Crafting Embedded Systems in C++

Dan & Ben Saks
Saks & Associates
www.dansaks.com

Overview

- Programming embedded systems is a very large topic.
- This presentation is intended to illustrate some of the challenges that embedded developers face through a few key examples.
- It will show how you can use the features of C++ to overcome those challenges.
- The goal is to help you think about how to craft objects that accurately model hardware.

- ***craft*** (verb):
  - "to make or manufacture (an object, objects, product, etc.) with skill and careful attention to detail."

Overview

- In many ways, embedded systems programming is just plain programming.
- However, embedded systems programming is a bit different.
  - Embedded systems often have stricter resource limitations:
    - run time
    - memory space
    - communication bandwidth
    - power consumption
  - Embedded systems often control hardware directly.

Your Choice

- You can write very simple declarations to model devices.
  - This is common practice.
  - It's less work up front, but then…
  - Code that accesses devices will be tedious and error-prone.
- You can write more detailed and accurate declarations to model devices.
  - This is what I'm about to show you to how to do.
  - It's more work up front, but then…
  - Code that accesses devices will be easier to write and more robust.

Your Choice

- You define the representation of each device at most once.
- However, you probably access each device many times.

- As Scott Meyers advises:

✓ ***Make interfaces:***

  ✓ ***easy to use correctly***

  ✓ ***hard to use incorrectly.***

Sample Hardware

- This talk uses examples based on ARM E7T (Evaluator-7T) single board computer.
- Programs running on the E7T communicate with devices via memory-mapped locations known as the ***device registers***:
  - a 64KB (16K word) region in the memory address space,
  - beginning at address 0x03FF0000.
- The device registers are grouped.
- Each group communicates with a single device or small set of devices.

Device Registers

- For example, the UART 0 group consists of six device registers:

Offset	Register	Description
0xD000	ULCON	line control register
0xD004	UCON	control register
0xD008	USTAT	status register
0xD00C	UTXBUF	transmit buffer register
0xD010	URXBUF	receive buffer register
0xD014	UBRDIV	baud rate divisor register

- The UART 1 group consists of another set of these same registers, but beginning at offset 0xE000.
- By the way, UART stands for "***U***niversal ***A***synchronous ***R***eceiver/***T***ransmitter".

Choosing the Right Integer Type

- You can declare a device register using the appropriately sized and signed integer type.
- For example:
  - A one-byte data register might be a plain `char`.
  - A two-byte status register might be an unsigned `short`.
- Each device register in these examples occupies a four-byte word.
  - Declaring each device register as `uint32_t` seems to work well.
  - Using a meaningful typedef is even better:

```
typedef uint32_t device_register;
```

Placing Memory-Mapped Objects

- Normally, you don't choose the memory locations where program objects reside.
- The compiler does, often with substantial help from the linker.
- For an object declaration at global scope, the compiler sets aside memory within a particular code segment.
- For an object declaration at local scope, the compiler sets aside memory within the stack frame of the function containing the declaration.

Placing Memory-Mapped Objects

- For an object representing memory-mapped device registers, the compiler doesn't get to choose where the object resides.
- The hardware has already chosen.
- We need to craft declarations for objects that let us access the hardware that fit with the memory-mapped locations chosen by hardware designer.

Locating Device Registers

- You can access a device register through a pointer whose value is the specified address.
- You can use ***pointer-placement*** to cast an integer value representing the address into a pointer value.
- You can encapsulate the cast-expression in a macro, as in:

```
#define UTXBUF0 ((device_register *)0x03FFD00C)
```

- Alternatively, you can use the cast-expression to initialize a pointer, as in:

```
device_register *UTXBUF0
    = (device_register *)(0x03FFD00C);
```

Placing Memory-Mapped Objects

- Once you've got the pointer initialized, you can manipulate the device register via the pointer.
- For example:

```
*UTXBUF0 = c;   // OK: send the value of c out the port
```

writes the value of character `c` to the UART 0's transmit buffer, sending the character value out the port.

