The Challenges of Implementing the C Standard Library in C++

Siva Chandra Reddy

CppNow 2023

Who am I

- Software engineer at Google
- Working on LLVM's libc project (https://libc.llvm.org) for over 3 years
  - Main author of some of the core infrastructure
  - Wrote the FILE related infrastructure
  - Wrote the Linux threading libraries
  - Wrote initial parts of the math library

Outline

- What is the C standard library?
- A short overview of LLVM's implementation of the C standard library
- **Challenges in building LLVM's C standard library and the employed solutions**
- Questions

General Points

- Lots of code snippets
  - Not a tutorial on modern C++ features
  - Some of the items could be anti-patterns

What is the C standard library?

- As the name says, it is an implementation of the C standard library
- Usually referred to as the **libc**
  - A **libc** is typically much more than the C standard library
- At a minimum, a **libc** consists of:
  - An implementation of C standard library API
  - An implementation of the runtime support system
    - Startup objects like crt1.o on Linux provide the runtime support system

Extensions

- Few platforms cram more into it:
  - POSIX extensions
    - The pthread library is a POSIX extension
  - GNU extensions
    - `sincosf` from `math.h`
  - Linux extensions

LLVM's libc

- A greenfield libc implementation
- Aims to be a drop-in replacement for the legacy libc on any platform
- **Implemented in C++**
  - The code should look and feel like a modern C++ library

Why C++

- Employ modern practices
  - RAII
  - Power of modern features like `constexpr`
  - Use templates instead of duplicated code
  - Use variadic templates instead of complicated preprocessor macros
  - Use concepts as a way to formalize abstract interfaces
  - ...

Other rules for LLVM's libc

- Hard rule - No assembly language files; if no other options, use inline assembly
  - The entire libc code can be instrumented by sanitizers
  - The entire libc should be static-analysis friendly
  - We are part of the LLVM project
    - improve the compiler if there is a codegen problem
    - No *shortcuts* with hand-written assembly

X86_64 longjmp

```
LLVM_LIBC_FUNCTION(void, longjmp, (__jmp_buf * buf, int val)) {
      register __UINT64_TYPE__ rbx __asm__("rbx");
      …
      register __UINT64_TYPE__ rsp __asm__("rsp");
      register __UINT64_TYPE__ rax __asm__("rax");

      val = val == 0 ? 1 : val;
      LIBC_INLINE_ASM("mov %1, %0\n\t" : "=r"(rax) : "m"(val) :);
      LIBC_INLINE_ASM("mov %1, %0\n\t" : "=r"(rbx) : "m"(buf->rbx) :);
      …
      LIBC_INLINE_ASM("mov %1, %0\n\t" : "=r"(rsp) : "m"(buf->rsp) :);
      LIBC_INLINE_ASM("jmp *%0\n\t" : : "m"(buf->rip));
}
```

Other goals for LLVM's libc

- Modularity
  - Users can define what a libc is for their platform and then pick and choose from LLVM's libc to construct their platform libc
    - Largely driven by build system design

Challenges

Why care about the challenges?

- An interesting problem space
  - Allows us to appreciate the power of certain C++ features
- Learn how to modernize archaic coding practices/styles

General Theme - Avoiding chicken and egg problems

- Can you use the C++ standard library?
  - Think goodies like `std::string_view`, `std::move`
- Can we depend on the C++ runtime?
  - Think virtual functions/dynamic classes/exceptions
- Can we use C++ features which require runtime support from the libc?
  - Think global variables with non-trivial constructors

C++ Standard Libraries Use and Include the C Headers

- C++ standard libraries assume the existence of a fully functional C standard library
  - They include the C standard headers
  - They call into the C runtime

- Can we build on platforms where the C++ standard library is not yet available?

Constraint - Cannot use the C++ standard library

May be freestanding C++ headers?

- Yes - when we can start using more recent C++ and compilers

Consequence

- What about modern idiomatic C++?

