Killing C++ Serialization Overhead & Complexity

EYAL ZEDAKA

Cppcon
The C++ Conference

2022
September 12th-16th

Note:
All the “godbolt” links point to http://localhost:10240, after you click the link, change the URL to godbolt.org, change the compiler to clang 15 for similar results to the presentation, please replace the include of
#include "zpp_bits.h" into
#include “https://raw.githubusercontent.com/eyalz800/zpp_bits/main/zpp_bits.h”

What is object serialization?

The process of converting a C++ object into a sequence of bytes

The reverse is often called “deserialization”

Why - To save objects to a file, transfer over the network, and communicate between programs

Let’s review some statements about C++ serialization…

Object serialization in modern C++ has zero runtime overhead!

```cpp
enum Color : std::uint8_t {
   Red,
   Green,
   Blue
};

struct Vec3 {
   float x;
   float y;
   float z;
};

struct Weapon {
   std::string name;
   std::int16_t damage;
};

struct Monster {
   Vec3 pos;
   std::int16_t mana;
   std::int16_t hp;
   std::string name;
   std::vector<std::uint8_t> inventory;
   Color color;
   std::vector<Weapon> weapons;
   Weapon equipped;
   std::vector<Vec3> path;
};
```

library	ser time	des time	total time
yas	2,114ms	1,558ms	3,672ms
bitsery	2,128ms	1,832ms	3,960ms
flatbuffers	9,812ms	3,472ms	13,284ms
msgpack	3,563ms	14,705ms	18,268ms
cereal	9,977ms	8,565ms	18,542ms
boost	16,011ms	13,017ms	29,028ms
protobuf	**18,125ms**	**20,211ms**	**38,336ms**

By hand	**1,391ms**	**1321ms**	**2712ms**

This talk	**?**	**?**	**?**

https://github.com/fraillt/cpp_serializers_benchmark/tree/a4c0ebfb

Many thanks to the benchmark author - Mindaugas Vinkelis

Results use the “general” configuration.

I can’t afford to use C++ serialization because it would not fit my embedded system

```
enum Color : std::uint8_t {
    Red,
    Green,
    Blue
};

struct Vec3 {
    float x;
    float y;
    float z;
};

struct Weapon {
    std::string name;
    std::int16_t damage;
};

struct Monster {
    Vec3 pos;
    std::int16_t mana;
    std::int16_t hp;
    std::string name;
    std::vector<std::uint8_t> inventory;
    Color color;
    std::vector<Weapon> weapons;
    Weapon equipped;
    std::vector<Vec3> path;
};
```

library	bin size
yas	51,000B
bitsery	53,728B
flatbuffers	62,512B
msgpack	77,384B
cereal	61,480B
boost	237,008B
protobuf	**2,032,712B**

By hand	**43,112B**

This talk	**?**

Note: The binary size measured has exceptions turned on and includes the benchmark code size (not only the serialization code).

https://github.com/fraillt/cpp_serializers_benchmark/tree/a4c0ebfb

Many thanks to the benchmark author - Mindaugas Vinkelis

Results use the “general” configuration.

This Talk – The zpp::bits serialization Library

- Start by beating our favorite C++ serialization libraries, in benchmarks that we didn't write

- **Simple to use** –
  Almost always – require not even a single change to our classes

- **Don't pay for what you don't use** –
  Control the serialization format and overhead, opt in solution for compatibility and cross language communication

- **Has RPC implementation** –
  Remote procedure call – binding serialization to function calls and serving them

- **Zero overhead** –
  Can't write better by hand!

  We will try, don't worry

- **Leave no room for a lower level language** –
  Completely freestanding / embedded friendly – with or without exceptions.

Who Am I – Eyal Zedaka

- **Technical leader** – C++, operating systems, low level, software security
- **Principal Manager @ Microsoft** – very recently moved from Magic Leap
- **C++ Lecturer** – On invite basis, every once in a while, I spoke last year at CppCon2021
- **Open Source** – Selected examples from my github:
    - C++ hypervisor PoC – for intel 64 bit, windows, linux and UEFI
    - From CppCon2021 – A library that implements C++ exceptions with.. coroutines!
- **Preferred editor – Vim or Neovim –** get my vim setup today from my github
- **Tabs or Spaces –** Spaces (better be 4 spaces)

Overview

- **Introducing the “zpp::bits” Library**
  - Review some open source benchmarks
  - Serialization example & format with zpp::bits
  - Compare handwritten code vs the library
  - The Zero Overhead Toolbox that zpp::bits uses
- **Implement Our Own Zero Overhead Serializer**
  - Serializer implementation walkthrough
  - Can it beat the handwritten code
  - Analysis of the overhead and attempt to improve
- **Key Features Discussion**
  - Reflection, in the pre-reflection era
  - Remote Procedure Call
  - Cross Programming Language
- **Summary**
  - What we achieved
  - What is still not perfect

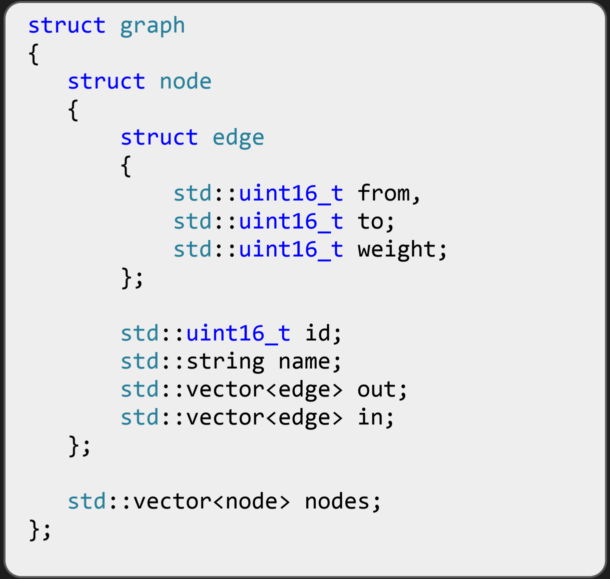
```
enum Color : std::uint8_t {
    Red,
    Green,
    Blue
};

struct Vec3 {
    float x;
    float y;
    float z;
};

struct Weapon {
    std::string name;
    std::int16_t damage;
};

struct Monster {
    Vec3 pos;
    std::int16_t mana;
    std::int16_t hp;
    std::string name;
    std::vector<std::uint8_t> inventory;
    Color color;
    std::vector<Weapon> weapons;
    Weapon equipped;
    std::vector<Vec3> path;
};
```

library	ser time	des time	total time	bin size	data size
yas	2,114ms	1,558ms	3,672ms	51,000B	10,463B
bitsery	2,128ms	1,832ms	3,960ms	53,728B	**6,913B**
flatbuffers	9,812ms	3,472ms	13,284ms	62,512B	14,924B
msgpack	3,563ms	14,705ms	18,268ms	77,384B	8,857B
cereal	9,977ms	8,565ms	18,542ms	61,480B	10,413B
boost	16,011ms	13,017ms	29,028ms	237,008B	11,037B
protobuf	**18,125ms**	**20,211ms**	**38,336ms**	**2,032,712B**	10,018B
By hand	**1,391ms**	**1321ms**	**2712ms**	**43,112B**	10,413B
zpp::bits	**790ms**	**715ms**	**1,505ms**	**47,128B**	**8,413B**

https://github.com/fraillt/cpp_serializers_benchmark/tree/a4c0ebfb

Many thanks to the benchmark author - Mindaugas Vinkelis

Results use the “general” configuration.



```cpp
struct graph
{
   struct node
   {
      struct edge
      {
         std::uint16_t from,
         std::uint16_t to;
         std::uint16_t weight;
      };

      std::uint16_t id;
      std::string name;
      std::vector<edge> out;
      std::vector<edge> in;
   };

   std::vector<node> nodes;
};
```

library	serialize	deserialize	total
cista (slim)	20.6ms	0.184ms	20.784ms
capnproto	164ms	0.001ms	164.001ms
cereal	207ms	192ms	399ms
flatbuffers	3059ms	93.7ms	3152.7ms

zpp::bits	**8.91ms**	**7.87ms**	**16.78ms**

https://github.com/felixguendling/cpp-serialization-benchmark/tree/f8216ebe

Many thanks to the benchmark author - Felix Gündling

Note: The results on this slide are from my fork’s github CI pipeline, rather than the official repo.

```cpp
enum class OrderSide : std::uint8_t {
   BUY, SELL
};

enum class OrderType : std::uint8_t {
   MARKET, LIMIT, STOP
};

struct Order {
   int Id;
   char Symbol[10];
   OrderSide Side;
   OrderType Type;
   double Price;
   double Volume;
};

struct Balance {
   char Currency[10];
   double Amount;
};

struct Account {
   int Id;
   std::string Name;
   Balance Wallet;
   std::vector<Order> Orders;
};
```

library	serialize	deserialize	total
sbe	53ns	83ns	136ns
fbe	117ns	100ns	217ns
capnproto	298ns	290ns	588ns
flatbuffers	403ns	107ns	510ns
protobuf	412ns	574ns	986ns
zpp::bits	**27ns**	**26ns**	**53ns**

https://github.com/chronoxor/CppSerialization/tree/f73fbc66

Many thanks to the benchmark author - Ivan Shynkarenka

Note: The results on this slide are from my fork's github CI pipeline, rather than the official repo.