Placing Memory-Mapped Objects

- Device registers typically have fixed locations.
- UTXBUF0's pointer value shouldn't change during run time.
- The compiler should reject any attempt to modify the value of UTXBUF0, as in:

```
UTXBUF0 = UTXBUF1;  // should not compile
++UTXBUF0;          // should not compile
```

Locating Device Registers

- If you declare `UTXBUF0` as a pointer object, then you can modify the pointer's value.
- You should declare `UTXBUF0` as a `const` pointer:

```
device_register *const UTXBUF0
    = ((device_register *)0x03FFD00C);
```

- Using a `const` pointer allows things that should work to still work, but rejects things that shouldn't work:

```
*UTXBUF0 = c;          // still OK
UTXBUF0 = UTXBUF1;     // now a compile error - good
++UTXBUF0;             // now a compile error - good
```

Reference-Placement

- In C++, you can use ***reference-placement*** as an alternative to pointer-placement:

```
device_register &UTXBUF0
    = *(device_register *)0x03FFD00C;
```

- A reference refers to an object.
- Here, you must initialize `UTXBUF0` to refer to a `device_register`.
- However, the cast above yields a "pointer to `device_register`", not a `device_register`.
- You must dereference the result of the cast to obtain an object to which the reference can bind.

Reference-Placement

- Using reference-placement, you can treat `UTXBUF0` as the register itself, not a pointer to the register, as in:

```
UTXBUF0 = c;     // OK: send the value of c out the port
```

- A reference acts like a const pointer in that:
  - You must bind the reference to an object at initialization.
  - After that, you can't rebind it to another object.

New-Style Casts

- C++ provides an alternative notation that distinguishes portable casts from potentially non-portable ones:

```
                                  // behavior:
static_cast<T>(e)                 //     explicitly specified
reinterpret_cast<T>(e)            //     implementation-defined
const_cast<T>(e)                  //     const only
```

- These “new-style” casts don’t provide any additional functionality beyond the “old-style” casts.
- They just offer a chance to write code that reveals its intent more clearly.

New-Style Casts

- In general, you should avoid using casts.

✓ *If you must use a cast in C++, use a new-style cast.*

- These new-style casts are easier to spot in source code:
  - for humans and
  - for search tools such as grep.
- This is good, because casts are hazardous.
- A hazard that's easier to spot is usually less of a hazard.

New-Style Casts

- Using a new-style cast, the definition for `UART0` now looks like:

```
UART *const UART0 =
    reinterpret_cast<UART *>(0x03FFD000);
```

- As a reference, it looks like:

```
UART &UART0 =
    *reinterpret_cast<UART *>(0x03FFD000);
```

Traditional Register Representation

- C programs often define symbols for device register addresses as clusters of related macros:

```
// timer registers
#define TMOD   ((unsigned volatile *)0x3FF6000)
#define TDATA  ((unsigned volatile *)0x3FF6004)
~~~

// UART registers
#define ULCON0 ((unsigned volatile *)0x3FFD000)
#define UCON0  ((unsigned volatile *)0x3FFD004)
~~~
```

- This approach has a couple of weaknesses…

Traditional Register Representation

- It leads to awkward and inconvenient interfaces.
  - Many device operations involve more than one device register.
  - You either have to pass more than one register address, as in:

```
UART_put(USTAT0, UTXBUF0, c);
```

  - Or worse, you must treat all device registers as global objects.
- It leads to error-prone interfaces.
  - All device register addresses have the same pointer type.
  - Type checking can't catch accidents such as:

```
UART_put(USTAT0, TDATA, c);  // put c to a timer?
```

Using Structures and Classes

- ✓ ***Use structures in C and classes in C++ to implement abstract types.***
- A structure or class provides a more accurate declaration for the collection of registers for a given device:

```
struct UART {
    device_register ULCON;
    device_register UCON;
    device_register USTAT;
    ~~~
};

void UART_put(UART *u, int c);
```