A collection of self-contained C++ standard library goodies

- Yes - we implemented a lightweight, header only library of goodies from the C++ standard library
  - https://github.com/llvm/llvm-project/tree/main/libc/src/__support/CPP
- Instead of the `std` namespace, they live in the `cpp` namespace.
- Extended on a need basis - Not as as generic as the C++ standard library

  - `cpp::array`
  - `cpp::atomic`
  - `cpp::bit_cast`
  - `cpp::bitset`
  - `cpp::byte`
  - `cpp::expected`
  - `cpp::unexpected`
  - `cpp::function`
  - `cpp::numeric_limits`
  - `cpp::optional`
  - `cpp::span`
  - `cpp::string`
  - `cpp::string_view`
  - A bunch of `type_traits`
  - `cpp::integer_sequence`
  - `cpp::make_integer_sequence`
  - `cpp::forward`
  - `cpp::move`
  - `cpp::enable_if`

Internal API is idiomatic C++

Instead of	Use
`const char *` pointer and a `size_t` arguments and return values	`cpp::string_view` arguments and return values
raw pointer and size arguments and return values	`cpp::span`
returning `NULL`/special value on error but normal value otherwise	`cpp::optional` return value
setting `errno` on error	`cpp::expected` return value
`reinterpret_cast`	`cpp::bit_cast`
returning using pointer arguments	Return value of a focused type

The convenience of `string_view`

```
LLVM_LIBC_FUNCTION(char *, getenv, (const char *name)) {
  char **env_ptr = reinterpret_cast<char **>(__llvm_libc::app.envPtr);
  if (name == nullptr || env_ptr == nullptr)
    return nullptr;
  __llvm_libc::cpp::string_view env_var_name(name);
  if (env_var_name.size() == 0)
    return nullptr;
  for (char **env = env_ptr; *env != nullptr; env++) {
    __llvm_libc::cpp::string_view cur(*env);
    if (!cur.starts_with(env_var_name))
      continue;
    if (cur[env_var_name.size()] != '=')
      continue;
    // Remove the name and the equals sign.
    cur.remove_prefix(env_var_name.size() + 1);
    // We know that data is null terminated, so this is safe.
    return const_cast<char *>(cur.data());
  }
  return nullptr;
}
```

Use of `cpp::string_view` as function args

```
LIBC_INLINE void write_to_stderr(cpp::string_view msg) {
  __llvm_libc::syscall_impl(SYS_write, 2 /* stderr */,
                            msg.data(), msg.size());
}
```

Returning the error instead of setting `errno` ... 1/2

The POSIX **`dirent.h`** API

```
DIR *opendir(const char *);

    On error,  these functions shall return a null pointer and
    set errno to indicate the error.

struct dirent *readdir(DIR *);

    When an error is encountered, a null pointer shall be returned
    and errno shall be set to indicate the error.

int  closedir(DIR *);

    On error, -1 shall be returned and errno set to indicate the
    error.
```

Returning the error instead of setting `errno` ... 2/2

```
class Dir {
  …
    static ErrorOr<Dir *> open(const char *path);

    ErrorOr<struct ::dirent *> read();

    // Returns the error number to indicate the success or failure.
    int close();
  …
};
```

What is ErrorOr<T>

```
namespace __llvm_libc {

    template <class T>
    using ErrorOr = cpp::expected<T, int>;

} // namespace __llvm_libc
```

The public function addresses the standard requirements

```
LLVM_LIBC_FUNCTION(::DIR *, opendir, (const char *name)) {
  auto dir = Dir::open(name);
  if (!dir) {
    libc_errno = dir.error();
    return nullptr;
  }
  return reinterpret_cast<DIR *>(dir.value());
}
```

Example of bit_cast

- `FPBits<T>` - A template class which stores the encoded floating point number as an integer value

```
template <typename T>
class FPBits {
  UIntType bits;
  …
  LIBC_INLINE T get_val() const {
    return *reinterpret_cast<T *>(&bits);
  }
  LIBC_INLINE void set_val(T value) {
    bits = *reinterpret_cast<UIntType *>(&value);
  }
  …
};
```

```
template <typename T>
class FPBits {
  UIntType bits;
  …
  LIBC_INLINE T get_val() const {
    return cpp::bit_cast<T>(bits);
  }
  LIBC_INLINE void set_val(T value) {
    bits = cpp::bit_cast<UIntType>(value);
  }
  …
};
```

Example of returning a focused `struct`

The C standard functions for string to integer conversion:

```
long int strtol(const char * restrict nptr,
                char ** restrict endptr, int base);
long long int strtoll(const char * restrict nptr,
                      char ** restrict endptr, int base);
unsigned long int strtoul(const char * restrict nptr,
                          char ** restrict endptr, int base);
unsigned long long int strtoull(const char * restrict nptr,
                                char ** restrict endptr, int base);
```

Example of returning a focused `struct` 2/3

Wording in the standard:

> **A pointer to the final string is stored in the object pointed to by *endptr*, provided that *endptr* is not a null pointer.**
>
> **The strtol, strtoll, strtoul, and strtoull functions return the converted value, if any. *If no conversion could be performed, zero is returned.* If the correct value is outside the range of representable values, LONG_MIN, LONG_MAX, LLONG_MIN,LLONG_MAX, ULONG_MAX, or ULLONG_MAX is returned (according to the return type and sign of the value, if any), and the value of the macro *ERANGE is stored in errno*.**

Example of returning a focused `struct` 3/3

The internal string-to-integer conversion utility

```
template <class T>
StrToNumResult<T> strtointeger(const char *__restrict src, int base);

template <typename T> struct StrToNumResult {
  T value;
  int error; // For out of range value
  ptrdiff_t parsed_len;
  ...
};
```

The public function addresses the standard requirements

```
LLVM_LIBC_FUNCTION(long, strtol,
                   (const char *__restrict str, char **__restrict str_end,
                    int base)) {
  auto result = internal::strtointeger<long>(str, base);
  if (result.has_error())
    libc_errno = result.error;

  if (str_end != nullptr)
    *str_end = const_cast<char *>(str + result.parsed_len);

  return result;
}
```

Constraint - No dependence on the C++ Runtime

- **Strictly no use of the `virtual` keyword**
  - No virtual functions
  - No virtual inheritance

Why?

- **`virtual` related mechanics are implementation defined**
  - Use of pure virtual function can insert calls to `__cxa_pure_virtual`

Consequence

- What about allocation and deallocation?
- What about abstract interfaces?

Allocation / Deallocation

- Recollect the goal:
  - **The libc source code should look and feel like that of a normal and modern C++ library**

- **Allocations and deallocations with `new` and `delete`**
  - Cannot use `new` and `delete` from the C++ runtime library
  - All allocations and deallocations have to go through the libc internal `new` and `delete`

libc internal placement new

```
LIBC_INLINE void *operator new(size_t size,
                               __llvm_libc::AllocChecker &ac) noexcept {
 return __llvm_libc::AllocChecker::alloc(size, ac);
}
```

The AllocChecker class

```
namespace __llvm_libc {
class AllocChecker {
  bool success = false;
  AllocChecker &operator=(bool status) {
    success = status;
    return *this;
  }
public:
  AllocChecker() = default;
  operator bool() const { return success; }
  LIBC_INLINE static void *alloc(size_t s, AllocChecker &ac) {
    void *mem = ::malloc(s);
    ac = (mem != nullptr);
    return mem;
  }
  …
};
} // namespace __llvm_libc
```

Using the libc internal `operator new`

```
__llvm_libc::AllocChecker ac;
auto *obj = new (ac) Type(...);
if (!ac) {
  // handle allocation failure.
}
...
```

libc internal `operator delete`

- Standard does not permit inline implementation of **operator delete**
- Out of line **`operator delete`** replaces **`operator delete`** for the entire application
- Use ELF trickery to get around these problems

libc internal `operator delete`

- In the header file, declared as:

```
void operator delete(void *) noexcept __asm__("__llvm_libc_delete");
```

- In the `.cpp` file, implemented as:

```
void operator delete(void *mem) noexcept { ::free(mem); }
```

Using the libc internal `operator delete`

```
#include <new.h>
…
__llvm_libc::AllocChecker ac;
auto *obj = new (ac) Type(...);
if (!ac) {
  // handle allocation failure.
}
…
delete obj;
```

Constraint - No runtime init/destruction of global objects

- Global objects cannot require runtime initialization
- Clean up of global objects should be explicit
  - Should not require a destructor call at program termination
  - Can be handled by registering an atexit callback

Why?

- The libc facilitates the runtime initialization
- If libc uses them, then we have to be very careful about where we use them
- Keep it simple - just disallow runtime initialization of global objects

Use constexpr constructors for global objects

- Even trivial/defaulted constructors should be constexpr
  - Ensures compile time initialization

Destructors of global objects

- Destructors should be trivial and defaulted
  - Cannot have members which require destruction
- Use explicit destruction for cleanup

`BlockStore` instead of a vector data structure

- The vector data structure allocates the memory on construction
  - Cannot have a constexpr constructor
- The **`BlockStore`** is linked list of blocks of memory
  - Starts with a block of memory
  - When the size has to grow, instead of allocating a larger chunk of memory and copying items over, a new block is added to the linked list
  - Used to store the list of atexit functions
    - Since all items are visited in order, linked list overhead is minimal