How to use zpp::bits?

```cpp
struct address_book
{
    enum class phone_type : int {
        mobile, home, work,
    };

    struct phone_number {
        std::string number;
        phone_type type;
    };

    struct person {
        std::string name;
        int id;
        std::string email;
        std::vector<phone_number> phones;
    };

    std::vector<person> people;
};
```

→

```proto
syntax = "proto3";

message person {
  string name = 1;
  int32 id = 2;
  string email = 3;

  enum phone_type {
    mobile = 0;
    home = 1;
    work = 2;
  }

  message phone_number {
    string number = 1;
    phone_type type = 2;
  }

  repeated phone_number phones = 4;
}

message address_book {
  repeated person people = 1;
}
```

No!

Unnecessary friction

+

Outside the language!

How to use zpp::bits?

```cpp
struct address_book
{
   enum class phone_type : int {
       mobile, home, work,
   };

   struct phone_number {
       std::string number;
       phone_type type;
   };

   struct person {
       std::string name;
       int id;
       std::string email;
       std::vector<phone_number> phones;
   };

   std::vector<person> people;
};
```

→

```cpp
struct address_book
{
   enum class phone_type : int {
       mobile, home, work,
   };

   void serialize(auto & archive) {
       archive(people); // Serialize member "people"
   }

   struct phone_number {
       void serialize(auto & archive) {
           archive(number, type);
       }

       std::string number;
       phone_type type;
   };

   struct person {
       void serialize(auto & archive) {
           archive(name, id, email, phones);
       }

       std::string name;
       int id;
       std::string email;
       std::vector<phone_number> phones;
   };

   std::vector<person> people;
};
```

No!

Exceptions – not freestanding

+

Unnecessary friction – we can do better

How to use zpp::bits?

```cpp
struct address_book
{
   enum class phone_type : int
   {
       mobile = 0,
       home = 1,
       work = 2,
   };

   struct phone_number
   {
       std::string number;
       phone_type type;
   };

   struct person
   {
       std::string name;
       int id;
       std::string email;
       std::vector<phone_number> phones;
   };

   std::vector<person> people;
};
```

→

```cpp
struct address_book
{
   enum class phone_type : int
   {
       mobile = 0,
       home = 1,
       work = 2,
   };

   struct phone_number
   {
       std::string number;
       phone_type type;
   };

   struct person
   {
       std::string name;
       int id;
       std::string email;
       std::vector<phone_number> phones;
   };

   std::vector<person> people;
};
```

No changes required, how? We'll be back with that later

Serialization Example with zpp::bits

```cpp
struct address_book
{
   enum class phone_type : int
   {
       mobile = 0,
       home = 1,
       work = 2,
   };

   struct phone_number
   {
       std::string number;
       phone_type type;
   };

   struct person
   {
       std::string name;
       int id;
       std::string email;
       std::vector<phone_number> phones;
   };

   std::vector<person> people;
};
```

```cpp
std::size_t serialize(const address_book & book,
                      std::span<std::byte> data)
{






}
```

```cpp
std::size_t deserialize(address_book & book,
                        std::span<const std::byte> data)
{






}
```

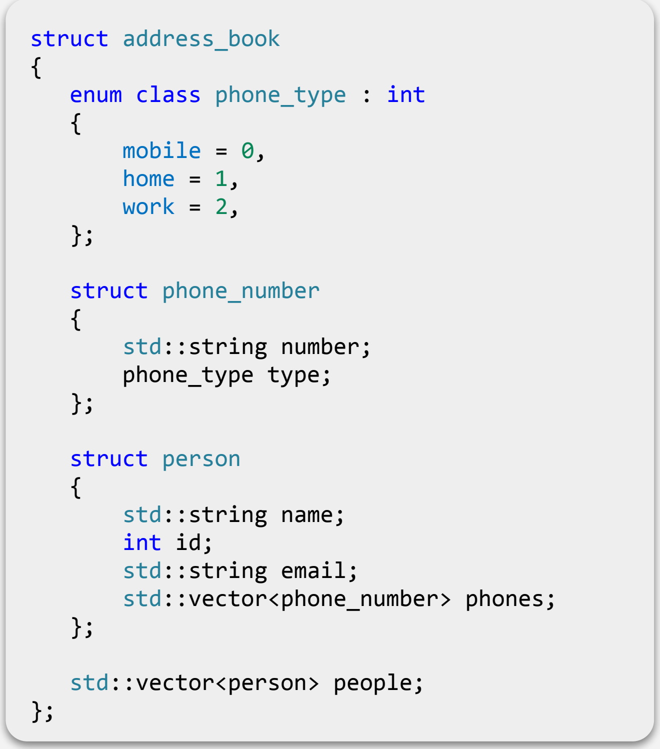
Serialization Example with zpp::bits

```cpp
struct address_book
{
   enum class phone_type : int
   {
       mobile = 0,
       home = 1,
       work = 2,
   };

   struct phone_number
   {
       std::string number;
       phone_type type;
   };

   struct person
   {
       std::string name;
       int id;
       std::string email;
       std::vector<phone_number> phones;
   };

   std::vector<person> people;
};
```

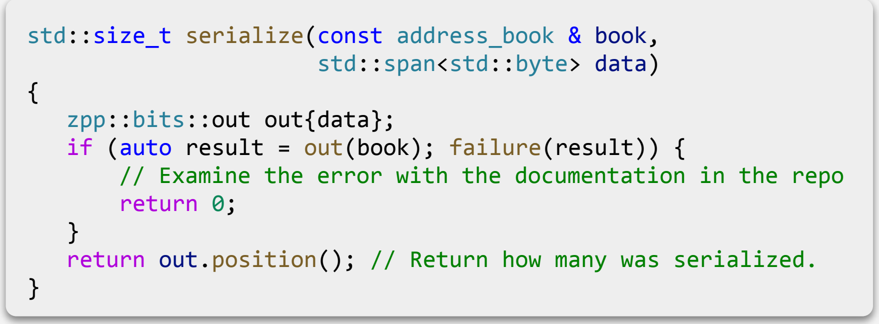
```cpp
std::size_t serialize(const address_book & book,
                      std::span<std::byte> data)
{
   zpp::bits::out out{data};
   if (auto result = out(book); failure(result)) {
       // Examine the error with the documentation in the repo
       return 0;
   }
   return out.position(); // Return how many was serialized.
}
```

```cpp
std::size_t deserialize(address_book & book,
                        std::span<const std::byte> data)
{







}
```

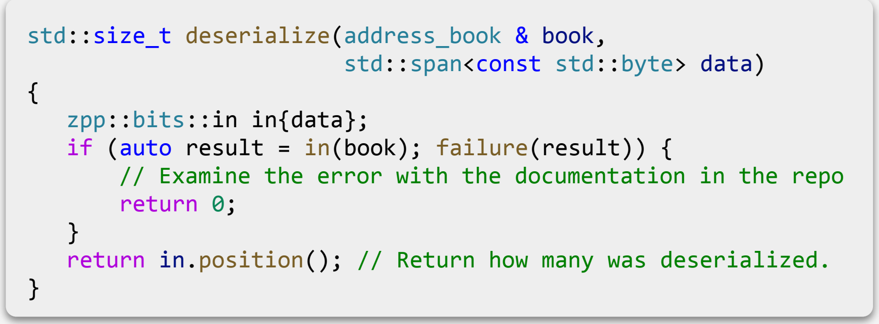
Serialization Example with zpp::bits



```cpp
struct address_book
{
    enum class phone_type : int
    {
        mobile = 0,
        home = 1,
        work = 2,
    };

    struct phone_number
    {
        std::string number;
        phone_type type;
    };

    struct person
    {
        std::string name;
        int id;
        std::string email;
        std::vector<phone_number> phones;
    };

    std::vector<person> people;
};
```



```cpp
std::size_t serialize(const address_book & book,
                      std::span<std::byte> data)
{
   zpp::bits::out out{data};
   if (auto result = out(book); failure(result)) {
       // Examine the error with the documentation in the repo
       return 0;
   }
   return out.position(); // Return how many was serialized.
}
```



```cpp
std::size_t deserialize(address_book & book,
                        std::span<const std::byte> data)
{
   zpp::bits::in in{data};
   if (auto result = in(book); failure(result)) {
       // Examine the error with the documentation in the repo
       return 0;
   }
   return in.position(); // Return how many was deserialized.
}
```

Let’s Compile

```
std::size_t serialize(const address_book & book,
                      std::span<std::byte> data)
{
   zpp::bits::out out{data};
   if (auto result = out(book); failure(result)) {
       // Examine the error with the documentation in the repo
       return 0;
   }
   return out.position(); // Return how many was serialized.
}
```

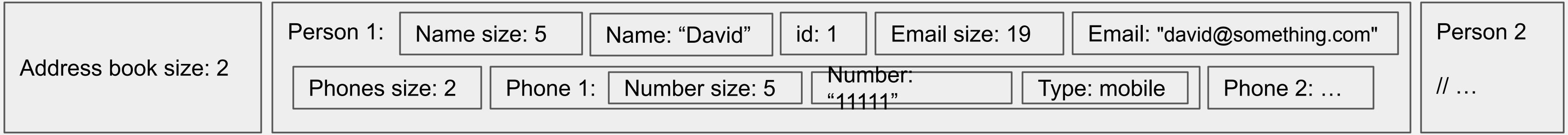
```
std::size_t deserialize(address_book & book,
                        std::span<const std::byte> data)
{
   zpp::bits::in in{data};
   if (auto result = in(book); failure(result)) {
       // Examine the error with the documentation in the repo
       return 0;
   }
   return in.position(); // Return how many was deserialized.
}
```

