n’;
std::ofstream with tricks:	0.388153462s	//fout<<i; rdbuf.sputc(‘\n’);
std::to_chars + ofstream rdbuf tricks:	0.220710758s//rdbuf.sputn result buffer of std::to_chars
std::to_chars + obuf_file:	0.150544426s	//std::to_chars and then call obuf_file’s write
obuf_file:	0.07302087s		//my library
obuf_file floating:	0.608490188s	//my library using ryu algorithm
obuf_file int hint:	0.125377798s	//my library using ryu algorithm + an integer hint
transform_text:	0.15563236500000002s	//my library then convert to text file (replace ‘\n’
with windows “\r\n”)
transform_ebcdic:	0.112890296s	//Use EBCDIC encoding for the result based on IBM’s
documentation
u8obuf_file:	0.06986633s		//my library but the character type is char8_t
u8obuf_file:	0.093287853s		//my library but the character type is char8_t. use a slow but
space-saving algorithm
onative_file dynamic:	0.070962353s	//Use buffer and do virtual function calls with write
c_file:	0.130164323s		//unlock the file and then exploit FILE*’s internal. Use my library’s
algorithm + avoid locale + implementation’s buffer pointer.
c_file_unlocked:	0.08767485200000001s	//exploit FILE*’s internal implementation’s buffer
pointer, use my library’s algorithm + avoid locale + implementation’s buffer pointer and do not unlock
the file.
cpp_fout_view:	0.087135114s	//use a cpp_fout view for getting eptr() exploit stream’s
buffer, use my library’s algorithm + avoid locale + implementation’s buffer pointer.
obuf_file_mutex:	0.075808476s	//my library + mutex for synchronization
fbh:	0.156044328s //use a fout view for getting eptr() exploit stream’s buffer, use my library’s
algorithm + avoid locale + implementation’s buffer pointer.
fast_io::concat:	0.17758979100000002s	//my library convert to std::string and then print
fmt::format_int obuf_file:	0.14043275300000002s	//exploit fmt::format_int and then call write
on my library
fmt::print:	0.6204992690000001s	//use fmt::print
fmt::format:	0.6542460920000001s	//use fmt::format
```

stdio.h and iostream are ten times slower than my I/O library on windows for this benchmark.

https://bitbucket.org/ejsvifq_mabmip/fast_io/src/master/benchmarks/0000.10m_size_t/output_10M_size_t.cc
On windows the performance of stdio.h and iostream is even 2 times worse.

This benchmark explains a lot. Replacing any part of stdio and iostream won' t improve performance at all.

Using fmt is slower than what it should be on Linux since the formatter still to be parsed at runtime and then invoke very inefficient calls on ofstream internal rdbuf().

Bandfix solutions do not work at all. Charconv and fmt has to run fast with my I/O library since I re-implement output buffer and avoid any virtual function call. Even with these bandfix might gain performance back, it is still slow since they do not have an efficient way to talk with buffer stream' s internal buffer, like in place parsing without a temporary result buffer.

BTW, C++' s iostream and stdio.h are so slow that even 2-5 times slower than AES CTR encryption (with hardware acceleration) and speck (with no hardware acceleration).
https://bitbucket.org/ejsvifq_mabmip/fast_io/src/master/benchmarks/0000.10m_size_t/with_crypto/tenm_size_t.cc

```
u8obuf_file ecb speck: 0.223634507s
u8obuf_file ecb aes:      0.099731052s
u8obuf_file cbc speck: 0.24069229400000003s
u8obuf_file cbc aes:      0.19090358500000001s
u8obuf_file ctr le aes:   0.20166913900000003s
u8obuf_file ctr le speck:            0.217204805s
u8obuf_file ctr be aes:  0.204318464s
u8obuf_file ctr be speck:            0.213630696s
```

For floating point, it is the same pattern. And GCC does not even implement charconv for floating point since no fully implemented algorithms can be used for printing long double precisely.