Using Structures and Classes

- Each structure or class is a distinct type.
- You can't convert a pointer to one into a pointer to another without using a cast.
- Type checking can now catch accidents such as:

```
UART *const com0 = reinterpret_cast<UART *>(0x3FFD000);
timer *const timer0 =
    reinterpret_cast<timer *>(0x3FF6000);

UART_put(timer0, c);    // put c to a timer?  no
```

- Let's apply this approach to the UART in greater detail…

Modeling Devices

- As mentioned earlier, many UART operations involve accessing more than one UART register.
- For example:
  - The TBE bit (Transmit Buffer Empty) in the USTAT register indicates whether the UTXBUF register is ready for use.
  - You shouldn't store a character into UTXBUF until the TBE bit is set to 1.
  - Storing a character into UTXBUF initiates output to the port and clears the TBE bit.
  - The TBE bit goes back to 1 when the output operation completes.

Modeling Devices

- Here's code that waits for UART 0's transmit buffer to empty before sending it a character:

```
while ((USTAT0 & TBE) == 0)
    ;
UTXBUF0 = c;
```

- The sample hardware has two UARTs — any UART operation that works on one UART works on the other.
- Again, you shouldn't pass UART registers (such as `USTAT` and `UTXBUF`) as separate function arguments.
- Rather, collect all the UART registers into a single structure and pass a pointer or reference to that structure…

Modeling Devices

- Here's a structure representing the UART registers:

```
struct UART {
    device_register ULCON;
    device_register UCON;
    device_register USTAT;
    device_register UTXBUF;
    device_register URXBUF;
    device_register UBRDIV;
};

enum { TBE = 0x40 };  // mask for TBE bit in USTAT
```

Modeling Devices

- Now you can define the two UARTs as:

```
UART *const UART0 = reinterpret_cast<UART *>(0x03FFD000);
UART *const UART1 = reinterpret_cast<UART *>(0x03FFE000);
```

- You can pass either `UART0` or `UART1` to a function that sends characters from a null-terminated character sequence to a UART:

```
UART_put(UART0, "hello, world\n");
UART_put(UART1, "goodnight, moon\n");
```

Modeling Devices

- Here's that function:

```
void put(UART *u, char const *s) {
    for (; *s != '\0'; ++s) {
        while ((u->USTAT & TBE) == 0)
            ;
        u->UTXBUF = *s;
    }
}
```

Modeling Devices

- Here's it is again using a reference parameter in C++:

```
void put(UART &u, char const *s) {
    for (; *s != '\0'; ++s) {
        while ((u.USTAT & TBE) == 0)
            ;
        u.UTXBUF = *s;
    }
}
```

- These functions might work.
- Then again, they might not…

Overly Aggressive Optimization

- It might not work properly if the compiler's optimizer is too aggressive.
- Ordinarily, an object's value doesn't change unless the program changes it.
  - The compiler uses this knowledge when optimizing code.
- Hardware registers aren't ordinary objects.
  - The value of `u.USTAT` can change due to external hardware events.
  - The compiler doesn't know this.
  - How can we tell it?

The Volatile Qualifier

- Declaring an object `volatile` informs the compiler that the object may change state even though the program didn't change it.
- Specifically, declaring an object `volatile`:
  - tells the compiler to treat each access (read or write) to that object literally
  - prevents the compiler from "optimizing away" accesses to that object, even when it seems safe to do so

The Right Dose of Volatility

- This declares UART0 as a "const pointer to a volatile UART":

```
UART volatile *const UART0 =
    reinterpret_cast<UART *>(0x03FFD000);
```

- This declares UART1 as a "reference to a volatile UART".

```
UART volatile &UART1 =
    *reinterpret_cast<UART *>(0x03FFE000);
```

- In these declarations, volatile isn't part of the UART type.
- This is appropriate only if some UARTs aren't volatile.
- If volatility is inherent in every UART, volatile should be part of the UART type…

The Right Dose of Volatility

- You could try declaring the entire UART type volatile, as in:

```
volatile struct UART {
    ~~~
};  // error: missing declarator
```