Link

The Serialization Format of zpp::bits

```cpp
struct address_book
{
    enum class phone_type : int
    {
        mobile = 0, home = 1, work = 2,
    };

    struct phone_number
    {
        std::string number;
        phone_type type;
    };

    struct person
    {
        std::string name;
        int id;
        std::string email;
        std::vector<phone_number> phones;
    };

    std::vector<person> people;
};
```

```cpp
address_book book = {{

       {.name = "David",
        .id = 1,
        .email = "david@something.com",
        .phones = {{.number = "11111",
                    .type = address_book::phone_type::mobile},
                   {.number = "22222",
                    .type = address_book::phone_type::home}}},

       {.name = "Jane",
        .id = 2,
        .email = "jane@something.com",
        .phones = {{.number = "33333",
                    .type = address_book::phone_type::mobile},
                   {.number = "44444",
                    .type = address_book::phone_type::work}}},

}};
```



Address book size: 2

Person 1:

Name size: 5

Name: “David”

id: 1

Email size: 19

Email: "david@something.com"

Phones size: 2

Phone 1:

Number size: 5

Number: “11111”

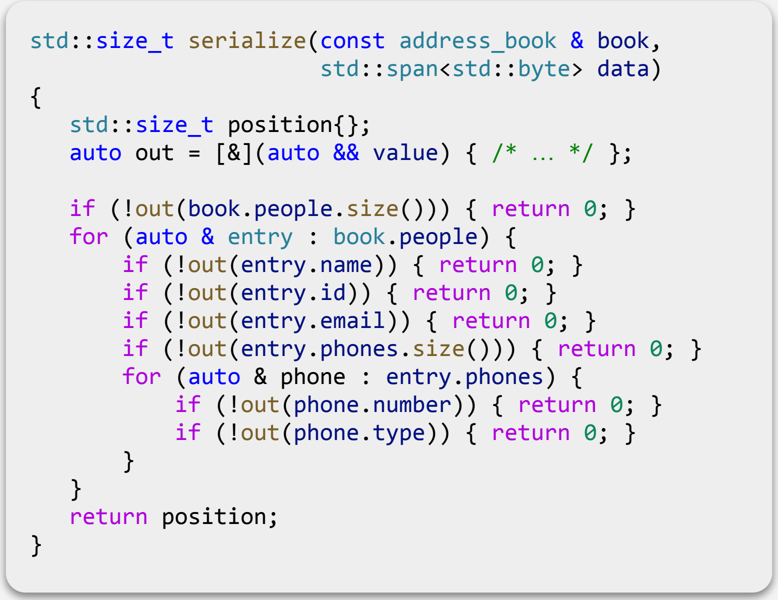
Type: mobile

Phone 2: …

Person 2

// …

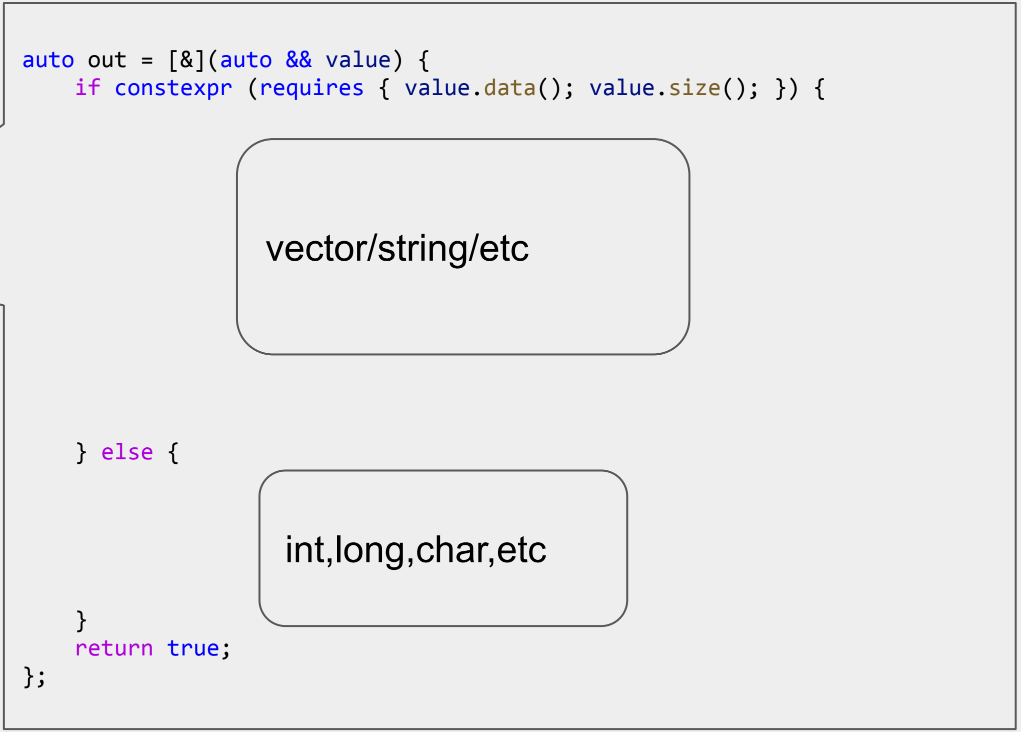
The Hand Written Code



```cpp
std::size_t serialize(const address_book & book,
                      std::span<std::byte> data)
{
   std::size_t position{};
   auto out = [&](auto && value) { /* … */ };

   if (!out(book.people.size())) { return 0; }
   for (auto & entry : book.people) {
       if (!out(entry.name)) { return 0; }
       if (!out(entry.id)) { return 0; }
       if (!out(entry.email)) { return 0; }
       if (!out(entry.phones.size())) { return 0; }
       for (auto & phone : entry.phones) {
           if (!out(phone.number)) { return 0; }
           if (!out(phone.type)) { return 0; }
       }
   }
   return position;
}
```

The Hand Written Code

```cpp
std::size_t serialize(const address_book & book,
                      std::span<std::byte> data)
{
   std::size_t position{};
   auto out = [&](auto && value) { /* … */ };

   if (!out(book.people.size())) { return 0; }
   for (auto & entry : book.people) {
       if (!out(entry.name)) { return 0; }
       if (!out(entry.id)) { return 0; }
       if (!out(entry.email)) { return 0; }
       if (!out(entry.phones.size())) { return 0; }
       for (auto & phone : entry.phones) {
           if (!out(phone.number)) { return 0; }
           if (!out(phone.type)) { return 0; }
       }
   }
   return position;
}
```



```cpp
auto out = [&](auto && value) {
    if constexpr (requires { value.data(); value.size(); }) {

    } else {

    }
    return true;
};
```

The Hand Written Code

```
std::size_t serialize(const address_book & book,
                      std::span<std::byte> data)
{
   std::size_t position{};
   auto out = [&](auto && value) { /* … */ };

   if (!out(book.people.size())) { return 0; }
   for (auto & entry : book.people) {
       if (!out(entry.name)) { return 0; }
       if (!out(entry.id)) { return 0; }
       if (!out(entry.email)) { return 0; }
       if (!out(entry.phones.size())) { return 0; }
       for (auto & phone : entry.phones) {
           if (!out(phone.number)) { return 0; }
           if (!out(phone.type)) { return 0; }
       }
   }
   return position;
}
```

```
auto out = [&](auto && value) {
    if constexpr (requires { value.data(); value.size(); }) {
```

vector/string/etc

```
    } else {
        if (sizeof(value) > data.size() - position) {
            return false;
        }
        std::memcpy(data.data() + position, &value, sizeof(value));
        position += sizeof(value);
    }
    return true;
};
```

The Hand Written Code

```cpp
std::size_t serialize(const address_book & book,
                      std::span<std::byte> data)
{
   std::size_t position{};
   auto out = [&](auto && value) { /* … */ };

   if (!out(book.people.size())) { return 0; }
   for (auto & entry : book.people) {
       if (!out(entry.name)) { return 0; }
       if (!out(entry.id)) { return 0; }
       if (!out(entry.email)) { return 0; }
       if (!out(entry.phones.size())) { return 0; }
       for (auto & phone : entry.phones) {
           if (!out(phone.number)) { return 0; }
           if (!out(phone.type)) { return 0; }
       }
   }
   return position;
}
```

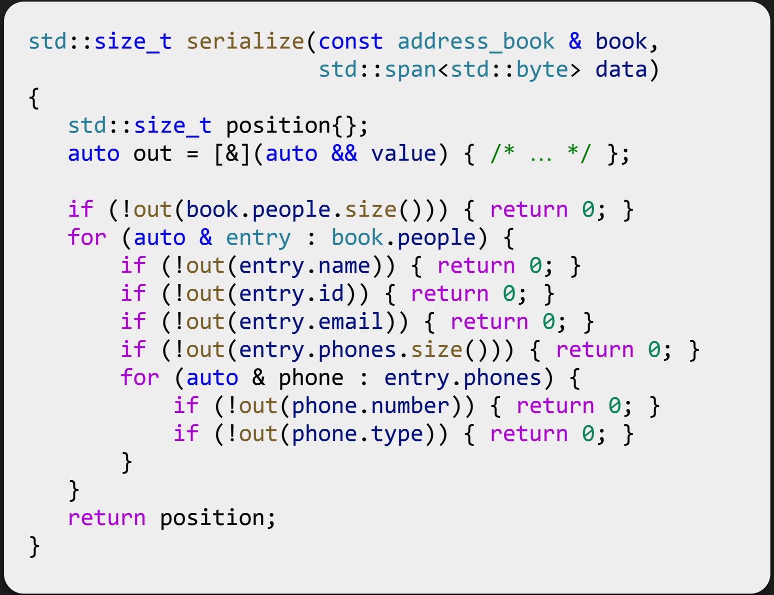
```cpp
auto out = [&](auto && value) {
    if constexpr (requires { value.data(); value.size(); }) {
        auto size = value.size();
        if (sizeof(size) > data.size() - position) {
            return false;
        }
        std::memcpy(data.data() + position, &size, sizeof(size));
        position += sizeof(size);





    } else {
        if (sizeof(value) > data.size() - position) {
            return false;
        }
        std::memcpy(data.data() + position, &value, sizeof(value));
        position += sizeof(value);
    }
    return true;
};
```

