# Writing Better Embedded Software

Dan Saks

Meeting Embedded
November, 2018

1

## Who Am I to be Speaking to You?

- Software developer from 1975 to 1981
  - programming languages and tools
- University Instructor from 1982 to 1986
  - programming languages
  - data structures
  - operating systems
- Software consultant (as Saks & Associates) from 1987 to 1989
  - embedded systems
  - systems analysis

2

## Who Am I to be Speaking to You?

- Secretary of the C++ Standards Committees from 1990 to 1997
- Co-author of the Plum Hall test suite for C++ from 1992 to 2005
- Contributing Editor/Columnist from 1990 to 2013
  - *The C/C++ Users Journal* (now at *drdobbs.com*)
  - ***Embedded Systems*** [***Programming*** ⇨ ***Design***]
  - ***embedded.com***
  - others
- ***Teaching C++ since 1990***
  - ***to embedded software developers since 1993***

3

## Embedded Systems

- ***embedded system***. *n*. A combination of computer hardware and software, and perhaps additional mechanical or other parts, designed to perform a dedicated function.
  - from *Embedded Systems Dictionary* by Jack Ganssle and Michael Barr. 2003, CMP Books.
- The job of a computer in an embedded system is to be something other than a general-purpose computer.

4

## Sample Embedded Systems

- Consumer products
  - cameras, audio/video players, game systems, home appliances, watches
- Financial equipment
  - ATMs, cash registers, credit card readers
- Industrial automation
  - robots, production monitors
- Medical equipment
  - biometric monitors, imaging equipment

5

## Sample Embedded Systems

- Navigation equipment
  - radar, guidance systems
- Computer peripherals
  - printers, scanners, video boards
- Automotive subsystems
  - braking, entertainment, navigation, steering, traction

6

## Sample Embedded Systems?

- A tablet or other handheld computer is ***not*** an embedded system.
  - It has requirements not uncommon to embedded systems:
    - power consumption
    - heat dissipation
    - communication bandwidth
  - It's really just a general-purpose computer in a small package.
- How about a mobile phone?
  - Yes, if it's just a phone.
  - Probably not, if it's a smart phone.

7

## Very Hard to Generalize

- Embedded systems vary widely.
- Broad statements rarely apply to all embedded systems.
- Take generalizations with a grain of salt.
  - This includes what I'm about to say.
- Embedded designers are more likely to have to think about things that other software developers usually don't…

8

## Possible Economic Concerns

- Development
  - How soon until we get our hands on the first unit?
  - What do we do until then?
- Production
  - How much will it cost to build each unit?
- Operating
  - How much will it cost to run it?

9

## Possible Physical Requirements

- Electrical
  - Does it use too much power?
  - Can it tolerate electrical noise?
- Ruggedness
  - Can it tolerate getting dirty?
  - Can it tolerate shock or vibration?
- Thermal
  - Can it stand the cold or heat?
  - Does it generate too much heat?

10

## Possible Performance Requirements

- Throughput
  - Can it keep up with all the data coming in?
  - How many responses can I get per unit of time?
- Responsiveness
  - How soon until I get a result?
  - Can I get it in ***real time***?

11

## Possible Real Time Requirements

- ***"Hard" real time*** = any late response is intolerable.
  - In some systems, a late response just makes the system unsatisfactory or unusable.
  - In the extremes, a late response could result in physical damage, injury, or death.
- ***"Soft" real time*** = an occasional late response is tolerable.
  - Too many late responses are not.

12

## The "Typical" Developer

- Most have college/university degrees.
  - Often:
    - Electrical Engineering (EE)
    - Computer Engineering (CE)
    - Mechanical Engineering (ME)
- Many have little or no formal training in software analysis, design, and programming.
- Again, this is based on developers I've encountered, not a broad statistical sampling.

13

## The "Typical" Developer

- What about embedded developers with Computer Science (CS) degrees?
  - They used to be rare.
  - They're more common now, especially on larger projects.
- Nonetheless, the EE perspective still dominates the field…

14

## Working, and Working Better

- "It's very rare that you can program an embedded system without understanding the circuitry and what it's trying to accomplish."

  —Mike Willey, hardware guy (CTO, Paragon Innovations)

- This matches my experience…

- "If I were staffing an embedded project, I'd hire a double-E first, and me second.
- "The double-E will make it work; I'll make it work better."

  —Dan Saks, software guy (me)

15

## Too Much for One Person

- Embedded development often requires a broad skill set, including:
  - hardware
  - software
  - mathematics
  - human factors
  - a bunch of other stuff

- It often requires more technical knowledge than is reasonable to expect from one person.

