
Core C++ 2026
20-21Oct. 2026 :: Tel-Aviv
ceph
Concepts IRL*
Using C++20 Concepts in a Large Codebase
Yuval Lifshitz, IBM
* In Real Life

What (I Was Thinking) Concepts Are Good For?

- Making error messages on templates more human readable
- Giving cool talks in C++ conferences

A Template with Unwritten Requirements

Exposing a C++ container to Lua scripts:
The library needs four static functions from the type

```
1  // lua_utils.h
2  //
3  // M is expected to be a class with the following members:
4  //   IndexClosure, NewIndexClosure, PairsClosure, LenClosure
5  //   (static functions that accept "lua_State*" and return "int")
6  template <typename M>
7  void create_metatable(lua_State* L) {
8    set_handler(L, "__index",    M::IndexClosure);    // <-- expecting M::IndexClosure static member
9    set_handler(L, "__newindex", M::NewIndexClosure); // <-- expecting M::NewIndexClosure static member
10   set_handler(L, "__pairs",    M::PairsClosure);    // <-- expecting M::PairsClosure static member
11   set_handler(L, "__len",      M::LenClosure);      // <-- expecting M::LenClosure static member
12 }
```

Using the Template for HTTP Request Headers

```cpp
// http_request_headers.cc
#include "lua_utils.h"

struct HTTPRequestHeadersMetaTable {
  static int IndexClosure(lua_State* L);
  static int NewIndexClosure(lua_State* L);
  static int LenClosure(lua_State* L);
  // oops: forgot PairsClosure
};

// trying to create a metatable of type HTTPRequestHeadersMetaTable
create_metatable<HTTPRequestHeadersMetaTable>(L);
```

The requirements on the struct live in a comment

The Error Message*

```
lua_utils.h:8:35: error: no member named 'PairsClosure' in 'HTTPRequestHeadersMetaTable'
    8 |   set_handler(L, "__pairs",    M::PairsClosure);
      |                                ~~~^
http_request_headers.cc:11:3: note: in instantiation of function template specialization
                'create_metatable<HTTPRequestHeadersMetaTable>' requested here
```

** Line numbers in compiler errors are those of the real code, not of the slides*

The Same Requirements, as a Concept

```cpp
template <typename T>
concept lua_metatable = requires(lua_State* L) {
  { T::IndexClosure(L) }    -> std::same_as<int>;
  { T::NewIndexClosure(L) } -> std::same_as<int>;
  { T::PairsClosure(L) }    -> std::same_as<int>;
  { T::LenClosure(L) }      -> std::same_as<int>;
};

template <lua_metatable M>
void create_metatable(lua_State* L); // <-- same implementation as before

// trying to create a metatable of type HTTPRequestHeadersMetaTable
// still failing, but with a better error message
create_metatable<HTTPRequestHeadersMetaTable>(L);
```

The comment is now code

The Error Message, with a Concept

```
http_request_headers.cc:11:3: error: no matching function for call to 'create_metatable'
   11 |   create_metatable<HTTPRequestHeadersMetaTable>(L);
      |   ^~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
lua_utils.h:13:11: note: because 'HTTPRequestHeadersMetaTable' does not satisfy 'lua_metatable'
lua_utils.h:9:8: note: because 'T::PairsClosure(L)' would be invalid:
                 no member named 'PairsClosure' in 'HTTPRequestHeadersMetaTable'
```

- The error moved from the library's implementation to the line of code where the problem occurred
- It names the requirement that was broken, not the statement that happened to trip

NICER. BUT IS IT WORTH TOUCHING WORKING CODE FOR?

Ceph Is a Large and Complex Codebase

- Open Source project started in 2004 (open sourced in 2006)
- ~2M lines of C++ code in ~4.5K C++ source/header files (excluding submodules)
- Has ~1,500 contributors from many organizations
- Deployed in ~3.7K clusters / ~1.7 exabytes
- E.g. CERN runs more than 100 petabytes of Ceph storage

NEED A BETTER REASON TO CHANGE CEPH'S CODE THAN "MORE READABLE COMPILATION ERRORS"

I Was Somewhat Wrong

SERIALIZATION LIBRARY

Every Ceph type that goes to disk or over the network is serialized with a free function `encode(value, buffer)`, and that includes any container of any such type.

Template resolution is a mess with `enable_if`

Encodable Types

Detecting if a type has an `encode(bufferlist&)` member:

```
// primary template: "no"
template <typename T, typename = void>
struct has_encode : std::false_type {};

// specialization: "yes", but only if v.encode(bl) compiles
template <typename T>
struct has_encode<T, std::void_t<decltype(
    std::declval<const T&>().encode(std::declval<bufferlist&>()))>>
  : std::true_type {};
```

Using it: `encode()` for the types that have it:

```
template <typename T>
std::enable_if_t<has_encode<T>::value>
encode(const T& v, bufferlist& bl) { v.encode(bl); }
```

Vectors

```
1 // vector of anything: one element at a time
2 template <typename T>
3 std::enable_if_t<!std::is_arithmetic_v<T>>  // <-- note the ! sign
4 encode(const std::vector<T>& v, bufferlist& bl) {
5   encode(uint32_t(v.size()), bl);
6   for (const auto& e : v) encode(e, bl);
7 }
```

