ACCU 2015

Embedded Programming Death Match: C vs. C++

Dan Saks
Saks & Associates
www.dansaks.com

1

Abstract

C offers various ways to represent and manipulate hardware devices. C++ offers additional object-oriented techniques that provide higher levels of abstraction. Many C programmers assert that using C++ objects for hardware accesses is too costly, yet they can offer no measurements to back that claim.

This session explains how to actually measure such claims. It also presents results from some measurements that show, at least for ARM processors, that some widely-used C techniques are actually slower than straightforward C++ techniques.

2

About Dan Saks

Dan Saks is the president of Saks & Associates, which offers training and consulting in C and C++ and their use in developing embedded systems.

Dan has written columns for numerous print publications including *The C/C++ Users Journal*, *The C++ Report*, *Software Development*, and *Embedded Systems Design*. He is on sabbatical from writing the online "Programming Pointers" column for *embedded.com*. With Thomas Plum, he wrote *C++ Programming Guidelines*, which won a *1992 Computer Language Magazine Productivity Award*. He has also been a Microsoft MVP.

Dan has taught thousands of programmers around the world. He has presented at conferences such as *Software Development* and *Embedded Systems*, and served on the advisory boards for those conferences.

3

About Dan Saks

Dan served as secretary of the ANSI and ISO C++ Standards committees and as a member of the ANSI C Standards committee. More recently, he contributed to the *CERT Secure C Coding Standard* and the *CERT Secure C++ Coding Standard*.

Dan collaborated with Thomas Plum in writing and maintaining *Suite++™, the Plum Hall Validation Suite for C++*, which tests C++ compilers for conformance with the international standard. He was a Senior Software Engineer for Fischer and Porter (now ABB), where he designed languages and tools for distributed process control. He also worked as a programmer with Sperry Univac (now Unisys).

Dan earned an M.S.E. in Computer Science from the University of Pennsylvania, and a B.S. with Highest Honors in Mathematics/ Information Science from Case Western Reserve University.

4

Legal Stuff

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

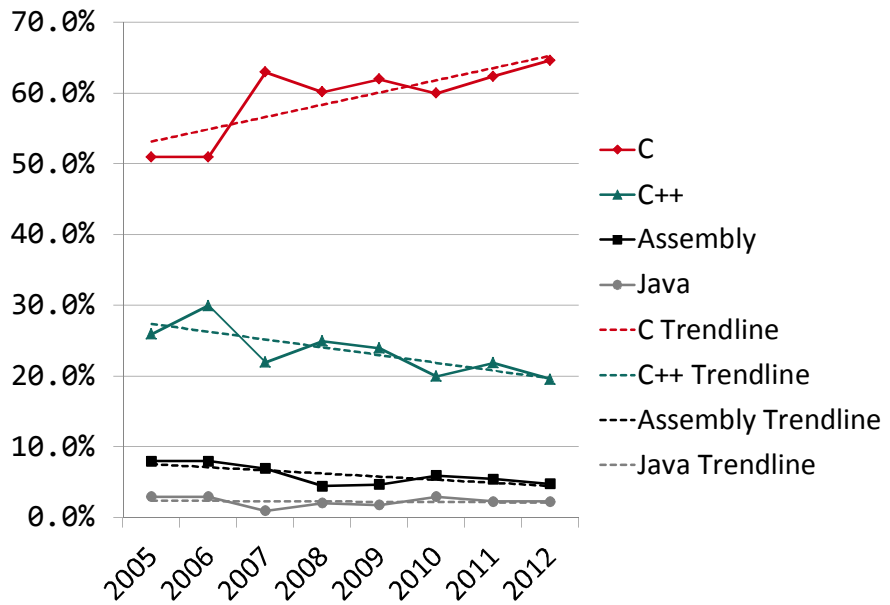
5

6

C vs. C++ for Embedded Programming

- Each year, *embedded.com* surveys its readers to complete the sentence ***"My current embedded project is programmed mostly in…"***
- Here are some fairly recent responses…

7



8

C vs. C++ for Embedded Programming

- ***C*** is clearly the dominant language for programming embedded systems.
- ***C++*** is a distant second.
- ***Assembler*** is a more distant third.
- ***Java*** is an even more distant fourth.
- No other programming language is the primary choice for more than 2% of embedded projects.

9

C vs. C++ for Embedded Programming

- My experience is that many embedded developers use C because:
  - It's what they know.
  - It's what "everybody does".
  - They've heard too many bad things about C++.