- It won't compile because the compiler wants to apply the volatile keyword to a UART object, as in:

```
volatile struct UART {  // type UART isn't volatile...
    ~~~
} u;                    // but object u is
```

The Right Dose of Volatility

- The compiler would also be happy applying `volatile` to a typedef name, as in either:

```
typedef volatile struct { // no struct tag
    ~~~
} UART;

typedef struct {          // no struct tag
    ~~~
} volatile UART;
```

- Either way, `UART` is now a volatile type, which could be what we want.
- But maybe not…

The Right Dose of Volatility

- Typically, all device registers (not just those in UARTs) are volatile.
- In that case, you should define `device_register` as a volatile type:

```
typedef uint32_t volatile device_register;
```

The Right Dose of Volatility

- If `device_register` is a volatile type, then you can revert to the original definition for `UART`:

```
struct UART {
    device_register ULCON;
    device_register UCON;
    device_register USTAT;
    ~~~
};
```

- This `UART` isn't a volatile type; however…

The Right Dose of Volatility

- Although `UART` isn't volatile, every non-static `UART` data member is volatile, which is really what we need.
- This actually simplifies using the `UART` type.
  - For example, you'll never need to declare a pointer or reference to a `UART` as a pointer or reference to a `volatile UART`.
- This form works fine in C:

```
UART *const UART0 = (UART *)0x03FFD000;
```

- This form is also just fine in C++:

```
UART &UART0 = *reinterpret_cast<UART *>(0x03FFD000);
```

Controlling a UART

- You can use a C++ class to package the UARTs as a abstract type.
- Our basic UART supports the following operations:
  - enable the UART
  - disable the UART
  - receive data from the UART
  - ask if the UART is ready to send data
  - send data to the UART

Receiving Data

- You receive (read) data from a UART by reading from its URXBUF register.
- You can read any time you want, but you might not get anything useful.
- The RDR (Receive Data Ready) bit indicates whether the URXBUF register contains valid data.
- Reading a character from URXBUF when RDR is 0 doesn't interfere with the current receive operation.
- Reading a character from URXBUF when RDR is 1 resets RDR to 0 and readies URXBUF to receive another transmission.

Receiving Data

- The `UART` class provides one input function:
  - `get()` returns an `int` whose value is either a character read from the UART, or -1 is no character is available.
- For example, if `port` is a reference to a `UART`, then:

```
int c;
while ((c = port.get()) < 0)
    ;
```

  waits for a valid character from the UART.
- Only the low-order byte of the URXBUF contains data.
  - The remaining bytes are unused.

Sending Data

- The `UART` class provides two output functions:
  - `ready_for_put()` returns true if the UART is ready to accept more output.
  - `put(c)` sends character `c` to the UART.
- For example, if `port` is a reference to a `UART`, then:

```
while (!port.ready_for_put())
    ;
port.put(c);
```

  waits for the UART to complete the previous transmission, and then initiates another.
- The UART transmits the low-order byte of UTXBUF.
  - It ignores the remaining bytes.

A UART Class in C++

```
class UART {
public:
    ~~~     // see the next few slides
private:
    device_register ULCON;
    device_register UCON;
    device_register USTAT;
    device_register UTXBUF;
    device_register URXBUF;
    device_register UBRDIV;
};
```

Controlling Transmission Speed

- These public members are for enabling and disabling the UART:

```
class UART {
public:
    ~~~
    enum mode { RXM = 1, TXM = 8 };
    void disable() { UCON = 0; }
    void enable() { UCON = RXM | TXM; }
    void init() { enable(); ~~~ }
    ~~~
};
```

- And these are the three i/o functions…

Controlling a UART

```
class UART {
public:
    ~~~
    int get() {
        return (USTAT & RDR) != 0 ?
            static_cast<int>(URXBUF) : -1;
    }
    bool ready_for_put() {
        return (USTAT & TBE) != 0;
    }
    void put(int c) {
        UTXBUF = static_cast<device_register>(c);
    }
    ~~~
};
```