The Hand Written Code

```cpp
std::size_t serialize(const address_book & book,
                      std::span<std::byte> data)
{
   std::size_t position{};
   auto out = [&](auto && value) { /* … */ };

   if (!out(book.people.size())) { return 0; }
   for (auto & entry : book.people) {
       if (!out(entry.name)) { return 0; }
       if (!out(entry.id)) { return 0; }
       if (!out(entry.email)) { return 0; }
       if (!out(entry.phones.size())) { return 0; }
       for (auto & phone : entry.phones) {
           if (!out(phone.number)) { return 0; }
           if (!out(phone.type)) { return 0; }
       }
   }
   return position;
}
```

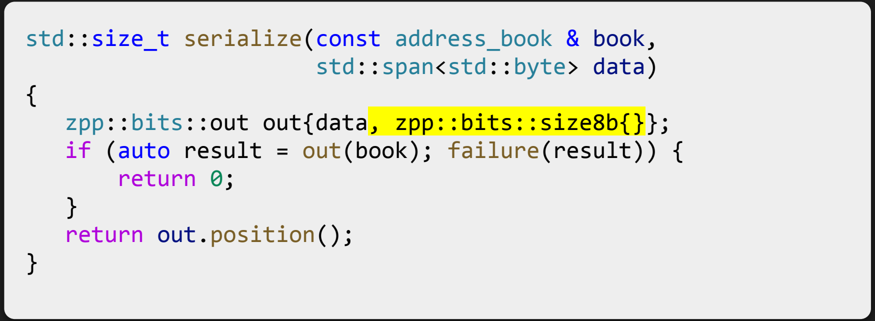
```cpp
auto out = [&](auto && value) {
    if constexpr (requires { value.data(); value.size(); }) {
        auto size = value.size();
        if (sizeof(size) > data.size() - position) {
            return false;
        }
        std::memcpy(data.data() + position, &size, sizeof(size));
        position += sizeof(size);
        auto size_in_bytes = size * sizeof(*value.data());
        if (size_in_bytes > data.size() - position) {
            return false;
        }
        std::memcpy(data.data() + position, value.data(), size_in_bytes);
        position += size_in_bytes;
    } else {
        if (sizeof(value) > data.size() - position) {
            return false;
        }
        std::memcpy(data.data() + position, &value, sizeof(value));
        position += sizeof(value);
    }
    return true;
};
```

Let’s Compile – Handwritten vs zpp::bits



```cpp
std::size_t serialize(const address_book & book,
                      std::span<std::byte> data)
{
   std::size_t position{};
   auto out = [&](auto && value) { /* … */ };

   if (!out(book.people.size())) { return 0; }
   for (auto & entry : book.people) {
       if (!out(entry.name)) { return 0; }
       if (!out(entry.id)) { return 0; }
       if (!out(entry.email)) { return 0; }
       if (!out(entry.phones.size())) { return 0; }
       for (auto & phone : entry.phones) {
           if (!out(phone.number)) { return 0; }
           if (!out(phone.type)) { return 0; }
       }
   }
   return position;
}
```

VS



```cpp
std::size_t serialize(const address_book & book,
                      std::span<std::byte> data)
{
   zpp::bits::out out{data, zpp::bits::size8b{}};
   if (auto result = out(book); failure(result)) {
       return 0;
   }
   return out.position();
}
```

Link

The Zero Overhead Toolbox that zpp::bits uses

```
std::size_t serialize(const address_book & book,
                      std::span<std::byte> data)
{
   zpp::bits::out out{data};
   if (auto result = out(book); failure(result)) {
       return 0;
   }
   return out.position();
}
```

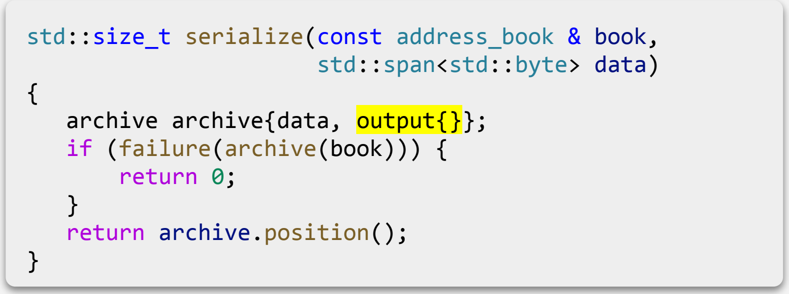
```
std::size_t deserialize(address_book & book,
                        std::span<const std::byte> data)
{
   zpp::bits::in in{data};
   if (auto result = in(book); failure(result)) {
       return 0;
   }
   return in.position();
}
```

1. Make everything available for inline – header only.
   Tip: Almost free to make everything constexpr as well.
2. No virtual functions – to maximize inlining
3. Use concepts and templates to be fully generic – For example, "data" does not have to be "std::span", can be a simple array, or std::array, or similar.
4. Customize & optimize by "if constexpr" –
   > Use memcpy where it would iterate byte by byte.
   > Serialization format determined in compile time
   > If a growing output buffer is needed, use std::vector or std::string which has "resize(..)". An "if constexpr" will detect it and use as needed.
5. Zero overhead error handling, with alternatives –
   Return values and then offer error checking alternatives for external usage:

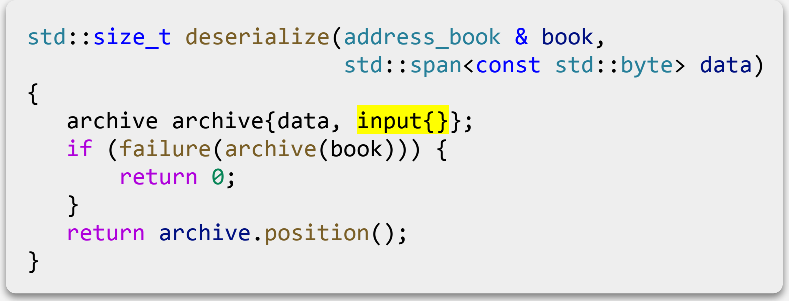
```
zpp::bits::out out{data};
out(data).or_throw();
return out.position();
```

```
zpp::bits::out out{data};
co_await out(data);
co_return out.position();
```

Lets Implement our own C++ Serializer –



```cpp
std::size_t serialize(const address_book & book,
                      std::span<std::byte> data)
{
   archive archive{data, output{}};
   if (failure(archive(book))) {
       return 0;
   }
   return archive.position();
}
```



```cpp
std::size_t deserialize(address_book & book,
                        std::span<const std::byte> data)
{
   archive archive{data, input{}};
   if (failure(archive(book))) {
       return 0;
   }
   return archive.position();
}
```

```cpp
struct address_book
{
   enum class phone_type : int
   {
       mobile = 0,
       home = 1,
       work = 2,
   };

   struct phone_number
   {
       std::string number;
       phone_type type;
   };

   struct person
   {
       std::string name;
       int id;
       std::string email;
       std::vector<phone_number> phones;
   };

   std::vector<person> people;
};
```

→

```cpp
struct address_book
{
   enum class phone_type : int { mobile, home, work, };

   struct phone_number {
       static auto serialize(auto & archive, auto & self) {
           return archive(self.number, self.type);
       }

       std::string number;
       phone_type type;
   };

   struct person {
       static auto serialize(auto & archive, auto & self) {
           return archive(self.name, self.id, self.email, self.phones);
       }

       std::string name;
       int id;
       std::string email;
       std::vector<phone_number> phones;
   };

   static auto serialize(auto & archive, auto & self) {
       return archive(self.people);
   }

   std::vector<person> people;
};
```

Our own C++ Serializer – The archive class

```
std::size_t serialize(const address_book & book,
                      std::span<std::byte> data)
{
   archive archive{data, output{}};
   if (failure(archive(book))) {
       return 0;
   }
   return archive.position();
}
```

```
std::size_t deserialize(address_book & book,
                        std::span<const std::byte> data)
{
   archive archive{data, input{}};
   if (failure(archive(book))) {
       return 0;
   }
   return archive.position();
}
```

```
template <typename ByteView, typename Kind>
struct archive
{
   // Our archive class is both input and output, depending on “Kind”

   ByteView & m_data;
   std::size_t m_position{};
};
```

Our own C++ Serializer – The archive class

```cpp
std::size_t serialize(const address_book & book,
                      std::span<std::byte> data)
{
   archive archive{data, output{}};
   if (failure(archive(book))) {
       return 0;
   }
   return archive.position();
}
```

```cpp
std::size_t deserialize(address_book & book,
                        std::span<const std::byte> data)
{
   archive archive{data, input{}};
   if (failure(archive(book))) {
       return 0;
   }
   return archive.position();
}
```

```cpp
template <typename ByteView, typename Kind>
struct archive
{
   explicit archive(ByteView & data, Kind) :
       m_data(data) {}

   ByteView & m_data;
   std::size_t m_position{};
};
```