BlockStore

```
template <typename T, size_t BLOCK_SIZE, bool REVERSE_ORDER = false>
class BlockStore {
 ...
public:
 constexpr BlockStore() = default;
 ~BlockStore() = default;
  ...
 static void destroy(
    BlockStore<T, BLOCK_SIZE, REVERSE_ORDER> *block_store);
  ...
};
```

Processing `atexit` callbacks

```
void call_exit_callbacks() {
  handler_list_mtx.lock();
  while (!exit_callbacks.empty()) {
    auto unit = exit_callbacks.back();
    exit_callbacks.pop_back();
    handler_list_mtx.unlock();
    unit.callback(unit.payload);
    handler_list_mtx.lock();
  }
  ExitCallbackList::destroy(&exit_callbacks);
}
```

C++ Goodies

More `constexpr`

- Compile time tables - used in the implementation of **`strerror`** function.

```
char *strerror(int errnum);
```

- At a high level it is implemented as a table lookup
- The interesting part - tables are constructed at compile time
- Other libc systems have implemented the construction of compile time tables using complicated preprocessor macros
- LLVM's libc does it in C++ using the power of **`constexpr`**

Error Tables

Table of the C standard errors:

```
namespace __llvm_libc::internal {

inline constexpr const MsgTable<4> STDC_ERRORS = {
    MsgMapping(0, "Success"),
    MsgMapping(EDOM, "Numerical argument out of domain"),
    MsgMapping(ERANGE, "Numerical result out of range"),
    MsgMapping(EILSEQ, "Invalid or incomplete multibyte or wide character"),
};

} // namespace __llvm_libc::internal
```

Error Tables

Table of POSIX errors:

```
namespace __llvm_libc::internal {

inline constexpr MsgTable<76> POSIX_ERRORS = {
    MsgMapping(EPERM, "Operation not permitted"),
    MsgMapping(ENOENT, "No such file or directory"),
    MsgMapping(ESRCH, "No such process"),
    MsgMapping(EINTR, "Interrupted system call"),
    MsgMapping(EIO, "Input/output error"),
    MsgMapping(ENXIO, "No such device or address"),
    MsgMapping(E2BIG, "Argument list too long"),
    …
};
```

The MsgMapping and MsgTable types

```
struct MsgMapping {
  int num;
  cpp::string_view msg;

  constexpr MsgMapping() : num(0), msg() {}

  constexpr MsgMapping(int init_num, const char *init_msg)
      : num(init_num), msg(init_msg) {}
};

template <size_t N> using MsgTable = cpp::array<MsgMapping, N>;
```

Error table for the platform

Linux example:

```
#ifdef __linux__
inline constexpr auto PLATFORM_ERRORS =
   STDC_ERRORS + POSIX_ERRORS + LINUX_ERRORS;
#else
inline constexpr auto PLATFORM_ERRORS = STDC_ERRORS;
#endif
```

constexpr operator+

```
template <size_t N1, size_t N2>
constexpr MsgTable<N1 + N2> operator+(const MsgTable<N1> &t1,
                                      const MsgTable<N2> &t2) {
  MsgTable<N1 + N2> res{};
  for (size_t i = 0; i < N1; ++i)
    res[i] = t1[i];
  for (size_t i = 0; i < N2; ++i)
    res[N1 + i] = t2[i];
  return res;
}
```

Using these tables

```
constexpr size_t TOTAL_STR_LEN =  total_str_len(PLATFORM_ERRORS);

constexpr size_t ERR_ARRAY_SIZE = max_key_val(PLATFORM_ERRORS);

static constexpr MessageMapper<ERR_ARRAY_SIZE, TOTAL_STR_LEN>
    error_mapper(PLATFORM_ERRORS);
```

Arranging the tables in order - more `constexpr` magic

```
template <size_t ARR_SIZE, size_t TOTAL_STR_LEN> class MessageMapper {
  int msg_offsets[ARR_SIZE] = {-1};
  char string_array[TOTAL_STR_LEN] = {'\0'};
public:
  …

  cpp::optional<cpp::string_view> get_str(int num) const {
    if (num >= 0 && static_cast<size_t>(num) < ARR_SIZE &&
        msg_offsets[num] != -1) {
      return {string_array + msg_offsets[num]};
    } else {
      return cpp::optional<cpp::string_view>();
    }
  }
};
```