Better I/O Functions

- UARTs are often used for sending and receiving character data.
- Unfortunately, our current UART interface requires that we use a cast to operate in terms of characters, as in:

```
int temp;
char c;
if ((temp = port.get()) != -1) {
    c = static_cast<char>(temp);
    ~~~
}
```

Better I/O Functions

- Here's a better set of UART functions:

```
bool get(char &c) {
    if ((USTAT & RDR) == 0)
        return false;
    c = static_cast<char>(URXBUF);
    return true;
}

void put(char c) {
    UTXBUF = c;
}
```

Better I/O Functions

- This `get` function mimics `std::istream::get`:
  - If it gets a character, it returns true and places the character value in a `char` (not an `int`) argument passed by reference.
  - Otherwise, it returns false.

```
UART &port = *new UART;
~~~
char c;
if (port.get(c)) {
    ~~~
}
```

Ensuring Proper Alignment

- Again, the UART 0 group consists of six device registers:

Offset	Register	Description
0xD000	ULCON	line control register
0xD004	UCON	control register
0xD008	USTAT	status register
0xD00C	UTXBUF	transmit buffer register
0xD010	URXBUF	receive buffer register
0xD014	UBRDIV	baud rate divisor register

- How can you be sure that each structure member is at the correct offset?
- Most compilers offer extensions to help you control structure layout…

Layout Guarantees

- For example, some compilers offer pragmas, as in:

```
#pragma pack(push, 4)
struct UART {
    ~~~
};
#pragma pack(pop)
```

- Some dialects of GNU C and C++ support type attributes such as:

```
struct UART __attribute__(packed) {
    ~~~
};
```

Using Static Assertions to Enforce Layout

- You can catch misalignments at compile time using static assertions, as in:

```
struct UART {
    device_register ULCON;
    device_register UCON;
    ~~~
};
static_assert(
    offsetof(UART, UCON) == 0x04,
    "UCON register must be at offset 4 in UART"
);
~~~
```

Guaranteed Initialization

- A `UART` must be initialized by calling `init` before it can be used.
  - A user could easily forget to do so.

✓ *Interfaces should be:*

✓ ***easy to use correctly***

✓ ***hard to use incorrectly.***

- How can we make sure that initialization for a `UART` is guaranteed?

Constructors

- We could turn `init` into a constructor to ensure that it's always called when a `UART` is created:

```
class UART {
public:
    ~~~
    UART() { enable(); ~~~ }
    ~~~
};
```

- Unfortunately, this doesn't quite work with our definitions…

Constructors

- Memory-mapped objects aren't normal objects.
  - You don't (or at least shouldn't) define any objects of the `UART` type.
  - You just set up pointers or references to existing memory-mapped registers using `reinterpret_cast`.
- This leaves the compiler no opportunity to generate code for constructor calls as it normally would.
- However, you can construct a `UART` object at a memory-mapped location by using a placement new-expression…

Constructors and New-Expressions

- An expression such as:

```
p = new T (v);  // (v) is optional
```

translates into something (sort of) like:

```
p = static_cast<T *>(operator new(sizeof(T)));
p->T(v);    // explicit constructor "call"
```

- The expression `p->T(v)` is a notation for "apply the `T` constructor that accepts argument `v` to the storage addressed by `p`".
  - It doesn't actually compile.
- If the new-expression lacks an initializer such as `(v)`, it uses `T`'s default constructor.

Constructors and Placement New

- The C++ Standard Library provides a version of `operator new` that you can use to "place" an object at a specified location.
- The function is defined as just:

```
void *operator new(std::size_t, void *p) throw () {
    return p;
}
```

- It ignores its first parameter and simply returns its second.
- The empty exception-specification, `throw()`, indicates that the function won't propagate any exceptions.
- This `operator new` is often an inline function, so the compiler can optimize calling the function down to just copying a pointer.