Our own C++ Serializer – The archive class

```cpp
std::size_t serialize(const address_book & book,
                      std::span<std::byte> data)
{
   archive archive{data, output{}};
   if (failure(archive(book))) {
       return 0;
   }
   return archive.position();
}
```

```cpp
std::size_t deserialize(address_book & book,
                        std::span<const std::byte> data)
{
   archive archive{data, input{}};
   if (failure(archive(book))) {
       return 0;
   }
   return archive.position();
}
```

```cpp
template <typename ByteView, typename Kind>
struct archive
{
   explicit archive(ByteView & data, Kind) :
       m_data(data) {}

   auto position() const { return m_position{}; }


   ByteView & m_data;
   std::size_t m_position{};
};
```

Our own C++ Serializer – The archive class

```
std::size_t serialize(const address_book & book,
                      std::span<std::byte> data)
{
   archive archive{data, output{}};
   if (failure(archive(book))) {
       return 0;
   }
   return archive.position();
}
```

```
std::size_t deserialize(address_book & book,
                        std::span<const std::byte> data)
{
   archive archive{data, input{}};
   if (failure(archive(book))) {
       return 0;
   }
   return archive.position();
}
```

```
template <typename ByteView, typename Kind>
struct archive
{
   explicit archive(ByteView & data, Kind) :
       m_data(data) {}

   auto position() const { return m_position{}; }

   auto operator()(auto && ... objects)
   {
       return serialize_many(objects...);
   }









   ByteView & m_data;
   std::size_t m_position{};
};
```

Our own C++ Serializer – The archive class

```cpp
std::size_t serialize(const address_book & book,
                      std::span<std::byte> data)
{
   archive archive{data, output{}};
   if (failure(archive(book))) {
       return 0;
   }
   return archive.position();
}
```

```cpp
std::size_t deserialize(address_book & book,
                        std::span<const std::byte> data)
{
   archive archive{data, input{}};
   if (failure(archive(book))) {
       return 0;
   }
   return archive.position();
}
```

```cpp
template <typename ByteView, typename Kind>
struct archive
{
   explicit archive(ByteView & data, Kind) :
       m_data(data) {}

   auto position() const { return m_position{}; }

   auto operator()(auto && ... objects)
   {
       return serialize_many(objects...);
   }

   auto serialize_many(auto & first,
                       auto && ... remains)
   {
       if (auto result = serialize_one(first); failure(result)) {
           return result;
       }

       return serialize_many(remains...);
   }




   ByteView & m_data;
   std::size_t m_position{};
};
```

Our own C++ Serializer – The archive class

```cpp
std::size_t serialize(const address_book & book,
                      std::span<std::byte> data)
{
   archive archive{data, output{}};
   if (failure(archive(book))) {
       return 0;
   }
   return archive.position();
}
```

```cpp
std::size_t deserialize(address_book & book,
                        std::span<const std::byte> data)
{
   archive archive{data, input{}};
   if (failure(archive(book))) {
       return 0;
   }
   return archive.position();
}
```

```cpp
template <typename ByteView, typename Kind>
struct archive
{
   explicit archive(ByteView & data, Kind) :
       m_data(data) {}

   auto position() const { return m_position{}; }

   auto operator()(auto && ... objects)
   {
       return serialize_many(objects...);
   }

   auto serialize_many(auto & first,
                       auto && ... remains)
   {
       if (auto result = serialize_one(first); failure(result)) {
           return result;
       }

       return serialize_many(remains...);
   }

   auto serialize_many() { return errc{}; }
   auto serialize_one(auto && object) { /* ... */ }

   ByteView & m_data;
   std::size_t m_position{};
};
```

Our own C++ Serializer – “serialize_one”

```cpp
auto serialize_one(auto && object)
{
    using type = std::remove_cvref_t<decltype(object)>;

    if constexpr (std::is_fundamental_v<type> || std::is_enum_v<type>) {
        return serialize_bytes_of(object);

    }

    // ...

}
```

```cpp
auto serialize_bytes_of(auto && object)
{
    if (sizeof(object) > m_data.size() - m_position) {
        return errc::result_out_of_range;
    }

    if constexpr (std::same_as<Kind, output>) {
        std::memcpy(m_data.data() + m_position,
                    &object,
                    sizeof(object));

    } else if constexpr (std::same_as<Kind, input>) {
        std::memcpy(&object,
                    m_data.data() + m_position,
                    sizeof(object));

    }
    m_position += sizeof(object);
    return errc{};
}
```

Our own C++ Serializer – “serialize_one”

```cpp
auto serialize_one(auto && object)

{
   using type = std::remove_cvref_t<decltype(object)>;

   if constexpr (std::is_fundamental_v<type> || std::is_enum_v<type>) {
       return serialize_bytes_of(object);

   } else if constexpr (requires { type{}.data(); type{}.size(); }) {
       auto size = object.size();
       if (auto result = serialize_one(size); failure(result)) {
           return result;
       }
       if constexpr (std::same_as<Kind, input>) {
           object.resize(size);
       }
       for (auto & element : object) {
           if (auto result = serialize_one(element); failure(result)) {
               return result;
           }
       }
       return errc{};

   }

   // …
}
```

```cpp
auto serialize_bytes_of(auto && object)
{
    if (sizeof(object) > m_data.size() - m_position) {
        return errc::result_out_of_range;
    }

    if constexpr (std::same_as<Kind, output>) {
        std::memcpy(m_data.data() + m_position,
                    &object,
                    sizeof(object));

    } else if constexpr (std::same_as<Kind, input>) {
        std::memcpy(&object,
                    m_data.data() + m_position,
                    sizeof(object));

    }
    m_position += sizeof(object);
    return errc{};
}
```

Our own C++ Serializer – “serialize_one”

```cpp
auto serialize_one(auto && object)
{
    using type = std::remove_cvref_t<decltype(object)>;

    if constexpr (std::is_fundamental_v<type> || std::is_enum_v<type>) {
        return serialize_bytes_of(object);

    } else if constexpr (requires { type{}.data(); type{}.size(); }) {
        auto size = object.size();
        if (auto result = serialize_one(size); failure(result)) {
            return result;
        }
        if constexpr (std::same_as<Kind, input>) {
            object.resize(size);
        }
        for (auto & element : object) {
            if (auto result = serialize_one(element); failure(result)) {
                return result;
            }
        }
        return errc{};

    } else if constexpr (requires { type::serialize(*this, object); }) {
        return type::serialize(*this, object);

    } else {
        static_assert(std::is_void_v<type>, "Currently Unsupported");
    }
}
```

```cpp
auto serialize_bytes_of(auto && object)
{
    if (sizeof(object) > m_data.size() - m_position) {
        return errc::result_out_of_range;
    }

    if constexpr (std::same_as<Kind, output>) {
        std::memcpy(m_data.data() + m_position,
                    &object,
                    sizeof(object));

    } else if constexpr (std::same_as<Kind, input>) {
        std::memcpy(&object,
                    m_data.data() + m_position,
                    sizeof(object));

    }
    m_position += sizeof(object);
    return errc{};
}
```

Let’s Compile – Handwritten vs Our First Serializer

```cpp
std::size_t serialize(const address_book & book,
                      std::span<std::byte> data)
{
   std::size_t position{};
   auto out = [&](auto && value) { /* … */ };

   if (!out(book.people.size())) { return 0; }
   for (auto & entry : book.people) {
       if (!out(entry.name)) { return 0; }
       if (!out(entry.id)) { return 0; }
       if (!out(entry.email)) { return 0; }
       if (!out(entry.phones.size())) { return 0; }
       for (auto & phone : entry.phones) {
           if (!out(phone.number)) { return 0; }
           if (!out(phone.type)) { return 0; }
       }
   }
   return position;
}
```

VS

Link

```cpp
std::size_t serialize(const address_book & book,
                      std::span<std::byte> data)
{
   archive archive{data, output{}};
   if (failure(archive(book))) {
       return 0;
   }
   return archive.position();
}
```

```cpp
std::size_t deserialize(address_book & book,
                        std::span<const std::byte> data)
{
   archive archive{data, input{}};
   if (failure(archive(book))) {
       return 0;
   }
   return archive.position();
}
```

Let’s Compile - Handwritten vs Our First Serializer

This time with **always_inline/__forceinline**

```cpp
std::size_t serialize(const address_book & book,
                      std::span<std::byte> data)
{
    std::size_t position{};
    auto out = [&](auto && value) { /* … */ };

    if (!out(book.people.size())) { return 0; }
    for (auto & entry : book.people) {
        if (!out(entry.name)) { return 0; }
        if (!out(entry.id)) { return 0; }
        if (!out(entry.email)) { return 0; }
        if (!out(entry.phones.size())) { return 0; }
        for (auto & phone : entry.phones) {
            if (!out(phone.number)) { return 0; }
            if (!out(phone.type)) { return 0; }
        }
    }
    return position;
}
```