- Teamwork can be essential to success.

16

## Embedded Systems Are Different…

- Again, writing embedded software can be different from writing desktop or server applications.
- Embedded systems often have strict resource limitations, such as:
  - memory space and type
  - communication bandwidth
  - power consumption
- They can have "hard" real-time requirements.
- They often control hardware directly.

17

## …But Not That Different

- Nonetheless…
- Most embedded programming is just plain programming.
- ***Good*** embedded programming is just ***good*** programming.

18

## Unnecessarily Poor Practice

- Unfortunately…
- Too many embedded developers use the differences from more conventional programming to justify unnecessarily poor practices.
- Here's an example…

19

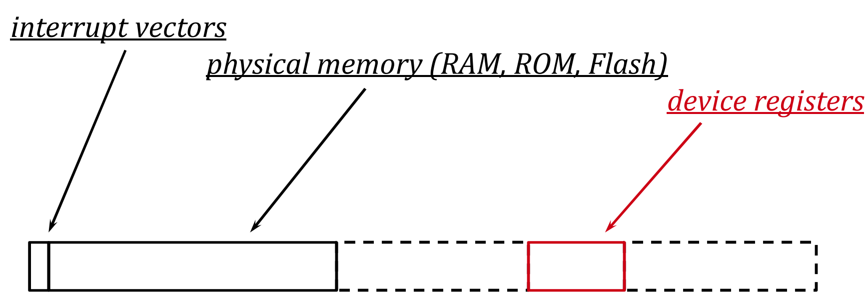
## Direct Hardware Control

- Again, some, possibly many, embedded systems control hardware directly.
- Software typically communicates with hardware devices through ***device registers***.
  - Also known as:
    - ***special function registers*** or
    - ***special registers***.
- Most modern processors use ***memory-mapped addressing***…

20

## Memory-Mapped Addressing

- The architecture disguises the device registers to be addressable like “ordinary” memory:




“Typical” address space

21

## Traditional Register Representation

- Hardware vendor libraries used to represent device register addresses as clusters of related macros.
- The registers often have the same type, such as:

```
#define TMOD    ((unsigned volatile *)0x3FF6000)
#define TDATA   ((unsigned volatile *)0x3FF6004)
```

- The sizes of the built-in scalar types can vary across platforms.
- Many C programmers prefer using ***exact width types***:

```
typedef uint32_t volatile dev_reg;
```

22

## Traditional Register Representation

```
// timer registers
#define TE      0x1                     // bit mask
#define TMOD    ((dev_reg *)0x3FF6000)  // address
#define TDATA   ((dev_reg *)0x3FF6004)  // address
~~~

// UART0 registers
#define ULCON0  ((dev_reg *)0x3FFD000)  // address
#define UCON0   ((dev_reg *)0x3FFD004)  // ~~~
~~~

// UART1 registers
#define ULCON1  ((dev_reg *)0x3FFE000)
#define UCON1   ((dev_reg *)0x3FFE004)
~~~
```

23

## Accessing Device Registers

- You can use these macros to fiddle with the registers:

```
*TMOD |= TE;     // OK: set the timer enable bit

*UTXBUF0 = c;    // OK: write c's value to UART0
```

24

## Too Easy to Use Incorrectly

- Unfortunately, using these macros is very error-prone:

```
void UART_put(dev_reg *stat, dev_reg *txbuf, int c);
~~~

UART_put(UTXBUF0, USTAT0, c);   // wrong order

UART_put(USTAT0, UTXBUF1, c);   // mismatching UART #s

UART_put(TMOD, UTXBUF1, c);     // wrong device
```

- The above calls will compile, but will have to be debugged.
- Wouldn't it be better if these calls simply didn't compile?

25

## An Unfortunate Mindset

- C programmers in general, and embedded developers in particular, just accept that code with errors might still compile.
- This leads to a fatalistic attitude…
- Just get the code to compile, so you can get to the real work… …debugging.

26

## A Different Focus on Tools

- Embedded developers rely heavily on run-time debugging tools such as:
  - debuggers
  - in-circuit emulators
  - logic analyzers
  - protocol analyzers
  - oscilloscopes
- Many are skeptical compile-time type checking and static analysis can improve the situation.
- In fact, designing a better interface is actually fairly easy…

27

## Using Structures is Better

- Cluster the registers into structures:

```
struct timer {
    dev_reg TMOD;
    dev_reg TDATA;
    dev_reg TCNT;
};

void timer_enable(timer *t);
uint32_t timer_get(timer *t);
```

- I'll address legitimate concerns about structure storage layout a little later.