Constructors and Placement New

- A *placement new-expression* such as:

```
p = new (region) T (v);      // (v) is optional
```

translates into something along the lines of:

```
p = static_cast<T *>(operator new(sizeof(T), region));
p->T(v);
```

- In effect, it just constructs a T object in the storage addressed by region.
  - The first line of the translation assigns p the address returned from operator new, namely the value of region.
  - The second line constructs a T object at that address.

UART with Placement New

- Using a placement new-expression, the definition for `UART0` now looks like:

```
UART *const UART0 =
    new(reinterpret_cast<void *>(0x03FFD000)) UART;
```

- As a reference, it looks like:

```
UART &UART0 =
    *new(reinterpret_cast<void *>(0x03FFD000)) UART;
```

Class-Specific New

- C++ lets you declare operator new as a class member.
- If T is a class type with a member operator new, then a new-expression such as:

```
p = new T (v);  // (v) is optional
```

  uses T's operator new rather than the global operator new.
- New-expressions for all other types continue to use the global operator new.

Class-Specific New

- You can implement an operator new as a class member that places a device at a specified memory-mapped address:

```
class UART {
public:
    void *operator new(std::size_t) {
        return reinterpret_cast<void *>(UART0_address);
    }
    ~~~
private:
    enum { UART0_address = 0x03FFD000 };
    ~~~
};
```

Class-Specific New

- Then you can create a fully constructed UART object at the desired memory-mapped location using just:

```
UART &UART0 = *new UART;
```

- This new-expression:
  - uses the UART's operator new to "place" the UART object in its memory-mapped location, and
  - uses the UART's default constructor to initialize the object.
- A member operator new is always a static member, even if not declared so explicitly.
- Unfortunately, this only works for UART 0…

Class-Specific New

- How can we make it work for UART 1 as well?
- You can overload `operator new` with a parameter that specifies the UART number:

```
class UART {
public:
    void *operator new(std::size_t, int n) {
        return reinterpret_cast<void *>(
            UART0_address + n * 0x1000
        );
    ~~~
};
```

Class-Specific New

- Using this `operator new`, you can write:

```
UART &port = new (0) UART;        // use UART0
```

- Unfortunately, this works only when the placement argument is 0 or 1, but it still compiles for other values:

```
UART &port = new (42) UART;       // compiles, but fails
```

- You can't prevent this with a static assertion.
- If you want to restrict the argument to 0 or 1, you must use a run-time check, or…
- You can use an enumeration as the placement parameter type…

Class-Specific New

```
class UART {
public:
    enum uart_number { zero, one };
    void *operator new(std::size_t, uart_number n) {
        return reinterpret_cast<void *>(
            UART0_address + n * 0x1000
        );
    }
    ~~~
private:
    enum { UART0_address = 0x03FFD000 };
    ~~~
};
```

Class-Specific New

- Using this operator new, you can write:

```
UART &port = new (UART::zero) UART;
```

- Now your choice of UART number is limited to only zero (= 0) and one (= 1).
- You can't place a UART somewhere else unless you go way out of your way:

```
UART &port = new (static_cast<UART::number>(42)) UART;
```

- Yet another reason to avoid casts.

Sooner Rather Than Later

- Static (compile-time) checking has advantages over run-time checking:
  - Fixing compile-time errors is almost always faster and easier than finding and fixing run-time errors.
  - Whereas you might be reluctant to pay for a run-time check, compile-time checks are free.
  - Whereas you can ship a program that might fail a run-time check, you can't ship a program that fails a compile-time check.
- Obviously, you can't detect all errors at compile time, but C++ offers lots of ways to turn would-be run-time errors into compile-time errors.

Modeling Devices More Accurately

- Most of the E7T's special registers support both read and write operations.
  - Class members of type `device_register` are read/write by default.
- But not all UART registers are read/write:
  - USTAT and URXBUF are read-only.
  - UTXBUF is write-only.