```
 ./output_10M_double
fprintf:  4.093168241s
ofstream:          5.268303850000001s
ofstream tricks:             5.105532644s
ofstream:          6.151441194s
cstyle file:          0.575491302s
cstyle file unlocked:      2.2435566860000002s
c_file_unlocked:           0.546750422s
obuf_file:          0.448763051s
u8obuf_file grisu_exact:            0.447062063s
u8obuf_file int hint:      0.47049970700000004s
stream_view:    0.8569448700000001s
obuf_file_mutex:          0.47617574900000004s
```

You can see the performance is really really bad for floating point and the precisions are half compared to algorithms like ryu and grisu-exact. De facto, the performance gap is at least 100 times.

```
-rw-r--r-- 1 cqwrteur cqwrteur 186618439 Feb 20 15:52 csfdb1.txt
-rw-r--r-- 1 cqwrteur cqwrteur 186618439 Feb 20 15:52 csfdb2.txt
-rw-r--r-- 1 cqwrteur cqwrteur  83889322 Feb 20 15:52 csfdb.txt
-rw-r--r-- 1 cqwrteur cqwrteur 186618439 Feb 20 15:52 obuf_filedb.txt
-rw-r--r-- 1 cqwrteur cqwrteur 186618439 Feb 20 15:52 obuf_file_mutexdb.txt
-rw-r--r-- 1 cqwrteur cqwrteur  83889322 Feb 20 15:52 ofs_tricks.txt
-rw-r--r-- 1 cqwrteur cqwrteur  83889322 Feb 20 15:52 ofs.txt
-rw-r--r-- 1 cqwrteur cqwrteur 186618439 Feb 20 15:52 smvdb.txt
-rw-r--r-- 1 cqwrteur cqwrteur 186618439 Feb 20 15:52 u8obuf_filedb_grisu_exact.txt
-rw-r--r-- 1 cqwrteur cqwrteur 186618439 Feb 20 15:52 u8obuf_filedb_hint.txt
```

2. C++ I/O system security is pretty bad. Think of this code

```
std::ofstream fout(“demo.txt”);
fout<<”Hello World\n”;
```

The code is bad since demo.txt might be a symlink. It might link to /*etc/passwd and the exploiter might gain privilege* and unfortunately unlike POSIX’s open/openat’s providing an O_NOFOLLOW option. It won’t fail at all.

Not only it has TOCTOU security vulnerabilities like Niall Douglas claimed in his LLFIO proposal, even without TOC, just TOU part is a vulnerability.

For C’s FILE*, the POSIX standard provides fdopen, which allows you for opening a FILE* with fdopen. However, no portable method in C++ of how to use an fd or a FILE* to construct a std::ofstream.

std::filesystem does not integrate with C++’s stream system. std::filesystem has been criticized by Herb Sutter and Niall Douglas by providing a bad interface. I think another issue is that no elegant way to use that as interface for constructing C++ fstream which leaves the TOCTOU unfix-able.

Second security vulnerablity is that std::ofstream is not process safe. No O_CLOEXEC tag for opening std::ofstream, which means file descriptors might leak to the child processes.

Sure, we can do bandfix to these issues for adding std::ios::nofollow and std::ios::cloexec options like POSIX does, but it does not solve the issue that C++ programs won’t use them in real practice. Most People do not read manuals.

Third, std::cout/std::cerr is not process safe, same with std::FILE* since they do not try to exploit the atomicity of system calls (write). To achieve that, the correct method should be to write to std::string and then flush out once. We are not that world today since C++ abstract machine does not understand the existence of other processes and virtual memory space.

Vtable injection and Type confusion issue of C++ stream system which leaves huge amount of attack space on C++ stream.

These security weakness are not fixable or native fixing will break a lot of existing C++ code base. LLFIO tried to solve the issue of that, but it still relies on some part of stream when you actually need to use it, which is same issue as filesystem, charconv and fmt in my opinion, just in another manner. They all have an integration issues which mean most people will not use it in real practice. It also does not worth the effort of library vendors to implement them.

3. No portable way to gain native handle from stream.
https://www.ginac.de/~kreckel/fileno/

Here is an article *fileno(3) on C++ Streams: A Hacker's Lament*
The guy tried to hack file descriptor in C++ streams.
I just extract some part of his code.