VS

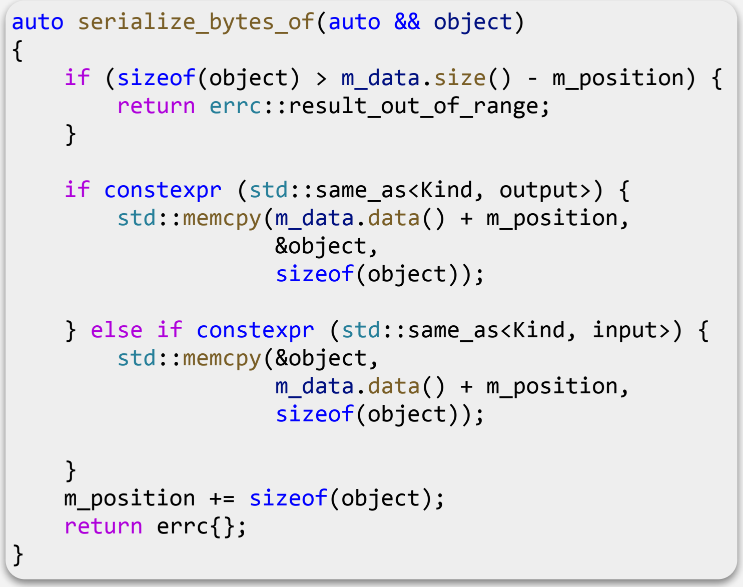
[Link]

```cpp
std::size_t serialize(const address_book & book,
                      std::span<std::byte> data)
{
    archive archive{data, output{}};
    if (failure(archive(book))) {
        return 0;
    }
    return archive.position();
}
```

```cpp
std::size_t deserialize(address_book & book,
                        std::span<const std::byte> data)
{
    archive archive{data, input{}};
    if (failure(archive(book))) {
        return 0;
    }
    return archive.position();
}
```

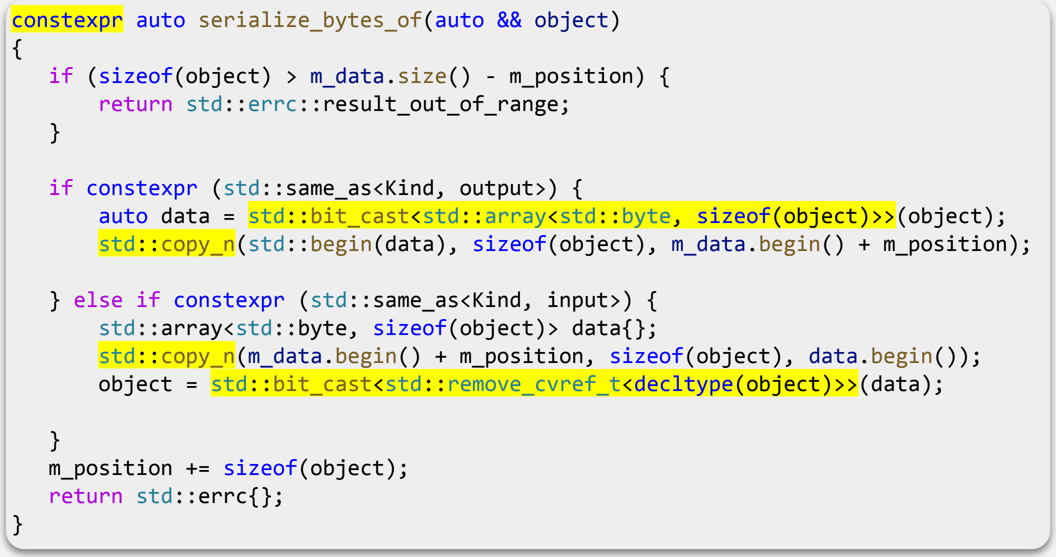
Note: this is non-standard, and some compilers complain in case there is a circular serialization - zpp::bits works around it by implementing a “self_referencing” concept in which case the dependency is killed.

Constexpr everything



```cpp
auto serialize_bytes_of(auto && object)
{
    if (sizeof(object) > m_data.size() - m_position) {
        return errc::result_out_of_range;
    }

    if constexpr (std::same_as<Kind, output>) {
        std::memcpy(m_data.data() + m_position,
                    &object,
                    sizeof(object));

    } else if constexpr (std::same_as<Kind, input>) {
        std::memcpy(&object,
                    m_data.data() + m_position,
                    sizeof(object));

    }
    m_position += sizeof(object);
    return errc{};
}
```

→

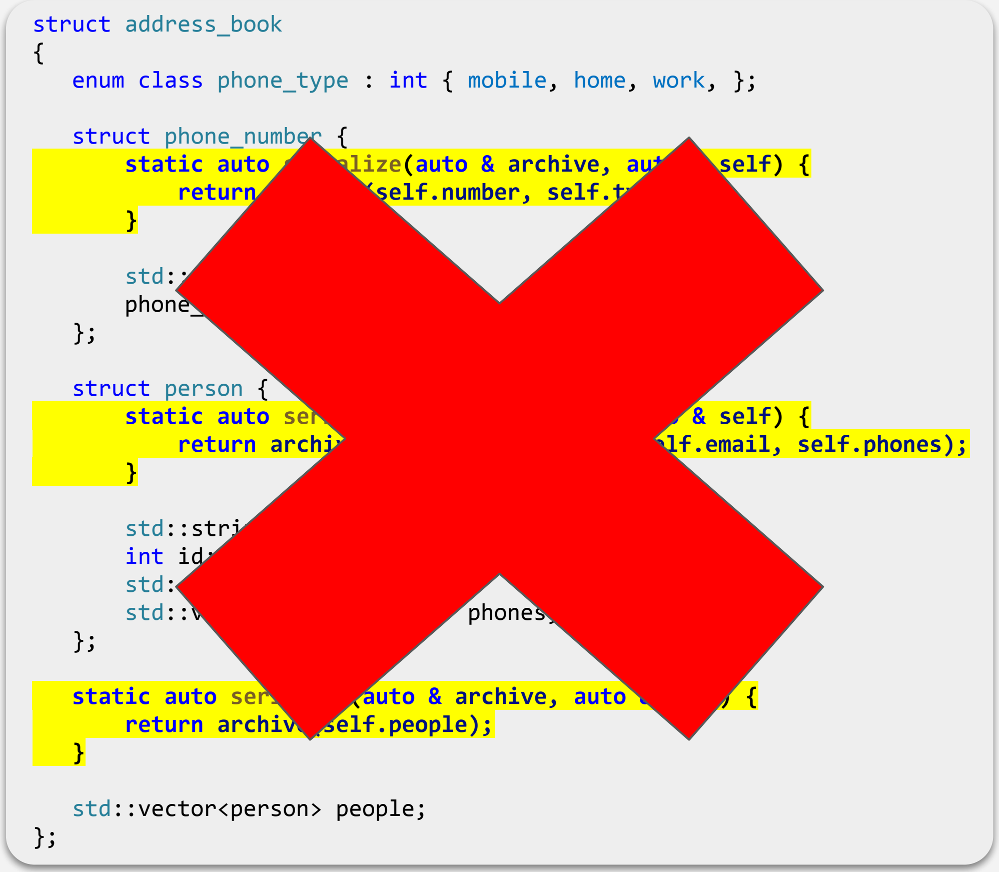


```cpp
constexpr auto serialize_bytes_of(auto && object)
{
   if (sizeof(object) > m_data.size() - m_position) {
       return std::errc::result_out_of_range;
   }

   if constexpr (std::same_as<Kind, output>) {
       auto data = std::bit_cast<std::array<std::byte, sizeof(object)>>(object);
       std::copy_n(std::begin(data), sizeof(object), m_data.begin() + m_position);

   } else if constexpr (std::same_as<Kind, input>) {
       std::array<std::byte, sizeof(object)> data{};
       std::copy_n(m_data.begin() + m_position, sizeof(object), data.begin());
       object = std::bit_cast<std::remove_cvref_t<decltype(object)>>(data);

   }
   m_position += sizeof(object);
   return std::errc{};
}
```

Link

Reflection in “The Pre Reflection” Era

```
struct address_book
{
    enum class phone_type : int
    {
        mobile = 0,
        home = 1,
        work = 2,
    };

    struct phone_number
    {
        std::string number;
        phone_type type;
    };

    struct person
    {
        std::string name;
        int id;
        std::string email;
        std::vector<phone_number> phones;
    };

    std::vector<person> people;
};
```



```
struct address_book
{
    enum class phone_type : int { mobile, home, work, };

    struct phone_number {
        static auto          lize(auto & archive, aut     self) {
            return          (self.number, self.t
        }

        std::
        phone_
    };

    struct person {
        static auto ser                               & self) {
            return archi                        lf.email, self.phones);
        }

        std::stri
        int id:
        std:
        std::v                  phones;
    };

    static auto ser       (auto & archive, auto c     ) {
        return archiv self.people);
    }

    std::vector<person> people;
};
```

Reflection in “The Pre Reflection” Era

```cpp
struct address_book
{
   enum class phone_type : int
   {
       mobile = 0,
       home = 1,
       work = 2,
   };

   struct phone
   {
       std::string number;
       phone_type type;
   };

   struct person
   {
       std::string name;
       int id;
       std::string email;
       std::vector<phone_number> phones;
   };

   std::vector<person> people;
};
```

```cpp
address_book address_book = {/*...*/};
address_book::person person = {/*...*/};
address_book::phone phone = {/*...*/};
```

```cpp
auto & [people] = address_book;
auto & [name, id, email, phones] = person;
auto & [number, type] = phone;
```

```cpp
auto & [m1] = <1-member>;
auto & [m1, m2] = <2-members>;
auto & [m1, m2, m3] = <3-members>;
```

```cpp
auto & [... m] = <sizeof...(m)-members>;
```