Modeling Devices More Accurately

- Writing to a read-only register can produce unpredictable misbehavior that can be hard to diagnose.
  - You're much better off enforcing read-only semantics at compile time.
- Fortunately, declaring a member as read-only is easy — just declare it `const`.
- Reading from a write-only register can also produce unpredictable misbehavior that can be hard to diagnose.
  - Again, you're better off catching this at compile time, too.
- Unfortunately, C++ doesn't have a write-only qualifier.
- However, you can enforce write-only semantics by using a class template…

A Write-Only Class Template

- `write_only<T>` is a simple class template for write-only types.
- For any type `T`, a `write_only<T>` object is just like a `T` object, except that it doesn't allow any operations that read the object's value.
- For example,

```
write_only<int> m = 0;
write_only<int> n;
n = 42;
m = n;  // error: attempts to read the value of n
```

A Write-Only Class Template

- The class template definition is:

```
template <typename T>
class write_only {
public:
    write_only() { }
    write_only(T const &v): m (v) { }
    void operator =(T const &v) { m = v; }
private:
    T m;
    // disallow copy operations...
    write_only(write_only const &);
    write_only &operator =(write_only const &);
};
```

Modeling Devices More Accurately

- Using the const qualifier and the `write_only<T>` template, the `UART` class data members look like:

```
class UART {
    ~~~
private:
    device_register ULCON;
    device_register UCON;
    device_register const USTAT;
    write_only<device_register> UTXBUF;
    device_register const URXBUF;
    device_register UBRDIV;
};
```

- However, compilers will complain about this, as they should…

Constructors and Const Members

- In C++, any class with non-static data members of const type must have at least one user-declared constructor:

```
class UART {
public:
    UART();     // no definition needed
    ~~~
    device_register const USTAT;
    ~~~
};
```

- You need not define the constructor provided you never actually define any UART objects.

Constructors and Const Members

- The compiler will complain if you define the constructor like this:

```
struct UART {
    UART() { }  // error (maybe just a warning)
    device_register const ULCON;
    ~~~
};
```

- Every constructor in a class with non-static data members of const scalar type must have a member-initializer for each such member.

Constructors and Const Members

- The const UART members are USTAT and URXBUF.
- If you define a default UART constructor, it should look like:

```
class UART {
public:
    UART(): USTAT (v1), URXBUF (v2) { }
    ~~~
};
```

- Whatever you use for v1 and v2, this is not good:
  - Constructing a memory-mapped UART will write values into read-only registers USTAT and URXBUF.
  - Even if you omit v1 and v2 (which you can do), the constructor will write zeros into those registers.

Constructors and Const Members

- If you don't want your program calling that constructor, then you should use access control to prevent the call:

```
class UART {
    ~~~
private:
    UART();     // no definition needed
    ~~~
    device_register ULCON;
};
```

- However, you won't be able to use member `operator new`.

A Read-Only Class Template

- There's another, better way to enforce read-only semantics.
- You can define a class template `read_only<T>` much like `write_only<T>`, and use that instead of `const`, as in:

```
class UART {
    ~~~
    read_only<device_register> USTAT;
    write_only<device_register> UTXBUF;
    read_only<device_register> URXBUF;
    ~~~
};
```

A Read-Only Class Template

- A `read_only<T>` object is not const:
  - You need not initialize a `read_only<T>` data member.
- Other than the default constructor, `read_only<T>` provides only two operations:
  - A conversion operator that lets you inspect the object, as in:

```
read_only<int> r;
~~~
int i = r;              // OK
```

  - An address-of operator that yields the address of the object as a "pointer to const".

```
int const *p = &r;  // OK
```

A Read-Only Class Template

- The class template definition is:

```
template <typename T>
class read_only {
public:
    read_only() { }
    operator T const &() const { return m; }
    T const *operator &() const { return &m; }
private:
    T m;
    // disallow copy operations...
    read_only(read_only const &);
    read_only &operator =(read_only const &);
};
```

Summary

- You can control memory-mapped i/o using only standardized language features (but they will have platform-specific behavior).
- Put your effort into writing data declarations that model the hardware as precisely as possible.
  - If you do that well, writing the code to manipulate the hardware will be much easier.
- Whatever you do, isolate language and hardware dependencies inside abstract types, either as structures in C or classes in C++.