Numeric Types and Vectors

```
1 // numbers: copy the bytes
2 template <typename T>
3 std::enable_if_t<std::is_arithmetic_v<T>>
4 encode(const T& v, bufferlist& bl);
```

```
1 // vector of numbers: one copy
2 template <typename T>
3 std::enable_if_t<std::is_arithmetic_v<T>>
4 encode(const std::vector<T>& v, bufferlist& bl);
```

Why the `!` Is There

Take the `!` away, and both overloads accept a vector of numbers:

```
1 template <typename T>                          // vector of anything
2 void encode(const std::vector<T>& v, bufferlist& bl);
3
4 template <typename T>
5 std::enable_if_t<std::is_arithmetic_v<T>>   // vector of numbers
6 encode(const std::vector<T>& v, bufferlist& bl);
7
8 encode(std::vector<Bucket>{}, bl);   // OK: only the first one matches
9 encode(std::vector<int>{}, bl);      // error: call to 'encode' is ambiguous
```

The compiler cannot tell that one overload is a special case of the other

Encodable Type Concept

```
1  template <typename T>
2  concept raw_bytes = std::is_arithmetic_v<T>;
3
4  template <typename T>
5  concept self_encoding = requires(const T& v, bufferlist& bl) { v.encode(bl); };
6
7  void encode(const raw_bytes auto& v, bufferlist& bl);
8  void encode(const self_encoding auto& v, bufferlist& bl) { v.encode(bl); }
9
10 template <typename T>
11 concept encodable = requires(const T& v, bufferlist& bl) { encode(v, bl); };
```

Range Concepts

```cpp
template <typename R>
concept encodable_range =
  std::ranges::sized_range<R> &&            // <-- it's a range that knows its size
  encodable<std::ranges::range_value_t<R>>; // <-- its elements are encodable
  // no need to mention it's *not* a raw_bytes_range

template <typename R>
concept raw_bytes_range =
  encodable_range<R> &&                     // <-- it's an encodable range
  std::ranges::contiguous_range<R> &&       // <-- it's a contiguous range
  raw_bytes<std::ranges::range_value_t<R>>; // <-- it holds numeric values
```

Encoding Ranges

```cpp
// any container of anything encodable: one element at a time
void encode(const encodable_range auto& r, bufferlist& bl) {
  encode(uint32_t(std::ranges::size(r)), bl);
  for (const auto& e : r) encode(e, bl);
}

// contiguous numbers: one copy
void encode(const raw_bytes_range auto& r, bufferlist& bl);
```

The Same Calls, with Concepts

```
1 void encode(const encodable_range auto& r, bufferlist& bl);   // any container
2 void encode(const raw_bytes_range auto& r, bufferlist& bl);   // contiguous numbers
3
4 encode(std::vector<Bucket>{}, bl);   // only the first one matches
5 encode(std::vector<int>{}, bl);      // both match: the second one is chosen
```

`raw_bytes_range` is defined as `encodable_range && ...`,
so the compiler knows it is the special case

I Was Really Wrong

RUNTIME PERFORMANCE IMPROVEMENT MADE FEASIBLE

- A concept is a question about a type that the compiler answers
- Ask the right question, and you can choose a faster implementation
- Without concepts it is a trade-off between speed and code complexity

Transparent Comparators

Can compare a key with something that is not a key

```
1  // case insensitive comparator that works on string views
2  struct no_case_less {
3    using is_transparent = void;   // <-- this tells std::map to enable this overload:
4                                   //     template <typename K> iterator find(const K& k);
5    bool operator()(std::string_view a, std::string_view b) const {
6      return std::ranges::lexicographical_compare(a, b, {}, ::tolower, ::tolower);
7    }
8  };
9
10 std::map<std::string, std::string>                plain;
11 std::map<std::string, std::string, no_case_less> headers;
```

Looking for a C String Coming from Lua

```
1 std::string_view key = lua_tostring(L, 2); // <-- Lua returns a C-string
2
3 plain.find(key);                           // <-- does not compile
4 plain.find(std::string{key});              // <-- compiles; allocates a temporary
5 headers.find(key);                         // <-- compiles; no temporary
```

When I try to write generic code that work on any map of strings
I just allocate the temporary `std::string`

Look Up a `string_view`

```
1  template <typename Map>
2  concept finds_by_string_view =
3    requires(Map& m, std::string_view key) { m.find(key); };
4
5  auto find_entry(auto& m, std::string_view key) {
6    return m.find(std::string{key});   // works for every map
7  }
8
9  auto find_entry(finds_by_string_view auto& m, std::string_view key) {
10   return m.find(key);                // used whenever it is possible
11 }
```