Not available currently. Proposed by P1061 – “Structured Bindings can introduce a Pack”

```cpp
auto serialize_one(auto && object)
{
   using type = std::remove_cvref_t<decltype(object)>;

   if constexpr (std::is_fundamental_v<type> || std::is_enum_v<type>) {
      // Copy the bytes of the fundamental/enumeration type

   } else if constexpr (requires { type{}.data(); type{}.resize(1); }) {
      // Serialize [Size][Elements…]

   } else if constexpr (requires { type::serialize(*this, object); }) {
      // Recursive call the serialize of the type with our archive
   } else {
      auto & [... members] = object;
      return serialize_many(members...);
   }
}
```

Reflection in “The Pre Reflection” Era

```
auto & [... members] = object;
return serialize_many(members...);
```

→

```
return visit_members(object, [&](auto & ... members) {
    return serialize_many(members...);
});
```

```
decltype(auto) visit_members(auto && object, auto && visitor)
{
   constexpr auto count = number_of_members<decltype(object)>();

   if      constexpr (count == 0) {                                   return visitor(); }
   else if constexpr (count == 1) { auto & [a1]             = object; return visitor(a1); }
   else if constexpr (count == 2) { auto & [a1, a2]         = object; return visitor(a1, a2); }
   else if constexpr (count == 3) { auto & [a1, a2, a3]     = object; return visitor(a1, a2, a3); }
   /*...*/
   else { static_assert(std::is_void_v<decltype(object)>, "visit_members: maximum reached."); }
}
```

```
template <typename Type>
constexpr std::size_t number_of_members()
{
   if      constexpr (requires { requires std::is_empty_v<Type> && sizeof(Type); })      { return 0; }
   else if constexpr (requires { [](Type & object) { auto & [a1]         = object; }; }) { return 1; }
   else if constexpr (requires { [](Type & object) { auto & [a1, a2]     = object; }; }) { return 2; }
   else if constexpr (requires { [](Type & object) { auto & [a1, a2, a3] = object; }; }) { return 3; }
   /*...*/
   else { static_assert(std::is_void_v<Type>, "number_of_members: maximum reached."); }
}
```

Error: The lambda implementation is not part of the "immediate context" that is required for a SFINAE error.

Reflection in “The Pre Reflection” Era

```
decltype(auto) visit_members(auto && object, auto && visitor)
{
   constexpr auto count = number_of_members<decltype(object)>();

   if      constexpr (count == 0) {                                   return visitor(); }
   else if constexpr (count == 1) { auto & [a1]             = object; return visitor(a1); }
   else if constexpr (count == 2) { auto & [a1, a2]         = object; return visitor(a1, a2); }
   else if constexpr (count == 3) { auto & [a1, a2, a3]     = object; return visitor(a1, a2, a3); }
   /*...*/
   else { static_assert(std::is_void_v<decltype(object)>, "visit_members: maximum reached."); }
}
```

```
struct any { template <typename Type> operator Type(); };
```

```
template <typename Type>
constexpr std::size_t number_of_members()
{
   if      constexpr (requires { requires std::is_empty_v<Type> && sizeof(Type); }) { return 0; }
   else if constexpr (requires { Type{any{}, any{}, any{}, /*...*/, any{}}; })      { return /*max*/; }
   /*...*/
   else if constexpr (requires { Type{any{}, any{}, any{}}; })                      { return 3; }
   else if constexpr (requires { Type{any{}, any{}}; })                             { return 2; }
   else if constexpr (requires { Type{any{}}; })                                    { return 1; }
   /*...*/
   else { static_assert(std::is_void_v<Type>, "number_of_members: maximum reached."); }
}
```

Count members using aggregate initialization – **less accurate and more restricted**

Reflection in “The Pre Reflection” Era

```
decltype(auto) visit_members(auto && object, auto && visitor)
{
   constexpr auto count = number_of_members<decltype(object)>();

   if        constexpr (count == 0) {                                  return visitor(); }
   else if constexpr (count == 1) { auto & [a1]             = object; return visito
   else if constexpr (count == 2) { auto & [a1, a2]         = object; return visito
   else if constexpr (count == 3) { auto & [a1, a2, a3]     = object; return visito
   /*...*/
   else { static_assert(std::is_void_v<decltype(object)>, "visit_members: maximum r
}
```

```
struct any { template <typename Type> operator Type(); };
```

```
template <typename Type>
constexpr std::size_t number_of_members()
{
   if        constexpr (requires { requires std::is_empty_v<Type> && sizeof(Type); }) { return 0;
   else if constexpr (requires { Type::serialize::members; }) { return Type::serialize::members; }
   else if constexpr (requires { Type{any{}, any{}, any{}, /*...*/, any{}}; })      { return /*max*/; }
   /*...*/
   else if constexpr (requires { Type{any{}, any{}, any{}}; })                        { return 3; }
   else if constexpr (requires { Type{any{}, any{}}; })                               { return 2; }
   else if constexpr (requires { Type{any{}}; })                                      { return 1; }
   /*...*/
   else { static_assert(std::is_void_v<Type>, "number_of_members: maximum reached."); }
}
```

Allow setting number of members manually:

```
struct phone_number
{
   std::string number;
   phone_type type;

   using serialize = zpp::bits::members<2>;
};
```

Count members using aggregate initialization – **less accurate and more restricted**

Remote Procedure Call – First Attempt

```
struct add_numbers
{
  auto operator()() const
  {
     return (x + y);
  }

  int x;
  int y;
};

struct multiply_numbers
{
  auto operator()() const
  {
     return (x * y);
  }

  int x;
  int y;
};
```

```
using rpc = std::variant<add_numbers,
                         multiply_numbers,
                         print_hello,
                         do_something,
                         do_something_else>;
```

```
// Server receives data from the client

// Server reads the request
rpc request;
in(request).or_throw();

// Server executes the request, and outputs the response
out(std::visit([](auto && request) { return request(); }, request)).or_throw();
```

```
// Client receives data from the server

// Client reads the result and prints it
int result;
in(result).or_throw();
```

Request: Variant Index : 0 | add_numbers : [x : 1], [y : 2]

Response: x + y : 3

Remote Procedure Call - Enhancing The Syntax

```cpp
auto add_numbers(int x, int y)
{
    return x + y;
}

auto multiply_numbers(int x, int y)
{
    return x * y;
}
```

```cpp
using rpc = zpp::bits::rpc<
   zpp::bits::bind<add_numbers, 0>,
   zpp::bits::bind<multiply_numbers, 1>
>;

// Client request
rpc::client client{in, out};
client.request<0>(1, 2).or_throw();

// Client transports data to server
```

```cpp
// Server receives data from the client

// Server reads and executes the request
rpc::server server{in, out};
server.serve().or_throw();

// Server transports data to the client
```

```cpp
// Client receives data from the server

// Client reads the result and prints it
auto response = client.response<0>(1, 2).or_throw();
```

Request: Id : 0 | add_numbers : [x : 1], [y : 2]

Response: x + y : 3

Remote Procedure Call - Enhancing The Syntax

```cpp
auto add_numbers(int x, int y)
{
    return x + y;
}

auto multiply_numbers(int x, int y)
{
    return x * y;
}
```

```cpp
using rpc = zpp::bits::rpc<
   zpp::bits::bind<add_numbers, "add_numbers"_sha256_int>,
   zpp::bits::bind<multiply_numbers, "multiply_numbers"_sha256_int>
>;

// Client request
rpc::client client{in, out};
client.request<"add_numbers"_sha256_int>(1, 2).or_throw();

// Client transports data to server
```

```cpp
// Server receives data from the client

// Server reads and executes the request
rpc::server server{in, out};
server.serve().or_throw();

// Server transports data to the client
```

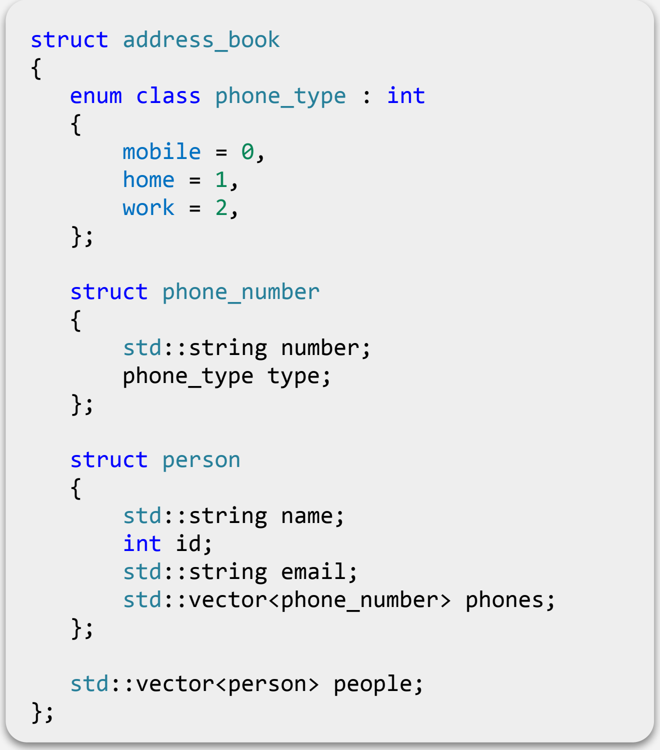
```cpp
// Client receives data from the server

// Client reads the result and prints it
auto response = client.response<"add_numbers"_sha256_int>(1, 2).or_throw();
```