10

C vs. C++ for Embedded Programming

- Many embedded developers are EEs or CEs by training.
- Many have no formal software training.
  - They taught themselves C.
- I suspect:
  - They expect to be able to teach themselves C++.
  - They find self-taught C++ to be too difficult.

11

Device Addressing Concepts

- Device drivers communicate with hardware devices through ***device registers***.
- These days, the most common device addressing scheme is…
- ***Memory-mapped i/o (memory-mapped device addressing)***:
  - It maps device registers into the conventional data space…
  - typically at high addresses, out of the way of physical memory.

12

Device Addressing Concepts

- For example, suppose the hardware interface to a programmable timer uses three 32-bit registers:
  - TMOD at location 0x3FF6000
  - TDATA at location 0x3FF6004
  - TCNT at location 0x3FF6008
- You can model these registers as a C or C++ structure:

```
typedef struct timer_registers timer_registers;
struct timer_registers {
    uint32_t TMOD;
    uint32_t TDATA;
    uint32_t TCNT;
};
```

13

Device Addressing Concepts

- You can declare a `timer_registers` object, as in:

```
timer_registers timer0;
```

- Now you can read the timer by a simple statement, such as:

```
ticks = timer0.TCNT;
```

- Unfortunately, the compiler places the `timer0` object where it wants, not where you want.
- You'll have a set of timer registers in ordinary memory, not in the mapped memory.
- Your timer code won't work.
- Let's try something else…

14

Device Addressing Concepts

- Declare `timer0` as a "pointer to timer_registers":

```
timer_registers *timer0;
```

- You can use dynamic memory allocation to acquire a value for the pointer, as in:

```
timer0 = malloc(sizeof(timer_registers));    // in C
```

or:

```
timer0 = new timer_registers;                // in C++
```

- `timer0` now points to a `timer_registers` object…

15

Device Addressing Concepts

- If `timer0` is a pointer, code to read the timer looks like:

```
ticks = timer0->TCNT;
```

- Unfortunately, the memory allocator places the `timer0` object where it wants, not where you want.
- Your timer code still won't work.
- Here's a common technique that does work…

16

Device Addressing Concepts

- Declare `timer0` as a "pointer to timer_registers", initialized with the address of the first timer register:

```
timer_registers *timer0 = (timer_registers *)0x03FF6000;
```

- Now `timer0` really points to the memory-mapped timer registers.
- Code to read the timer looks like:

```
ticks = timer0->TCNT;
```

- And it works!

17

What Prompted This Investigation

- In May 2010, I wrote a column on representing and manipulating memory-mapped devices in C. [Saks 2010a]
- I suggested that structures are the best way to represent device registers.
- In June, I explained why C++ classes are even better than C structures for this purpose. [Saks 2010b]

18

What Prompted This Investigation

- A few readers alleged:
  - Using a pointer to access a C++ class object somehow adds unnecessary indirect addressing.
  - The extra indirection incurs a performance penalty.
- Interestingly, no one complained that using a pointer to access a C structure incurs a similar performance penalty.
- What's the real concern?
  - Is it that using a pointer to access memory-mapped objects more costly than using other means?
  - Or, is it that using a C++ class is more expensive than using a C structure?

19

Is C++ Fast Enough for Embedded Programming?

Dan Saks
Saks & Associates
www.dansaks.com

20

What Prompted This Investigation

- I suspect the concern was about the alleged cost of using pointers.
- But maybe using C++ classes is more expensive than using comparable C structures.
- I measured both. [Saks 2010c].

21

Your Mileage May Vary

- Different processors support different addressing modes:
  - Some (such as ARM) are better at base+offset addressing.
  - Others may be better at absolute addressing.
- Some compilers may be better than others at leveraging the addressing modes on the target processor.
- Measured results will likely vary with compilers and target processors.
- You might want to run tests like these yourself.
- Thus, I'll explain how and why I designed the test programs as I did.

22

Measuring Instead of Speculating

Dan Saks
Saks & Associates
www.dansaks.com

23

Test Design Considerations

- For my tests, I used a 50MHz ARM processor with:
  - 512K of memory
  - a small assortment of memory-mapped devices
- I used three different compilers, each:
  - from a different vendor
  - of different vintage (2000, 2004, 2010)
  - supporting both C and C++
- I compiled using:
  - the ARM (rather than THUMB) instruction set
  - little-endian byte ordering
  - optimization for maximum speed
- I didn't turn the instruction cache on.

