Leveraging C++20/23 Features for Low Level Interactions

Overview

In a baremetal environment, we're going to demonstrate effective use of C++
How did we end up with C language HW interactions?
What are best practices for using C from C++?
How can we use C++ to make HW access cleaner, safer, and more testable?

Why is C so prolific?

It's the kernel, silly!
30 years ago, all low level interaction was done with C
Organizations are very comfortable with C in embedded applications
C++ Still viewed skeptically by many when considering resource constrained applications

What are the advantages of C++?

Obvious question: Why do this in C++?
C++ gives you:

- Lifetime Management
- Type Safety
- Sophisticated Strong Typing System
- Inheritance and Polymorphism

But you may still need:

- Kernel functions
- Existing APIs
- Conversion to raw pointers

So, use C++, but don't negate the advantages by creating dangerous C-like code.

Why not rewrite all the C code?

There's LOTS of code out there![1]

1. See also Stroustrup (2023), in references

Why not rewrite all the C code?

There's LOTS of code out there![1]
To rewrite a modest sized C code base would take years!
Rewriting the linux kernel is likely intractable.

1. See also Stroustrup (2023), in references

Why not rewrite all the C code?

There's LOTS of code out there![1]
To rewrite a modest sized C code base would take years!
Rewriting the linux kernel is likely intractable.
Ergo: Need to keep using C

1. See also Stroustrup (2023), in references

Why not rewrite all the C code?

There's LOTS of code out there![1]
To rewrite a modest sized C code base would take years!
Rewriting the linux kernel is likely intractable.
Ergo: Need to keep using C
But: How do we use it effectively in line with C++ best practices?

1. See also Stroustrup (2023), in references

Key C++ Safety Features

There's lots, but....

Lifetime Management

Smart pointers handle new/delete for you. We'll mostly be discussing `unique_ptr<T>`

Strict Typing

C++ has plenty of language elements that 'stop' you at the compilation step.

Standard Tests

std provides many useful tests well beyond libc

Naive crossing of the C/C++ boundary

In C++: Smart Pointers are GREAT!
Every smart pointer has an underlying reference.
The 'get' method accesses this naked pointer

```
unique_ptr<T> my_t_ptr = make_unique<T>( );
my_c_api( my_t_ptr -> get( ) );
```

Naive crossing of the C/C++ boundary

In C++: Smart Pointers are GREAT!
Every smart pointer has an underlying reference.
The 'get' method accesses this naked pointer

```
unique_ptr<T> my_t_ptr = make_unique<T>( );
my_c_api( my_t_ptr -> get( ) );
```

Then we can use the result, right?

```
next_function( my_t_ptr -> get( ) );
```

Naive crossing of the C/C++ boundary

In C++: Smart Pointers are GREAT!
Every smart pointer has an underlying reference.
The 'get' method accesses this naked pointer

```
unique_ptr<T> my_t_ptr = make_unique<T>( );
my_c_api( my_t_ptr -> get( ) );
```

Then we can use the result, right?

```
next_function( my_t_ptr -> get( ) );
```

Answer: **Maybe.**

Smart pointers across the C/C++ boundary

Use case:

- Allocate something in C++ that needs to go through a C API
- Pass that into C, get result

Risks:

- Synchronization Problem: Underlying C code may have changed the pointer
- Memory Safety Problem: Underlying C code may have deleted and reallocated

Smart pointers across the C/C++ boundary

Use case:

- Allocate something in C++ that needs to go through a C API
- Pass that into C, get result

Risks:

- Synchronization Problem: Underlying C code may have changed the pointer
- Memory Safety Problem: Underlying C code may have deleted and reallocated

Standard library has syntactic sugar for you: `std::inout_ptr` and `std::out_ptr`

Example: make_unique into C API and back

```
1 std::unique_ptr<T> my_t_ptr = std::make_unique<T>();
2 my_c_api( std::inout_ptr(my_t_ptr) );
3 next_function( std::inout_ptr(my_t_ptr) );
```

Example: make_unique into C API and back

```
1 std::unique_ptr<T> my_t_ptr = std::make_unique<T>();
2 my_c_api( std::inout_ptr(my_t_ptr) );
3 next_function( std::inout_ptr(my_t_ptr) );
```

Example: make_unique into C API and back

```
1 std::unique_ptr<T> my_t_ptr = std::make_unique<T>();
2 my_c_api( std::inout_ptr(my_t_ptr) );
3 next_function( std::inout_ptr(my_t_ptr) );
```

What's really going on with out_ptr and inout_ptr?