```
# if defined(__GLIBCXX__)  // >= GCC 3.4.0
  // This applies to cin, cout and cerr when not synced with stdio:
  typedef __gnu_cxx::stdio_filebuf<charT, traits> unix_filebuf_t;
  unix_filebuf_t* fbuf = dynamic_cast<unix_filebuf_t*>(stream.rdbuf());
  if (fbuf != NULL) {
    return fbuf->fd();
  }

  // This applies to filestreams:
  typedef std::basic_filebuf<charT, traits> filebuf_t;
  filebuf_t* bbuf = dynamic_cast<filebuf_t*>(stream.rdbuf());
  if (bbuf != NULL) {
    // This subclass is only there for accessing the FILE*.  Ouuwww, sucks!
    struct my_filebuf : public std::basic_filebuf<charT, traits> {
      int fd() { return this->_M_file.fd(); }
    };
    return static_cast<my_filebuf*>(bbuf)->fd();
  }

  // This applies to cin, cout and cerr when synced with stdio:
  typedef __gnu_cxx::stdio_sync_filebuf<charT, traits> sync_filebuf_t;
  sync_filebuf_t* sbuf = dynamic_cast<sync_filebuf_t*>(stream.rdbuf());
```

He is right, it sucks.

BTW, this code is not portable at all. It differs from versions and compiler. Clang, unfortunately, does not provide any public methods to gain native_handle(). FILE* is completely hided.

4. iostream relies on dynamic_cast and RTTI which mean RTTI will never get deprecated or removed from C++.

In CPPCON 2019, Herb Sutter proposed a std::down_cast to provide an efficient alternative to dynamic cast. However, stream does not apply to this since it is implemented by virtual inheritance. Just like code in 3 to gain file descriptor will break at all.

iostream’ s complex inheritance graph has been a long term issue of the langauge. It has been taught by many schools of bad examples of C++. Multi inheritance is considered bad in a lot of languages and java forbids multi-inheritance. C++ stream is probably the worst inheritance hierarchy in real world large code base.

5. mainstream iostream and fstream implementation relies on FILE*
Unfortunately, C++ standard never forbids that.

6. transmit and zero copy io issue
C++ does not provide a portable interface for zero copy I/O.
What really piss me off in network TS is the read/write pair. I think read/write pair is a problem. We should give it a name called transmit instead of read/write.
Unfortunately we focus too much on async I/O
In Niall Douglas’ s LLFIO proposal, he said that due to the improvement of operating system, sync I/O+ thread pool usually performs better than async I/O. I think he is totally right.
ASIO:
```
task<> tcp_echo_server() {
  char data[1024];
  for (;;) {
    size_t n = co_await socket.async_read_some(buffer(data));
    co_await async_write(socket, buffer(data, n));
  }
}

awaitable<void> listener()
{
  auto executor = co_await this_coro::executor;
  tcp::acceptor acceptor(executor, {tcp::v4(), 55555});
  for (;;)
  {
    tcp::socket socket = co_await acceptor.async_accept(use_awaitable);
    co_spawn(executor,
        [socket = std::move(socket)]() mutable
        {
          return echo(std::move(socket));
        },
        detached);
  }
}
```

vs