24

Test Design Considerations

- All of the tests are variations on the same theme.
- The `main` function:
  - repeatedly calls a function that accesses a memory-mapped device, and
  - counts the number of calls it makes in a given span of time.
- Here's what it looks like…

25

```
int main() {
    unsigned seconds = 0;
    unsigned iterations = 0;
    timer_counter prior = /* timer counts per second */;
    // enable the device
    // start the timer counting down from prior
    while (seconds < 15) {
        timer_counter next = /* current timer value */;
        if (next > prior)
            ++seconds;
        prior = next;
        // access the device
        ++iterations;
    }
    return 0;
}
```

26

Test Design Considerations

- Each program differs in:
  - how it represents the device registers,
  - how it accesses those registers, and
  - whether the access functions are inline or not.

27

Implementation Choices

- Each program tests either:
  - a ***polystate implementation***, which supports multiple instances of a given kind of device, or
  - a ***monostate implementation***, which supports only a single instance of a given kind of device.
- The monostate implementations came in two flavors:
  - a ***bundled monostate***, which wraps the device registers in a single structure, or
  - an ***unbundled monostate***, which declares each register separately.

28

Polystate Implementation

- A ***polystate*** implementation of a memory-mapped device uses a C structure or C++ class accessed via a pointer or reference.
- For example, a C polystate implementation for a programmable timer includes a structure definition:

```
typedef uint32_t volatile device_register;

typedef struct timer_registers timer_registers;
struct timer_registers {
    device_register TMOD;
    device_register TDATA;
    device_register TCNT;
};
```

29

Polystate Implementation

- It also includes additional constants, types and functions:

```
#define TE 0x01
#define TICKS_PER_SEC 50000000

typedef uint32_t timer_count_type;

void timer_disable(timer_registers *t);
void timer_enable(timer_registers *t);
void timer_set(timer_registers *t, timer_count_type c);
timer_count_type timer_get(timer_registers const *t);
```

30

- A C++ polystate implementation for the same timer is a just single class:

```
class timer_registers {
public:
    enum { TICKS_PER_SEC = 50000000 };
    typedef uint32_t count_type;
    void disable();
    void enable();
    void set(count_type c);
    count_type get() const;
private:
    enum { TE = 0x01 };
    device_register TMOD;
    device_register TDATA;
    device_register TCNT;
};
```

31

Polystate vs. Monostate

- It was this C++ class definition that provoked the reader comments that it depended on unnecessary indirection.
- Allegedly, monostate implementations eliminate the need for pointers or references to access the device registers.

32

Implementation Choices

- For each design, I wrote a C implementation and a C++ implementation.
- For each implementation, I wrote a version that used inline functions and another that did not.
- Thus, for example, I wrote:
  - a ***polystate*** implementation in ***C*** with ***inline*** functions,
  - a ***polystate*** implementation in ***C*** with ***non-inline*** functions,
  - a ***polystate*** implementation in ***C++*** with ***inline*** functions, and
  - a ***polystate*** implementation in ***C++*** with ***non-inline*** functions.
- I wrote a parallel set of tests using a ***bundled monostate***.
- I wrote a another parallel set of tests using an ***unbundled monostate***.

33

Implementation Choices

- Not all the C compilers I used support C99's `inline` keyword.
- I implemented inline functions in C using function-like macros.

34

Pointer-Placement

- Again, Standard C and C++ won't let you declare a memory-mapped object at a specified absolute address
- However, you can use ***pointer-placement*** to access the registers via a pointer whose value is the specified address.
- For example, to access a `timer_registers` object residing at location 0x03FF6000, you can declare a pointer called `the_timer` as a macro:

```
#define the_timer ((timer_registers *)0x03FF6000)
```

35

Pointer-Placement

- You can use a constant pointer instead of a macro:

```
timer_registers *const the_timer
    = (timer_registers *)0x03FF6000;
```

- My C tests mostly used this form for pointer-placement.
- However, I used macros in a few C test cases and saw no difference in the generated code.
- In C++, I wrote the pointer casts using the `reinterpret_cast` operator, as in:

```
timer_registers *const the_timer
    = reinterpret_cast<timer_registers *>(0x03FF6000);
```

36

Reference-Placement

- In C++, you can use ***reference-placement*** instead of pointer-placement, as in:

```
timer_registers &the_timer
    = *reinterpret_cast<timer_registers *>(0x03FF6000);
```

37

Linker-Placement

- As an alternative to pointer- and reference-placement, you can use ***linker-placement***.
- That is, you can declare a memory-mapped object using a standard `extern` declaration such as:

```
extern timer_registers the_timer;
```

- You then use linker commands to force `the_timer` into the desired address.