28

## Using Structures is Better

- This, too, is better:

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

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

29

## Easier to Use Correctly

- Using structures is better because it simplifies device interfaces.
  - The caller no longer needs to know which specific registers a given operation uses.
- You can pass all the registers for a device as a single unit:

```
UART *const com0 = (UART *)0x3FFD000;
~~~
UART_put(com0, c);      // put c to a UART object
```

- And this is still *just* C.

30

## Harder to Use Incorrectly?

- Each structure type has a distinct type.
- Type checking can now catch accidents such as this:

```
UART *const com0 = (UART *)0x3FFD000;
timer *const timer0 = (timer *)0x3FF6000;
~~~
UART_put(timer0, c);    // compile error?
UART_put(com0, c);      // OK: can put to a UART
```

- Maybe…

31

## Harder to Use Incorrectly?

- This is an aspect where C and C++ differ.
- A C++ compiler ***will*** flag the first call as an error:

```
UART *const com0 = (UART *)0x3FFD000;
timer *const timer0 = (timer *)0x3FF6000;
~~~
UART_put(timer0, c);    // error in C++; warning in C
UART_put(com0, c);      // OK: can put to a UART
```

- A C compiler ***might*** issue a warning.
- It probably will, but it doesn't have to.

32

## But, But, But…

- “But I can get better type checking with C by using a static analyzer.”
- But you can’t get nearly as much with C as you can with C++.

33

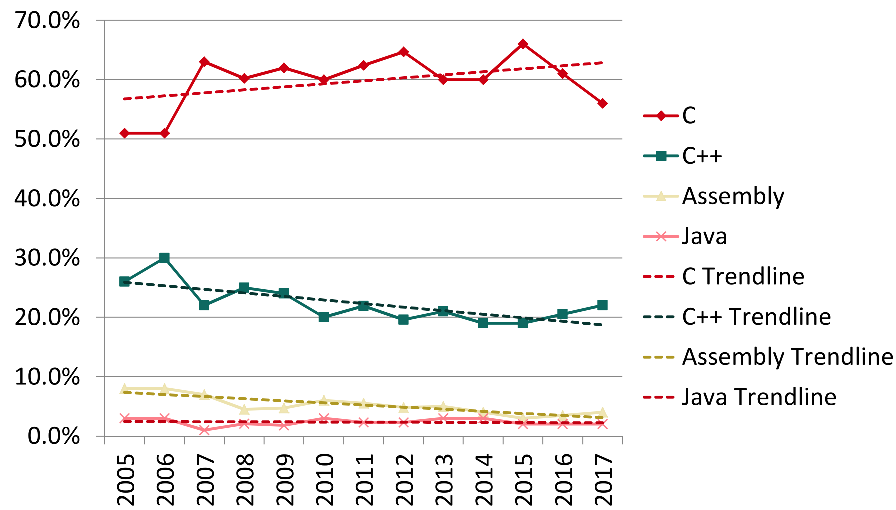
## Here’s Where We Are

- More embedded developers use C than anything else.
  - By far.
- *embedded.com*’s annual reader survey asks participants to complete this sentence:

  *“My current embedded project is programmed mostly in…”*

34

## It's Mostly C, Some C++, and Not Much Else




35

## Developers and Their Tools

- In general, language tools for embedded systems lag behind those for the desktop.
- For example:
  - C wasn't widely available for embedded development until a few years after it was established on the desktop.
  - Vendors were so slow to implement aspects of C99 (e.g., VLAs), C11 made them optional.
  - Until this year, I still had clients who restricted their C++ usage to C++03.

36

## Developers and Their Tools

- Why the lag?
- My speculation:
  - The embedded software market doesn't offer the economies of scale of the desktop and server market.
- My observation:
  - Embedded systems developers are more cautious about embracing new software tools and methods.

37

## I'm Not Making This Up

- From an email I just received last week:
  - *"I have heard many C programmers state the concern, 'If I start a project by moving to C++ and it doesn't work out [ed. C++ gets too complex], I won't be able to come back to C.'"*

38

## Loss Aversion

- From psychology, behavioral economics and decision theory:
  - Fear of loss > desire for gain
- Possibly:
  - Fear of loss == 2 * (desire for gain)
- What to do?
  - Be sensitive to this concern.
  - Don't get impatient with people, even if you think they're being irrational.

39

## On Being Persuasive

- "So the only way … to influence other people is to talk about what they want and show them how to get it."
  - — Dale Carnegie: *How to Win Friends and Influence People*

40

## A Change in Thinking

- Moving from C to C++ requires a change in thinking.
- "Make interfaces easy to use correctly and hard to use incorrectly."

  —Scott Meyers, *The Most Important Design Guideline?*
- C++ makes this more attainable by providing a more robust type system…

41

## Static Data Types

- For the most part, C and C++ use ***static data types***.
- An object's declaration determines its static type:

```
int n;          // n is "[signed] integer"
double d;       // d is "double-precision floating point"
char *p;        // p is "pointer to character"
```

- An object's static type doesn't change during program execution.
- It doesn't matter what you try to store into the object.
  - The type doesn't change.

42

## What's a Data Type?

- A ***data type*** is a bundle of compile-time properties for an object:
  - size and alignment
  - set of valid values
  - set of permitted operations

43

## What's a Data Type?

- On a typical 32-bit processor, type `int` has:
  - size and alignment of 4 (bytes)
  - values from -2147483648 to 2147483647, inclusive
    - integers only
  - operations including:
    - unary `+ - ! ~ & ++ --`
    - binary `= + - * / % < > == != & | && ||`

44

## What's a Data Type?

- What a type can't do is as important as what it can.
- An int can't do…

```
*i      // indirection (as if a pointer)

i.m     // member selection

i()     // call (as if a function)
```

- This is a big difference between C++ and C:
  - C++ will reject at compile-time questionable operations that C will accept.

45

## Implicit Type Conversions

- A type's operations may include implicit type conversions to other types:

```
int i;
long int li;
double d;
char *p;
~~~
li = i;     // OK: convert int into long int
d = i;      // OK: convert int into double
d = p;      // error: can't convert pointer into double
```

- Here, again, C++ will reject at compile time questionable conversions that C will accept.

46

## The Real Change in Thinking

- Again, moving from C to C++ requires a change in thinking…
- It's learning to use the type system to turn potential run-time errors into compile-time errors.
  - Fixing compile-time errors is easier than diagnosing and fixing run-time errors.
  - It's easy to ship a program with run-time errors.
  - It's much harder to ship a program that doesn't compile.

47

## Another Benefit

- Type information supports operator overloading:

```
char c, d;
int i, j;
double x, y;
~~~
c = d;          // char = char
i = j + 42;     // int = (int + int)
x = y + 42;     // double = (double + int)
```

- Both C and C++ do this.
- But C++ lets you extend this to user-defined types.

48

## A Bar Too High?

- The C++ community may be making it harder for embedded developer to embrace C++ by setting the bar too high…

49

## A Bar Too High?

- The "modern" approach to teaching C++:
  - Use streams instead of FILEs.
  - Use vectors instead of arrays.
  - Use strings instead of null-terminated character sequences.
- For non-C programmers, this is probably the best approach.
- I spend a lot of time teaching C programmers who make a living writing code for embedded systems.
- This is not the approach I use.

50

## A Bar Too High?

- C++ was once a "Better C".
- Now, it's touted as a "new language".
- That C++ is a "Better C" may be why C++ is as popular as it is.
  - Ironically, many in the C++ community now discount this aspect of C++.
- I'm not suggesting that you teach C before teaching C++.
- I am suggesting that you teach C++ to working C programmers by starting with what they know and helping them reshape it.

51

## A Bar Too High?

- Some, possibly many, projects stay with C because they can't bridge the widening gap to C++.
- For many current C users, especially embedded developers, moving incrementally from C to C++ is probably much more practical.

52

## Other Pragmatic Concerns

- Legacy embedded code:
  - Most of it is in C.
  - It's too valuable to discard.

- Learning schedules:
  - For even experienced C programmers, learning most of C++ takes two or three work weeks.
  - Few teams can block out that much time at once.
  - They need to learn C++ in shorter sessions.
  - Each course must cover something they can use right away.

53

## A Legitimate Cause for Concern

- Earlier, I recommended using structures to represent memory-mapped devices:

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

- Some programmers are reluctant to use these because they've been burned…

54

## A Legitimate Cause for Concern

- Using macros, you can place each register at its exact address:

```
// UART0 registers
#define ULCON0  ((dev_reg *)0x3FFD000)
#define UCON0   ((dev_reg *)0x3FFD004)
~~~
```

- With a structure, the compiler might insert unused padding bytes after any member.
- How do you prevent this, and do it cheaply?