Request: Id : <hash> add_numbers : [x : 1], [y : 2]

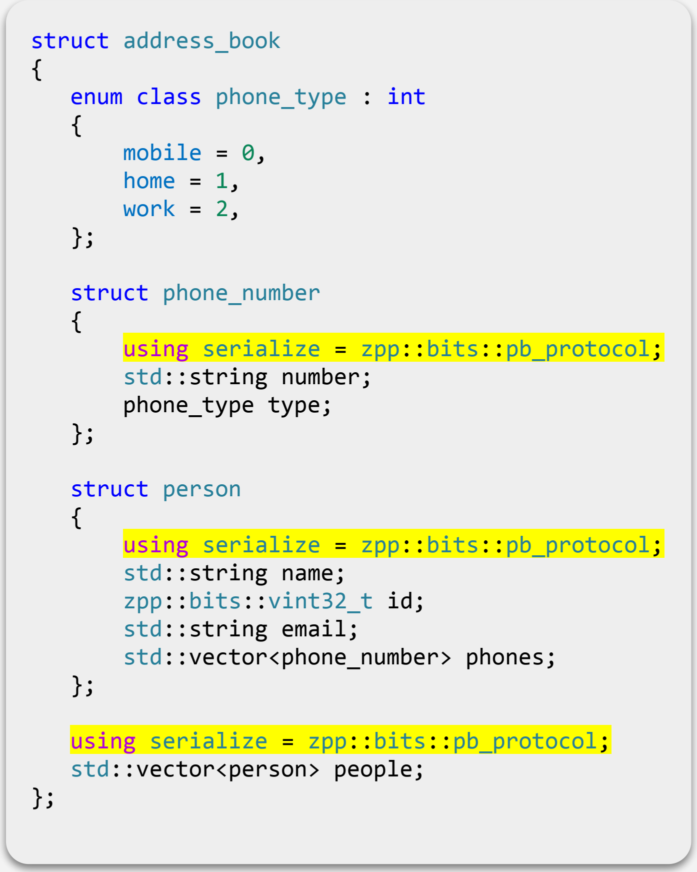
Response: x + y : 3

Note: Id collision/doesn't exist == does not compile

Cross Language Serialization – Support Custom Protocols



```cpp
struct address_book
{
   enum class phone_type : int
   {
       mobile = 0,
       home = 1,
       work = 2,
   };

   struct phone_number
   {
       std::string number;
       phone_type type;
   };

   struct person
   {
       std::string name;
       int id;
       std::string email;
       std::vector<phone_number> phones;
   };

   std::vector<person> people;
};
```



```cpp
struct address_book
{
   enum class phone_type : int
   {
       mobile = 0,
       home = 1,
       work = 2,
   };

   struct phone_number
   {
       using serialize = zpp::bits::pb_protocol;
       std::string number;
       phone_type type;
   };

   struct person
   {
       using serialize = zpp::bits::pb_protocol;
       std::string name;
       zpp::bits::vint32_t id;
       std::string email;
       std::vector<phone_number> phones;
   };

   using serialize = zpp::bits::pb_protocol;
   std::vector<person> people;
};
```

Link

Summary – Achieved vs Still not Perfect

- **What we achieved**
  - **Zero overhead** – As we saw in godbolt and comparing versus other libraries
  - **Freestanding** – no dependencies except C++ headers (no linkage with C++ runtime)
    - Compile flags in the appendix are provided for clang with libc++ headers.
  - **Simple to use** – almost always no additional lines added to the class, or just one line to provide the number of members.

- **What is still not perfect**
  - Use of non standard always_inline/__forceinline
  - Some complexity is still necessary due to lack of reflection/P1061

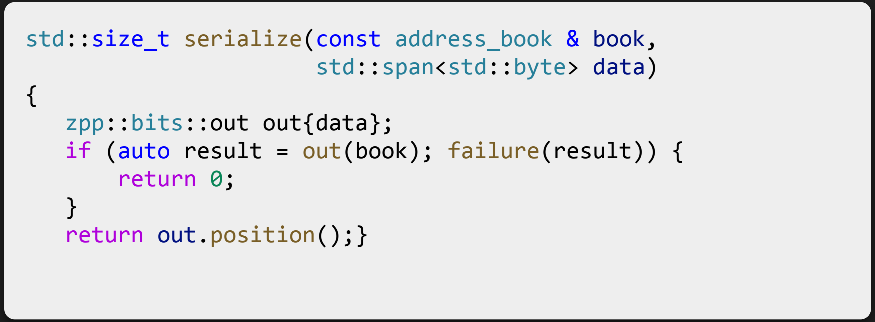
```
zpp::bits::out{data}("Thank You!"sv).or_throw();
```

- The *zpp::bits* library: https://github.com/eyalz800/zpp_bits
- My email: eyal.zedaka@gmail.com
- Feel free to submit questions as github issues / email.

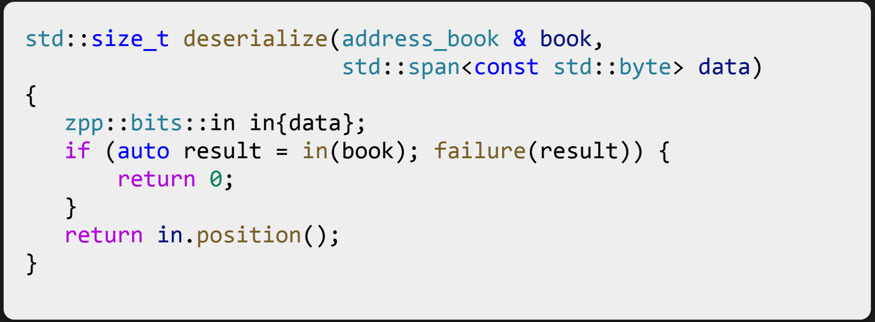
Even the zero overhead language ***can't always be zero overhead on its own,*** *though we probably wouldn't have C++ be called that way,* ***had we ever accepted it***

Appendix

Freestanding



```
std::size_t serialize(const address_book & book,
                      std::span<std::byte> data)
{
   zpp::bits::out out{data};
   if (auto result = out(book); failure(result)) {
       return 0;
   }
   return out.position();}
```

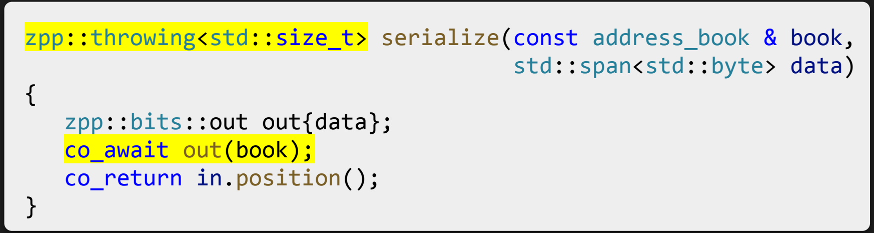


```
std::size_t deserialize(address_book & book,
                        std::span<const std::byte> data)
{
   zpp::bits::in in{data};
   if (auto result = in(book); failure(result)) {
       return 0;
   }
   return in.position();
}
```

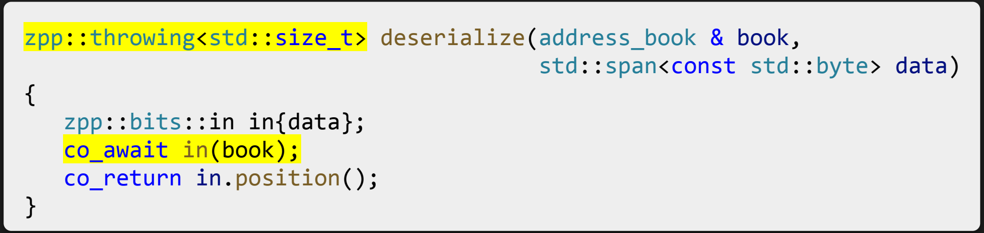
-std=c++20 -stdlib=libc++ -nostdlib -fno-exceptions -fno-rtti -fno-unwind-tables -fno-threadsafe-statics
-e _start -fuse-ld=lld -fno-stack-protector
-D_LIBCPP_DISABLE_VISIBILITY_ANNOTATIONS
-D_LIBCPP_HAS_NO_VENDOR_AVAILABILITY_ANNOTATIONS
-D_LIBCPP_DISABLE_EXTERN_TEMPLATE
-D_LIBCPP_HAS_NO_THREADS

Link

Bonus - Freestanding & Throwing



```cpp
zpp::throwing<std::size_t> serialize(const address_book & book,
                                     std::span<std::byte> data)
{
   zpp::bits::out out{data};
   co_await out(book);
   co_return in.position();
}
```



```cpp
zpp::throwing<std::size_t> deserialize(address_book & book,
                                       std::span<const std::byte> data)
{
   zpp::bits::in in{data};
   co_await in(book);
   co_return in.position();
}
```

-std=c++20 -stdlib=libc++ -nostdlib -fno-exceptions -fno-rtti -fno-unwind-tables -fno-threadsafe-statics
-e _start -fuse-ld=lld -fno-stack-protector
-D_LIBCPP_DISABLE_VISIBILITY_ANNOTATIONS
-D_LIBCPP_HAS_NO_VENDOR_AVAILABILITY_ANNOTATIONS
-D_LIBCPP_DISABLE_EXTERN_TEMPLATE
-D_LIBCPP_HAS_NO_THREADS

Link