constextr Constructor of MessageMapper... 1/2

```
template <size_t N>
constexpr MessageMapper(const MsgTable<N> &table) {
    cpp::string_view string_mappings[ARR_SIZE] = {""};
    for (size_t i = 0; i < table.size(); ++i)
      string_mappings[table[i].num] = table[i].msg;

    ...
}
```

constextr Constructor of MessageMapper... 2/2

```
template <size_t N>
constexpr MessageMapper(const MsgTable<N> table) {
    ...
    int string_array_index = 0;
    for (size_t cur_num = 0; cur_num < ARR_SIZE; ++cur_num) {
      if (string_mappings[cur_num].size() != 0) {
        msg_offsets[cur_num] = string_array_index;
        for (size_t i = 0; i < string_mappings[cur_num].size() + 1;
             ++i, ++string_array_index) {
          string_array[string_array_index] = string_mappings[cur_num][i];
        }
      } else {
        msg_offsets[cur_num] = -1;
      }
    }
  }
```

The resulting object file has only one physical table

```
Num:  Value    Size Type    Bind   Vis      Ndx Name
  0: 00000000     0 NOTYPE  LOCAL  DEFAULT  UND
  1: 00000000     0 FILE    LOCAL  DEFAULT  ABS error_to_string.cpp
  2: 00000000    14 OBJECT  LOCAL  DEFAULT   13 .L.str
  3: 00000000  3652 OBJECT  LOCAL  DEFAULT   14 _ZN11__llvm_libc8internalL12error_mapperE
  4: 00000000     0 SECTION LOCAL  DEFAULT   14 .rodata._ZN11__llvm_libc8internalL12error_mapperE
  5: 00000000   577 FUNC    GLOBAL DEFAULT    3 [_ZN11__llvm_libc8internal18build_error_stringE…]
  6: 00000000    89 FUNC    GLOBAL DEFAULT    5 _ZN11__llvm_libc16get_error_stringEi
  7: 00000000    26 TLS     GLOBAL DEFAULT   12 _ZN11__llvm_libc8internal12error_bufferE
  8: 00000000    68 FUNC    GLOBAL DEFAULT    7 _ZN11__llvm_libc16get_error_stringEiNS_3cpp4spanIcEE
  9: 00000000    17 FUNC    WEAK   HIDDEN    10 _ZTWN11__llvm_libc8internal12error_bufferE
```

Templates, of course!

- Avoid to duplicated
  - C standard math function have `float`, `double` and `long double`
  - Standard specification of the `logb` family of functions

```
float       logbf(float);
double      logb(double);
long double logbl(long double);
```

Template implementation of `logb`

```
template <typename T, cpp::enable_if_t<cpp::is_floating_point_v<T>, int> = 0>
LIBC_INLINE T logb(T x) {
  FPBits<T> bits(x);
  if (bits.is_zero()) {
    return T(FPBits<T>::neg_inf());
  } else if (bits.is_nan()) {
    return x;
  } else if (bits.is_inf()) {
    // Return positive infinity.
    return T(FPBits<T>::inf());
  }

  NormalFloat<T> normal(bits);
  return static_cast<T>(normal.exponent);
}
```

Variadic Templates

```
template <typename T> LIBC_INLINE T polyeval(T, T a0) { return a0; }

template <typename T, typename... Ts>
LIBC_INLINE T polyeval(T x, T a0, Ts... a) {
  return multiply_add(x, polyeval(x, a...), a0);
}
```

Future Work

Still more `constexpr`

- `constexpr` doesn't play well with compiler builtins
  - Compilers builtins are usually specialized instruction sequences
  - They are usually target agnostic interfaces to generate target specific optimizations
  - Examples:
    - `__builtin_addcb`, `__builtin_addcs` ...
    - `__builtin_subcb`, `__builtin_subcs` ...

Current solution for functions which call builtins

- Include an r-value reference overload which does not call builtins
- Potential problem: When the function is not evaluated at compile time, we don’t get the benefits of compiler builtins

Example - The `UInt` type

```
template<size_t Bits>
struct UInt {
…

 UInt<Bits> operator+(const UInt<Bits> &other) const {
    …
    … = add_with_carry(val[i], other.val[i], s.carry);
    …
 }


 constexpr UInt<Bits> operator+(UInt<Bits> &&other) const {
    …
    … = add_with_carry_const(val[i], other.val[i], s.carry);
    …
 }
…
};
```

Abstract interfaces using concepts

- Hopefully can fill the gap we have because disallowing `virtual` functions
  - Need a clean type erasure solution
    - Limited function pointers?
    - `cpp::function`?

Thank You