38

Unbundled Monostate

- You can use linker-placement to implement the timer registers as an unbundled monostate in C:

```
extern device_register TMOD;
extern device_register TDATA;
extern device_register TCNT;

void timer_disable();
void timer_enable();
void timer_set(timer_count_type c);
timer_count_type timer_get();
```

- You can also use linker-placement to implement an unbundled monostate in C++...

39

- A C++ unbundled monostate implementation uses static data members (and static member functions):

```
class timer_registers {
public:
    enum { TICKS_PER_SEC = 50000000 };
    typedef uint32_t count_type;
    static void disable();
    static void enable();
    static void set(count_type c);
    static count_type get() const;
private:
    enum { TE = 0x01 };
    static device_register TMOD;
    static device_register TDATA;
    static device_register TCNT;
};
```

40

Unbundled vs. Bundled Monostate

- An unbundled monostate declares each register separately:

```
extern device_register TMOD;
extern device_register TDATA;
extern device_register TCNT;
```

- A bundled monostate wraps the registers into a single structure:

```
typedef struct timer_registers timer_registers;
struct timer_registers {
    device_register TMOD;
    device_register TDATA;
    device_register TCNT;
};
```

41

Bundled Monostate

- Here's the timer registers as a bundled monostate in C:

```
typedef struct timer_registers timer_registers;
struct timer_registers {
    device_register TMOD;
    device_register TDATA;
    device_register TCNT;
};

extern timer_registers the_timer;

void timer_disable();
void timer_enable();
void timer_set(timer_count_type c);
timer_count_type timer_get();
```

42

Bundled Monostate

- In C++, a bundled monostate uses a nested private structure:

```
class timer_registers {
public:
    ~~~
private:
    enum { TE = 0x01 };
    struct bundle {
        device_register TMOD;
        device_register TDATA;
        device_register TCNT;
    };
    static bundle b;
};
```

43

At-Placement

- Some C and C++ compilers provide ***at-placement*** via language extensions that let you position an object at a specified memory address.
- For example, to declare a `timer_registers` object residing at location 0x03FF6000, you might write something in one of these forms:

```
timer_registers the_timer @ 0x03FF6000;

timer_registers the_timer _at(0x03FF6000);

timer_registers the_timer
    __attribute__((at(0x03FF6000)));
```

44

Placement Choices

- All three tested compilers support pointer-placement.
  - They should because it's standard.
- All also support reference-placement.
  - C++ only.
- Only one compiler supports at-placement.
- I suspect all the compilers support linker-placement.
  - I could figure out how to do it with only one.
  - However, I easily simulated linker-placement…
  - …and made the tests more portable in the process…

45

A Software "Device"

- To time the execution of function calls that manipulate memory-mapped devices, I needed just two devices:
  - a timer
  - something else (to time)
- My hardware platform has a small assortment of devices such as lights, switches and serial ports.
- However…
  - I wanted these tests to be easy to migrate.
  - I couldn't rely on any these devices being available elsewhere.

46

A Software "Device"

- Rather than use a real hardware device:
  - I invented a software "device" that manipulates memory.
  - I wrote the tests to address the "device" as if it really were a memory-mapped device.
- The "device" is a Fibonacci sequence generator:
  - Each "get" operation applied to the device returns the next number in the Fibonacci sequence…

47

```
// Fibonacci "device" in C++ (polystate with inlining)

class fibonacci_registers {
public:
    void enable() {
        a0 = 0;
        a1 = 1;
    }
    unsigned get() {
        unsigned a2 = a0 + a1;
        a0 = a1;
        a1 = a2;
        return a0;
    }
private:
    unsigned volatile a0, a1;
};
```

48

A Software "Device"

```
// Fibonacci "device" in C
// (polystate with inline member functions as macros)

typedef struct fibonacci_registers fibonacci_registers;
struct fibonacci_registers {
    unsigned volatile a0, a1;
};

// more ...
```

49

A Software "Device"

```
// Fibonacci "device" in C (continued)
// (polystate with inline member functions as macros)

#define fibonacci_enable(_this) \
    ((_this)->a0 = 0, (_this)->a1 = 1)

#define fibonacci_get(_this, v) do { \
    unsigned a2 = (_this)->a0 + (_this)->a1; \
    (_this)->a0 = (_this)->a1; \
    (_this)->a1 = a2; \
    *(v) = (_this)->a0; \
} while (0)
```