Underneath any smart pointer is a raw pointer or reference
Both out_ptr and inout_ptr pass that raw pointer to the C function when it's called.
When the function returns, inout_ptr is reset to the returned value.
Roughly equivalent to:

```
1 std::unique_ptr<T> my_t_ptr = std::make_unique<T>( );
2 T* my_raw_t_ptr = my_t_ptr -> get( );
3 my_c_api(my_raw_t_ptr);
4 my_t_ptr -> reset(my_raw_t_ptr);
5 next_function( my_t_ptr -> get( ) );
```

What's really going on with out_ptr and inout_ptr?

Underneath any smart pointer is a raw pointer or reference
Both out_ptr and inout_ptr pass that raw pointer to the C function when it's called.
When the function returns, inout_ptr is reset to the returned value.
Roughly equivalent to:

```
std::unique_ptr<T> my_t_ptr = std::make_unique<T>();
T* my_raw_t_ptr = my_t_ptr -> get();
my_c_api(my_raw_t_ptr);
my_t_ptr -> reset(my_raw_t_ptr);
next_function( my_t_ptr -> get() );
```

What's really going on with out_ptr and inout_ptr?

Underneath any smart pointer is a raw pointer or reference
Both out_ptr and inout_ptr pass that raw pointer to the C function when it's called.
When the function returns, inout_ptr is reset to the returned value.
Roughly equivalent to:

```
std::unique_ptr<T> my_t_ptr = std::make_unique<T>();
T* my_raw_t_ptr = my_t_ptr -> get();
my_c_api(my_raw_t_ptr);
my_t_ptr -> reset(my_raw_t_ptr);
next_function( my_t_ptr -> get() );
```

What's really going on with out_ptr and inout_ptr?

Underneath any smart pointer is a raw pointer or reference
Both out_ptr and inout_ptr pass that raw pointer to the C function when it's called.
When the function returns, inout_ptr is reset to the returned value.
Roughly equivalent to:

```
std::unique_ptr<T> my_t_ptr = std::make_unique<T>();
T* my_raw_t_ptr = my_t_ptr -> get();
my_c_api(my_raw_t_ptr);
my_t_ptr -> reset(my_raw_t_ptr);
next_function( my_t_ptr -> get() );
```

What's really going on with out_ptr and inout_ptr?

Underneath any smart pointer is a raw pointer or reference
Both out_ptr and inout_ptr pass that raw pointer to the C function when it's called.
When the function returns, inout_ptr is reset to the returned value.
Roughly equivalent to:

```
1 std::unique_ptr<T> my_t_ptr = std::make_unique<T>();
2 T* my_raw_t_ptr = my_t_ptr -> get();
3 my_c_api(my_raw_t_ptr);
4 my_t_ptr -> reset(my_raw_t_ptr);
5 next_function( my_t_ptr -> get() );
```

The smart pointer get function is a *value*

When the underlying point is get-ed, the caller receives a *copy* of the pointer address.

```
pointer get() const noexcept;[^2]
```

So, if C moves the pointer or deletes the pointer and re-allocates, the C++ pointer will be stale

Details matter

We can't let the compiler do funny things
... like memory layout reordering

```
class Foo {
private:
    int32_t a;

public:
    int32_t b;
    int32_t c;
}
```

This is one example where the compiler is allowed to[1] reorder memory – in this case due to multiple access specifiers.

1. But probably doesn't

Other C++ guidance at the C boundary

There are a handful of rules [1] * Use same access control * No `virtual` in the object or any base class * No non-static data members in the most derived class * All base class(es) are standard layout * All members of an object need to follow these rules as well

1. From Fertig (2020), see references

How to test for memory compatibility

We need the object to be both trivial and standard layout

Trivial via `std::is_trivial`

Ensures the type supports static initialization

Standard Layout via `std::is_standard_layout`

Ensures the type memory layout is the same in C and C++

Break free of the kernel

We use C because kernels use C
Baremetal embedded is really doing what the kernel does, so just be like the kernel right?