55

## Use Static Assertions

- You can use a static assertion to check that each structure member is at the expected offset:

```
struct UART {
    dev_reg ULCON;
    dev_reg UCON;
    ~~~
};
static_assert(
    offsetof(UART, UCON) == 4,
    "UCON member of UART is at the wrong offset"
);
```

- Doing this for all the members usually isn't necessary…

56

## Use Static Assertions

- You can just check that there's no padding anywhere in the structure:

```
struct UART {
    dev_reg ULCON;
    dev_reg UCON;
    ~~~
};
static_assert(                          // no padding
    sizeof(UART) == 6 * sizeof(dev_reg),
    "UART contains extra padding bytes"
);
```

57

## Further Constraining What You Can Do

- Thus far, code in this example compiles in either C or C++.
- However, using a structure for an entire device still leaves the individual registers exposed to misuse.
- Rather, you can use a C++ class with private members to cut down on improper register accesses…

58

## Using Classes is Even Better

```
class UART {
public:
    void put(int c);
    int get();
    ~~~
private:                // even better
    dev_reg ULCON;
    dev_reg UCON;
    ~~~
};
~~~

com0->put(c);
```

59

## Using Classes is Even Better

- How much more does it cost to use a class instead of a structure?
  - Zero. Zip. Zilch. Nothing. Nil. Nada.
- The code is essentially the same size and speed either way.
- Sometimes, the C++ version is even faster.

60

## Not Yet at the Point of No Return

- By the way, converting back to C is still pretty easy:

```
com0->put(c);          // C++

UART_put(com0, c);     // equivalent C
```

61

## A Mistrust of Abstractions

- Again, some embedded developers are very forward-looking.
  - They're eager for better methods and tools.
- However, many have a deep-seated mistrust of abstractions.
  - This is somewhat surprising…
  - They're in the business of automating manual tasks.
- This mistrust shows in one reader's response to a column I wrote on interrupt handling a while back.
- Here's more or less what I explained…

62

## Interrupt Handling

- Most processors support devices that issue interrupts:
  - A device notifies the processor by issuing an ***interrupt request***.
  - The processor responds by transferring control to:
    - an ***interrupt service routine (ISR)*** or
    - an ***interrupt handler***.

63

## Interrupt Handling

- Most processors:
  - convert the requested signal into an ***interrupt number***, and
  - use that number to index into an ***interrupt vector table (IVT)***.
- The IVT is usually a table of function addresses in low memory.

64

## Registering a Handler

- For example, a typical processor supports interrupt numbers from 0 to 7, inclusive.
- In that case, the IVT might be a table of eight 4-byte pointers starting at a low memory address, say 0x20.
- To prepare to handle interrupt request 6, you have to store a function address into location 0x20 + 6 × 4 = 0x38.

65

## Registering a Handler

- Here's how an EE colleague of mine first showed me to do it:

```
*(void **)0x38 = (void *)IRQ_handler;
```

- `IRQ_handler` is a function:

```
void IRQ_handler();
```

- The code worked in this case, but:
  - It's cryptic.
  - Strictly speaking, it has undefined behavior…

66

## Undefined Behavior

- `IRQ_handler` is a function.
- When you use a function name in an expression, the compiler treats it like a pointer — a "pointer to function".
- `void *` is a "pointer to data".
- The cast-expression on the right converts a "pointer to function" into "pointer to data":

```
*(void **)0x38 = (void *)IRQ_handler;
```

- The cast has undefined behavior.

67

## A Better Way

- Rather, define an alias for a "pointer to handler" type, either:

```
typedef void (*ptr_to_handler)();   // C++03 or C++11
using ptr_to_handler = void (*)();  // C++11
```

- Using the alias simplifies the assignment:

```
*(void **)0x38 = (void *)IRQ_handler;       // before
*(ptr_to_handler *)0x38 = IRQ_handler;      // after
```

- In C++, a new-style cast is probably better:

```
*reinterpret_cast<ptr_to_handler *>(0x38) = IRQ_handler;
```

68

## An Even Better Way

- We can do better…
- Start by defining the interrupt numbers as symbolic constants:

```
enum interrupt_number {
    reset,
    undefined_instruction,
    SWI,
    prefetch_abort,
    data_abort,
    reserved,   // for future use
    IRQ,        // "plain" device interrupts
    FIQ         // "fast" device interrupts
};
```

69

## An Even Better Way

- Define a pointer to the initial interrupt vector in the IVT:

```
ptr_to_handler *const IVT = (ptr_to_handler *)0x20;
```

- This is valid C as well as C++.
- Modern C++ programmers probably prefer:

```
auto const IVT
    = reinterpret_cast<ptr_to_handler *>(0x20);
```

70

## An Even Better Way

- Either way, you can now install the handler using just:

```
IVT[IRQ] = IRQ_handler;         // OK in C or C++
```

- This is quite an improvement over the original:

```
*(void **)0x38 = (void *)IRQ_handler;
```

- What's not to like?

71

## A Mistrust of Abstractions

- Here's how one reader responded…
- "Dan Saks thinks we should have tidy interrupt vector code like

```
IVT[IRQ] = IRQ_handler;
```

  instead of crude stuff like

```
*(void **)0x38 = (void *)IRQ_handler;
```

  I think I disagree…"

72

## A Mistrust of Abstractions

- [Me] You can't make this stuff up…

  "If you're using a well-known commercial environment you trust, and they have a cute mechanism like the first example, perhaps. But if you're rolling your own, I'd stick with the crude weird stuff — *because it'll be easier to figure out at debug time* [my emphasis], which is the most important part particularly with interrupts."

- Is this an extreme example?
  - Yes. The entertaining ones usually are.
- However, it's just the far end of a continuous spectrum.

73

## The Mistrust Runs Even Deeper

- I still see programmers writing code like this:

```
if ((48 <= c) && (c <= 57))     // is c a digit?
```

- This is better:

```
if (('0' <= c) && (c <= '9'))   // is c a digit?
```

- This is even better:

```
if (isdigit(c))                 // probably faster, too
```

74

## Really Earning Trust

- Let's look again at the interrupt vector example:

```
IVT[IRQ] = IRQ_handler;         // C or C++
```

- What could go wrong?
- You could accidentally use an invalid index:

```
IVT[42] = IRQ_handler;          // oops
```

- How can you prevent that cheaply and reliably?

75

## An Even Better Way

- Recall our enumeration of the interrupt numbers:

```
enum interrupt_number {
    reset,
    undefined_instruction,
    SWI,
    prefetch_abort,
    data_abort,
    reserved,   // for future use
    IRQ,        // "plain" device interrupts
    FIQ         // "fast" device interrupts
};
```

- Let's wrap this in a class…

76

## An IVT Class

- ... with an `operator[]` that accepts only interrupt numbers:

```
class IVT {
public:
    using pointer = void (*)();
    enum number {                  // was interrupt_number
        begin, reset = begin, ~~~, IRQ, FIQ, end
    };
    pointer &operator[](number n) {
        return table[n];
    }
private:
    pointer table[end - begin];
};
```

77

## An IVT Class

- You can define a constant pointer to the IVT as either:

```
auto const the_ivt = reinterpret_cast<IVT *>(0x20);
```

- But then the indexing operation looks a little odd:

```
(*the_ivt)[IVT::IRQ] = IRQ_handler;
```

- Using a reference is probably better...

78

## An IVT Class

- Notice the * operator in front of the cast:

```
IVT &the_ivt = *reinterpret_cast<IVT *>(0x20);
```

- This looks right:

```
the_ivt[IVT::IRQ] = IRQ_handler;    // yes!
```

- And it's harder to screw up:

```
the_ivt[42] = IRQ_handler;          // compile error!
```

79

## Parting Thoughts

- Embedded development environments and embedded software developers vary widely.
- However, the developers often share common characteristics:
  - Greater concern for hardware issues.
  - Often justified paranoia about resource scarcity.
  - Wariness of new languages and techniques.

80

## Parting Thoughts

- To improve the development process:
  - Learn to meet other software developers on their ground.
    - Respect their expertise.
    - Respect their concerns.
  - Apply gentle, but steady, pressure to embrace improved tools and techniques.

81

## Migrating from C to C++

- Focus on the parts of C++ that turn:
  - potential run-time errors into compile-time errors
  - run-time computations into compile-time computations
- In particular, focus on:
  - enumerations
  - (lvalue) reference types
  - `const` and `constexpr`
  - function and operator overloading
  - classes as structures with:
    - constrained behavior
    - guaranteed initialization and destruction

82

## Thanks for Listening



## Saks & Associates

- 
- You are free to use them for self study.
- If you'd like permission to use these notes for other purposes, or for information on our training and consulting services, contact:

Saks & Associates
393 Leander Drive
Springfield, OH 45504-4906 USA
+1-937-324-3601
service@saksandassociates.com