```
int main()
{
  tcp_server hd(2000);
  thread_pool_accept<acceptor>(hd,[](auto& acc)
  {
    transmit(acc,acc);
  });
}
```

An async echo server should be simple as this and I do not need char data[1024] at all. It creates security vulnerability. Directly accessing buffer should be avoided by C++ I/O library and replace it with transmit, print and scan.

However, network TS does not integrate with other part of C++ system which means we cannot do file to network zero copy I/O at all with network ts. Using transmit would permit further optimization by the library vendors since they could optimize more than users could do.

8. error codes of iostream and stdio.
They do not use exceptions and ignoring error codes are serious security vulnerabilities of C and C++ code. Herb Sutter’s zero-overhead deterministic exception will provide solutions but stdio and iostream won’t change due to compatibility of issue. That is why we need a new I/O library to completely replace stdio and iostream

9. iomanip’s exception safety issue

https://en.wikipedia.org/wiki/Criticism_of_C%2B%2B#Global_format_state_of_%3Ciostream%3E

C++ <iostream> unlike C <stdio.h> relies on a global format state. This fits very poorly together with exceptions, when a function must interrupt the control flow, after an error, but before resetting the global format state. One fix for this is to use Resource Acquisition Is Initialization (RAII) which is implemented in Boost[14] but is not a part of the C++ Standard Library.

This is another issue why iostream is not fixable at all. Global format is a mistake

10. Terrible diagnostic of iostream. Think of how terrible the diagnostic would be when try to output a type without defining operator<< or mis-define operator<<

For example, try to use

```
std::cout<<std::cin;    //The user wants to do std::cout<<std::cin.rdbuf();
```

GCC emits 255 Lines of error messages, most of them are useless because of overloading. Just one line of code produces pages of error messages are terrible.