Break free of the kernel

We use C because kernels use C
Baremetal embedded is really doing what the kernel does, so just be like the kernel right?
NO!
C++ has advantages, so take advantage of it

Naive HW access

First concept: Hide all the HW access within functions

```
class my_uart {
...
  void set_baud_rate(const uint32_t& rate){
    *BAUD_RATE_REG = rate;
  }
  uint32_t get_baud_rate( ) const {
    return *BAUD_RATE_REG;
  }
};
```

pimpl Idiom

pimpl = pointer to implementation

```
class my_class {
...
private:
  unique_ptr<my_class_impl> p_impl;
}
```

Why use this design pattern?
Changes to the implementation don't impact the class definition. Hence, no recompile of the callers/users of the class.

pimpl inspired register access

pimpl = pointer to implementation
Why not pointer to register set?

```
struct my_uart_regs {
  ...
  volatile uint32_t BAUD_RATE_REG;
  ...
};

class my_uart {
...
private:
  unique_ptr<my_uart_regs> p_regs;
}
```

Why use a pointer to a register set?

Make more testable code!

```
class my_uart {
public my_uart(unique_ptr<my_uart_regs> regs);
...
private:
  unique_ptr<my_uart_regs> p_regs;
}
```

Any object that can be casted to my_uart_regs or is inherited from my_uart_regs can be handed to the class.

Easy substitution of test harnesses.[1]

1. See also my talk from cppcon2023 re: HookableRegister

How to define the struct of the register set

We need a struct of the registers in order, including padding of empty spaces.
Don't change the data width – we need the structure packed, aligned, and volatile

```
typedef struct __attribute__((packed)) __attribute__((aligned(4
  volatile uint32_t reg1; // Offset 0
  volatile uint32_t reg2; // Offset 4
  volatile uint32_t pad[4];
  volatile uint32_t reg3; // Offset 20
  ...
} regs_t;
```

Managing the smart pointer

The interface for a smart pointer allows initialization

```
uint32_t my_regs_addr = 0x10D030000;
regs_t* my_regs_raw_ptr = reinterpret_cast<regs_t*>(my_regs_add
unique_ptr<regs_t> p_regs.reset(my_regs_raw_ptr);
```

Managing the smart pointer

The interface for a smart pointer allows initialization

```
unique_ptr<regs_t> p_regs.reset(reinterpret_cast<regs_t*>(0x10D
```

Managing the smart pointer

The interface for a smart pointer allows initialization

```
unique_ptr<regs_t> p_regs.reset(reinterpret_cast<regs_t*>(0x10D
```

But you're likely to crash when this leaves scope

Managing the smart pointer

The interface for a smart pointer allows initialization

```
unique_ptr<regs_t> p_regs.reset(reinterpret_cast<regs_t*>(0x10D
```

But you're likely to crash when this leaves scope

Why? **The deleter.**

A note about unique_ptr

unique_ptr may intuitively be thought to have the declaration

```
template<typename T>
class unique_ptr {...};
```

But it's actually more like

```
template<typename T, typename deleter = std::default_delete<T>>
class unique_ptr {...};
```

The custom deleter for shared_ptr and unique_ptr are different

We're used to this construct of a smart pointer:

```
shared_ptr<T> my_ptr = rhs;
```

But this will cause a 'delete' when the scope changes So we need to tell the language to not delete

```
unique_ptr<T,D> my_uniqueptr = rhs;
shared_ptr<T> my_sharedptr(ref,D);
```

Where D is a (dangerous!) custom deleter defined by:

```
template<T>
struct no_deleter {
  operator ( ) (T* ptr) {
    // do nothing
  }
}
```

The custom deleter for shared_ptr and unique_ptr are different

We're used to this construct of a smart pointer:

```
shared_ptr<T> my_ptr = rhs;
```

But this will cause a 'delete' when the scope changes So we need to tell the language to not delete

```
unique_ptr<T,D> my_uniqueptr = rhs;
shared_ptr<T> my_sharedptr(ref,D);
```

Where D is a (dangerous!) custom deleter defined by:

```
~~template<T>~~
struct no_deleter {
  operator ( ) (~~T~~* ptr) {
    // do nothing
  }
}
```

Don't template the custom no_deleter!