50

A Software "Device"

- The address range of the memory on my target evaluation board is from 0 to 0x7FFFF.
- My test programs didn't use any memory at the higher addresses, so I placed the `fibonacci_registers` object at 0x7FF00.
- For example, C test cases that use pointer-placement declare a pointer to the `fibonacci_registers` using a declaration such as:

```
fibonacci_registers *const fibonacci
    = (fibonacci_registers *)0x7FF00;
```

- Programs that use at-placement declare the device as:

```
fibonacci_registers fibonacci @ 0xF7700;
```

51

Default-Placement

- Using the Fibonacci "device" let me fake linker-placement when I couldn't figure out the linker commands to really do it.
- With linker-placement, the test program declares the "device" using the extern declaration:

```
extern fibonacci_registers fibonacci;
```

52

Default-Placement

- When I couldn't use genuine linker-placement, I simply wrote the definition for the object in another source file, as in:

```
// fibonacci.c

#include "fibonacci.h"

fibonacci_registers fibonacci;
```

- The linker places the compiled definition for the `fibonacci` object among the other global objects in the program, not at 0x7FF00.
- I call this technique ***default-placement***.

53

Default-Placement

- Using default-placement let me measure the performance of linker-placement without actually using linker-placement.
- On the one compiler that I actually tested linker-placement, default-placement did indeed produce the same run times as linker-placement.

54

Results from One Compiler

Language	Design	Implementation	Relative Performance
either	*any*	inline	1 (*fastest)*
C++	polystate	non-inline	1.56 x *fastest*
C++	bundled	non-inline	1.65 x *fastest*
C	polystate	non-inline	1.70 x *fastest*
C	bundled	non-inline	1.79 x *fastest*
C++	unbundled	non-inline	1.82 x *fastest*
C	unbundled	non-inline	1.95 x *fastest*

- The other compilers produced remarkably similar results.

55

Analysis

- Using inline functions does more than anything to improve the performance of memory-mapped accesses.
- With each compiler, ***every inline implementation outperformed every non-inline implementation***.
- In fact, with the exception of one slightly unusual case with one compiler (the C++ polystate implementation using reference-placement at global scope), inlining erased every other factor from consideration.
- Among the non-inline implementations, three things stood out…

56

Analysis

- First…
- ***In general, the best non-inline polystate implementations outperform the best non-inline monostate implementations.***
- This is the case for all three compilers.
- Likely reason(s):
  - The ARM architecture uses base+offset addressing.
  - The polystate implementations work well with base+offset addressing.

57

Analysis

- Second…
- ***The non-inline unbundled monostate implementations with linker-placement (or default-placement) have the worst performance of all.***
- Some readers argued that the C unbundled monostate was best, when in fact…
- ***The C implementation is worse than the C++ implementation.***
- This is true for every compiler.
- Why?
  - The unbundled monostate shifts knowledge of the data layout from the compiler to the linker.
  - The shift robs the compiler of information it could use to improve code quality.

58

Analysis

- Third and possibly the most striking…
- ***Except for the non-inline unbundled monostate in C++, every non-inline C++ implementation outperformed every non-inline C implementation.***
- Once again, this was true for every compiler, regardless of its age.
- This surprised me.
- I expected:
  - The C code to be clearly better in the oldest compiler.
  - The gap between C and C++ to disappear over time.
- I didn't expect:
  - The non-inline C++ to be better than the non-inline C almost across the board.

59

Parting Thoughts

- "You can observe a lot just by watching."—Yogi Berra

60

ACCU 2016

Unscientific Computing

Dan Saks
Saks & Associates
www.dansaks.com

61

Acknowledgements

- Thanks to Greg Davis, Bill Gatliff, and Nigel Jones for their valuable feedback.

62

Further Reading

- [Hundt 2011] Hundt, Robert, "Loop Recognition in C++/Java/Go/Scala", *Proceedings of Scala Days 2011.* http://research.google.com/pubs/archive/37122.pdf
- [Saks 2010a] Saks, Dan. "Alternative models for memory-mapped devices", *Embedded Systems Design*, May 2010, p. 9. http://www.embedded.com/224700534
- [Saks 2010b] Saks, Dan. "Memory-mapped devices as C++ classes", *Embedded.com*, June 2010. http://www.embedded.com/4200572
- [Saks 2010c] "Measuring instead of speculating", *Embedded Systems Design*, December 2010, p. 9. http://www.embedded.com/4211118

63

64