```
/usr/local/include/c++/10.0.1/bits/ostream.tcc:106:20: note:   no known conversion for argument 1 from
  ‘std::istream’  {aka  ‘std::basic_istream<char>’ } to  ‘int’
  106 |    operator<<(int __n)
      |            ~~~~^~~
In file included from /usr/local/include/c++/10.0.1/iostream:39,
                 from wwwwww.cc:1:
/usr/local/include/c++/10.0.1/ostream:192:7: note: candidate:   ‘std::basic_ostream<_CharT,
_Traits>::__ostream_type& std::basic_ostream<_CharT, _Traits>::operator<<(unsigned int) [with _CharT =
char; _Traits = std::char_traits<char>; std::basic_ostream<_CharT, _Traits>::__ostream_type =
std::basic_ostream<char>]’
  192 |      operator<<(unsigned int __n)
      |      ^~~~~~~~
/usr/local/include/c++/10.0.1/ostream:192:31: note:   no known conversion for argument 1 from
  ‘std::istream’  {aka  ‘std::basic_istream<char>’ } to  ‘unsigned int’
  192 |      operator<<(unsigned int __n)
      |               ~~~~~~~~~~~~~^~~
/usr/local/include/c++/10.0.1/ostream:201:7: note: candidate:   ‘std::basic_ostream<_CharT,
_Traits>::__ostream_type& std::basic_ostream<_CharT, _Traits>::operator<<(long long int) [with _CharT =
char; _Traits = std::char_traits<char>; std::basic_ostream<_CharT, _Traits>::__ostream_type =
std::basic_ostream<char>]’
  201 |      operator<<(long long __n)
      |      ^~~~~~~~
```

11. locale

locale is useful. However, I think locale can be implemented in many better ways than C and C++ locale do. Even World of Warcraft AddOns Ace3Locale library does better job of localization than C and C++ locale system could do.

BTW, it should not be as a default option. It hurts the deterministic and performance of C and C++.

I have solutions to fix these issues but this is not a proposal for that.

12. charconv is a mistake.
C++17 filesystem is a problem, but charconv is worse. charconv was a C++ 17 library to fast format char arrays to integers or vice verse.

Unfortunately, this library is a mistake and Jonathan Wakely (libstdc++ tamer) completely agrees with me.

a. No algorithm in the world to satisfy all the requirements of C++ standard since it requires too many formats.

Precise fixed with precisions. Precise scientific with precisions. Precise hex with precisions. Precise shortest with precisions. These requirements cannot be fully implemented. No algorithms could achieve that. Even Ryu algorithm could not.

long double in VC is as long as double. However, long double GCC and Clang is 128 bits. Ryu algorithm does not provide fixed precisions, scientific precisions formatting for long double.

b. error codes
Why does the library produces error codes instead of exceptions??? It is another mistake since people will ignore it and become a security vulnerability.

c. directly access character buffer provides security vulnerability.
Again. Why?? Why do I need to touch buffer. Buffer is a problem, just like async_read and async_write has. Buffer should be considered as pointers. Avoid it as much as possible for a library interface.

d. does not provide efficient std::string formatting.
It does not produce std::string. Sure you can use charconv on local buffer and then copy content to construct std::string. However, that is another performance lost when we cannot do it in std::string' s internal space, because like I said, these libraries have integration issues with other libraries.

e. does not provide character support for wchar_t, char8_t, char16_t, char32_t etc.

Bandfix to charconv is not efficient and proved by my benchmarks in previous point. I think charconv should be deprecated in the future standard just like std::regex. std::fmt might face the same fate in the future, we will see what it would happen.

13. Debugging issue of std::cout/printf
One reason people banned std::cout/print is that they only use that for debugging. I also have solutions to fix this part of issue.

14. GUI application
GUI usually does not have a console and that is one reason my friend who writes machine learning code is not using it. I have solution to that problem as well just like 13 point.

15. Freestanding I/O
I do not buy the argument I/O is completely useless in freestanding environment. At least most part of I/O templates should work. I am creating a new testing operating system to test that.

16. No way to talk with stream and stdio directly and efficiently

Unfortunately, we use them everyday but they cannot talk with each other. I have solution to that as well.

Here are the N+1 issue of C++ io system. I think we need a new I/O library to replace stdio.h and iostream at all instead of creating separate part and become more and more N+1 issues. Finally they shared the same fate as std::regex did. I just want a fast integer to file output methods.

Sure my library proves they can all be fixed. My library can even coexist and exploit the performance existing facilities. However, other people might create better solutions than I have. However, I hope it is not an N+1 problem. It better be N+1-N = 1 problem. I think time to fix it is appropriate since Herb Sutter’s deterministic exception would be a good time point to fix this part of issue as well.

References

[1]charconv https://en.cppreference.com/w/cpp/header/charconv
[2]filesystem https://en.cppreference.com/w/cpp/filesystem
[3]Time-of-check to time-of-use https://en.wikipedia.org/wiki/Time-of-check_to_time-of-use
[4]Secure Coding in C and C++
https://resources.sei.cmu.edu/asset_files/BookChapter/2005_009_001_52710.pdf
[5]Return-oriented programming https://en.wikipedia.org/wiki/Return-oriented_programming
[6]https://github.com/fmtlib/fmt
[7]fast_io library https://bitbucket.org/ejsvifq_mabmip/fast_io
[8]Zero-overhead deterministic exceptions: Throwing values
http://open-std.org/JTC1/SC22/WG21/docs/papers/2018/p0709r0.pdf
[9]P1031R2: Low level file i/o library
http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2019/p1031r2.pdf
[10]Asio C++ Library https://github.com/chriskohlhoff/asio
[11]CWE-403: Exposure of File Descriptor to Unintended Control Sphere ('File Descriptor Leak')
http://cwe.mitre.org/data/definitions/403
[12]An Example of Multiple Inheritance in C++: A Model of the Iostream Library
https://dl.acm.org/doi/pdf/10.1145/70931.70935
[13]Debloating Software through Piece-Wise Compilation and Loading
https://www.usenix.org/system/files/conference/usenixsecurity18/sec18-quach.pdf
[14]P1750R1 A Proposal to Add Process Management to the C++ Standard Library http://open-std.org/JTC1/SC22/WG21/docs/papers/2019/p1750r1.pdf
[15]Criticism of C++ https://en.wikipedia.org/wiki/Criticism_of_C%2B%2B#Global_format_state_of_%3Ciostream%3E
[16]C++ IO survey https://docs.google.com/forms/d/1z1iwXSFn_gBl2I8Um8Vix_CTScMgMzfgDf1RyxOIkQs