So, making meaningful functions...

```
class my_uart {
...
  void set_baud_rate(const uint32_t& rate) {
    p_regs -> BAUD_RATE_REG = rate;
  }
  uint32_t get_baud_rate( ) const {
    return p_regs -> BAUD_RATE_REG;
  }

  private:
    unique_ptr<my_uart_regs, my_uart_no_deleter>-> p_regs;
};
```

How to do pointer-to-register HW access in ‘pure’ C++

1. Define a struct that represents the HW, packed and aligned

```
struct my_uart_regs __attribute__((packed)) __attribute__((alig
  ...
  volatile uint32_t BAUD_RATE_REG;
  ...
};
```

2. Smart pointer initialized to HW address

```
class my_uart {
...
  my_uart(uintptr_t base_addr) : p_regs(base_addr){
    ...
  }
...
  private:
    unique_ptr<my_uart_regs, uart_regs_no_deleter>-> p_regs;
...
};
```

3. Custom deleter – don’t try to deallocate the HW

```
struct uart_regs_no_deleter {
  operator () (my_uart_regs* ptr) {
    // do nothing
  }
}
```

How to operate on the registers

C++ supports all the logic operators to manipulate bits that you have in C

```
uint32_t value = 0;
value |= 0x1;
value &= ~(0x1);
```

How to operate on the registers

C++ supports all the logic operators to manipulate bits that you have in C

```
uint32_t value = 0;
value |= 0x1;
value &= ~(0x1);
```

This will all work! But...

C++ can do these operations more safely through strong typing

How to operate on the registers

C++ supports all the logic operators to manipulate bits that you have in C

```
uint32_t value = 0;
value |= 0x1;
value &= ~(0x1);
```

This will all work! But...

C++ can do these operations more safely through strong typing

And **constexpr** can shift work into the compiler

Break from your `#define`d habits

Preprocessor directive #define is incredibly dangerous
...But it's also not unusual to come across macros and constants via #define

```
#define REPLACE_BITS(x, mask, bits) ((x & (~(mask))) | (bits &
```

Break from your #defined habits

Preprocessor directive #define is incredibly dangerous

...But it's also not unusual to come across macros and constants via #define

```
#define REPLACE_BITS(x, mask, bits) ((x & (~(mask))) | (bits &
```

Hello std::bitset!

```
template<std::size_t N>
std::bitset<N> replace_bits(std::bitset<N> x, std::bitset<N> ma
  return ((x & (~(mask))) | (bits & mask));
}
```

Hence, strongly typed macro-like constexpr capable bit manipulation functions!

Bit Manipulation via constexpr positives

Use your literals

```
std::bitset<32> my_bits = 0x3433;
```

std::bit_cast is great for floats:

```
std::bit_cast<std::uint32_t>(1.0f);
```

Avoid maximal munch problems around ‘E’

```
constexpr std::bitset<32> bad_bits = 0xE+1; // compiler doesn't
constexpr std::bitset<32> fourteen = 0xE;
constexpr std::bitset<32> good_bits = fourteen + 1; // compiler
```

What about endianness?

There are not infrequent reasons to swap bytes:

- Network endiannes to Processor endianness
- Cryptographic endianness to Processor endianness
- HW Bugs (yes, they do happen!)

C++23 gives us std::byteswap (which can be a constexpr)

What about endianness?

There are not infrequent reasons to swap bytes:

- Network endiannes to Processor endianness
- Cryptographic endianness to Processor endianness
- HW Bugs (yes, they do happen!)

C++23 gives us std::byteswap (which is a constexpr)

```
auto swapped = std::byteswap(my_integer);
```

In a loop, std::byteswap supports byte swizzle in an array

```
std::uint16_t array = {...};
for(auto a : array ){
  lhs = std::byteswap(a);
}
```

Upcoming in C++26

Key things I'm looking forward to experimenting with...

Saturated Arithmetic

```
add_sat(uint32_t x, uint32_t y);
```

Let std do your rollover checks for you!

Static Reflection

All sorts of fun ideas here

- enum to string
- members by index
- Testability and compile time improvements

Wrapping up

Developers do a lot in C to make it ‘safer’
But C++ has advantages:

- Strong typing and a more thorough type system makes code safer
- Lifetime management is important and C++ takes many of those questions off the table
- Still need static analysis, but the compiler does more for you

Altera is doing this, I encourage you to do this too!

QA

–

References and Credits

Stroustrup, Bjarne (2023) “Delivering Safe C++”, cppcon2023 https://www.youtube.com/watch?v=I8UvQKvOSSw
Fertig, Andreas (2020) “C++20: Aggregate, POD, trivial type, standard layout class, what is what.” https://andreasfertig.blog/2021/01/cpp20-aggregate-pod-trivial-type-standard-layout-class-what-is-what/
Fertig, Andreas (2024) “Understanding the inner workings of C++ smart pointers - The unique_ptr with custom deleter.” https://andreasfertig.blog/2024/08/understanding-the-inner-workings-of-cpp-smart-pointers-the-unique_ptr-with-custom-deleter/