One call, `find_entry(map, key)`, for every map type in the code base

The Same Thing, Before C++20

```cpp
template <typename Map, typename = void>
struct finds_by_string_view : std::false_type {};

template <typename Map>
struct finds_by_string_view<Map, std::void_t<decltype(
    std::declval<Map&>().find(std::declval<std::string_view>()))>>
  : std::true_type {};

template <typename Map>
auto find_entry_impl(Map& m, std::string_view key, std::false_type) {
  return m.find(std::string{key});
}

template <typename Map>
auto find_entry_impl(Map& m, std::string_view key, std::true_type) {
  return m.find(key);
}

template <typename Map>
auto find_entry(Map& m, std::string_view key) {
  return find_entry_impl(m, key, finds_by_string_view<Map>{});
}
```

ceph

Convert an Infinite Loop to a Compile Time Error

- Some bugs are a property of the *type*, not of the values
- A concept can turn "works with any container (almost)" into a rule the compiler enforces

Stateless Iterators

A script iterates over a C++ container that we expose to it:

```lua
for key, value in pairs(RequestHeaders) do
  print(key, value)
end
```

Since `RequestHeaders` is a C++ container pretending to be a Lua table,
we supply our own `next()` function, through the table's `__pairs` metamethod.

C++ Binding for the `next()` Function

Lua drives the loop by calling `key, value = next(table, key)`, again and again:

```cpp
template <typename Map>
int next(lua_State* L) {
  auto& m = get_table<Map>(L);             // stored as opaque user data
  auto it = m.begin();
  // nil key marks the start of the loop
  // if key is nil, continue at line 13
  if (!lua_isnil(L, 2)) {
    const char* key = lua_tostring(L, 2);  // get the input "key" parameter
    it = m.find(key);                      // find the key we returned last time
    if (it != m.end()) ++it;               // ... and step to the one after it
  }
  // in Lua you push values to a stack to return them
  if (it == m.end()) {                     // end of the loop
    lua_pushnil(L);                        // return nil key
    lua_pushnil(L);                        // and nil value
  } else {
    lua_pushstring(L, it->first.c_str());  // return the next key
    lua_pushstring(L, it->second.c_str()); // return the next value
  }
  return 2;                                // returning 2 things: key, value
}
```

The Script Never Returns

Now we use the template with a container that allows duplicate keys:

```cpp
std::multimap<std::string, std::string> unlucky_headers{
  {"accept", "text/html"},
  {"accept", "application/json"},
  {"host",   "ceph.io"}};

expose(L, "RequestHeaders", unlucky_headers);   // pairs() will use next<std::multimap>
```

The Lua code:

```lua
for key, value in pairs(RequestHeaders) do
  print(key, value)
end
```

What it prints:

```
accept  text/html
accept  application/json
accept  application/json
accept  application/json
... (forever)
```

Make It a Compile Time Error

"Stateless iteration needs unique keys"
The standard containers already tell us which is which:

```
1 std::map<K, V>::insert(value)       // returns std::pair<iterator, bool>
2 std::multimap<K, V>::insert(value)  // returns iterator: no boolean, no failure on duplicate keys
```

The `unique_keys` Concept

```cpp
template <typename C>
concept unique_keys = requires(C& c, typename C::value_type v) {
  { c.insert(v) } -> std::same_as<std::pair<typename C::iterator, bool>>;
};

template <unique_keys Map>
int next(lua_State* L);

template <unique_keys Map>
void expose(lua_State* L, const char* name, Map& m);
```

Compile Time Error vs. Infinite Loop

```
1  expose(L, "RequestHeaders", unlucky_headers);
```

```
error: no matching function for call to 'expose'
   97 |   expose(L, "RequestHeaders", unlucky_headers);
      |   ^~~~~~
note: candidate template ignored: constraints not satisfied
      [with Map = std::multimap<std::string, std::string>]
note: because 'std::multimap<std::string, std::string>' does not satisfy 'unique_keys'
```

What (I Now Think) Concepts Are Good For

NON-INTRUSIVE SOLUTION *(*)* FOR A COMPLEX TYPE SYSTEM *(**)*

The class hierarchy says what a type *is*.
A concept says what we can *do* with it

The types did not change.
No base class was added

(*) no type was hurt in the making of these slides

(**) serializeable types and types with Lua bindings

More on C++ Concepts

Back to Basics: Concepts in C++ (Nicolai Josuttis - CppCon 2024)
https://youtu.be/jzwqTi7n-rg

From C++ Templates to C++ Concepts (Alex Dathskovsky - CppCon 2022)
https://youtu.be/_doRiQS4GS8

Concept-based Generic Programming (Bjarne Stroustrup - Paper/CppCon 2025)
https://www.stroustrup.com/Concept-based%20GP.pdf
https://youtu.be/VMGB75hsDQo

Concepts TS working draft (N4553, 2015):
https://www.open-std.org/jtc1/sc22/wg21/docs/papers/2015/n4553.pdf

Elements of Programming (Stepanov, McJones) - generic programming and bits on the machine:
https://elementsofprogramming.com/

Thank You

Q&A



Slides&Code

THE CHANGES BEHIND THIS TALK:

Serialization: https://github.com/ceph/ceph/pull/70678
Lua bindings: https://github.com/ceph/ceph/pull/70987

THANKS TO JESSE F. WILLIAMSON (IBM) FOR ASSISTANCE!