Copyright and Permission

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

1

Writing Better Embedded Software in C++

Dan Saks
Saks & Associates
www.dansaks.com

2

Abstract

Flight software, like much embedded software, demands high standards for reliability. The most effective way to keep bugs out of your programs is to not let them in there in the first place. And the best way to do that is to code in a style that turns potential run-time errors into compile-time errors.

Although C is the most popular language for writing embedded systems, C's type system is actually too weak to support this approach.

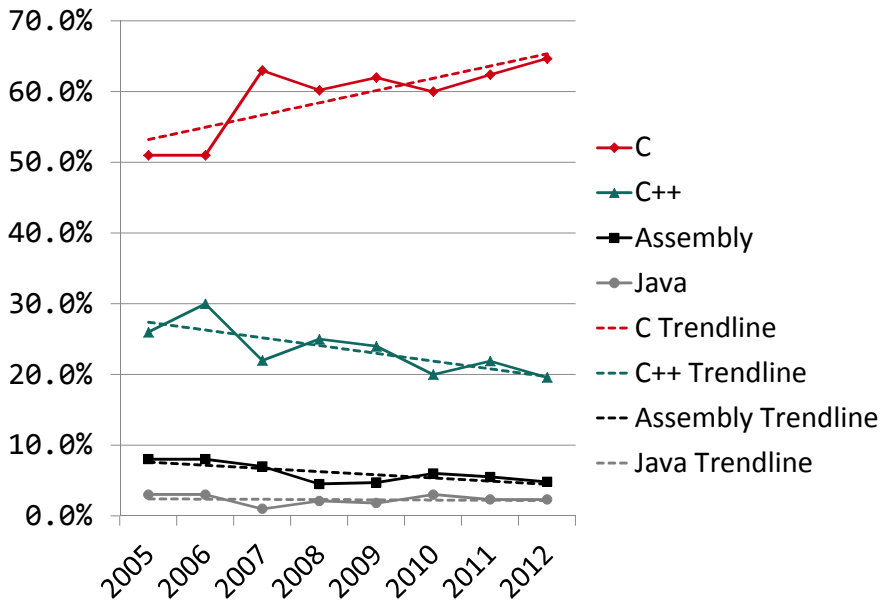
This talk explains how you can use the type facilities in C++ to catch defects at compile time much more effectively than you can in C.

3

Languages for Embedded Programming

- ***embedded.com*** surveys its readers annually.
- Complete this sentence:
- ***"My current embedded project is programmed mostly in…"***

4



5

Overview

- Most embedded software has to be reliable.
- The best way to keep bugs away is to not let them in at all.
- This is a lost cause in C.
- C++ gives you more of a fighting chance.

6

Tools and Thinking Habits

- "The tools we use have a profound (and devious!) influence on our thinking habits…"
  — Edsger W. Dijkstra
- Moving from C to C++ requires a change in mindset.

7

memcpy Copies Arrays

- C's memcpy function can copy one array to another:

```
int ix[10], iy[10];
~~~

memcpy(ix, iy, sizeof(ix));      // copy iy to ix
```

8

memcpy is Very Flexible

- It can copy array's of ***any*** type and ***any*** length:

```
struct widget wx[30], wy[30];
~~~

memcpy(wx, wy, sizeof(wx));      // copy wy to wx
```

- What's not to like?

9

memcpy is Lax

- memcpy can copy incompatible arrays:

```
int ix[10];
~~~

double dy[20];
~~~

memcpy(ix, dy, sizeof(ix));      // copies gibberish
```

- This leaves gibberish in ix.

10

memcpy is Very Lax

- memcpy can copy too much:

```
int ix[10], iy[20];
~~~

double dy[20];
~~~

memcpy(ix, dy, sizeof(dy));     // copies too far
```

- This overflows ix, probably clobbering iy.

11

C's Compile-Time Checking is Weak

- C will compile code with all sorts of bugs.
- More so than most languages.
- This breeds an unfortunate mindset…

12

An All-Too-Common C Mindset

- Just get the code to compile, so you can get to real work…
- …debugging.

13

The Alternative

- ✓ ***Program in a style that turns potential run-time errors into compile-time errors.***
- This is much more viable in C++ than in C.
- It's grounded in some basic language properties…

14

Static Data Types

- C and C++ use ***static data typing***.
- 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's doesn't matter what you try to store into it.

15

Data Types Simplify Programming

- 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)
```

16

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***

17

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 =, +, -, *, /, %, <, >, ==, !=, &, |, &&, ||

18

What's a Data Type?

- An `int` ***can't*** do…

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

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

19

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
```

20

C Structures vs. C++ Classes

- C++ classes have much in common with C structures, but…
- A ***C structure*** is a user-defined type with ***lax constraints*** on the permitted operations.
- A ***C++ class*** is a user-defined type with ***rigorous constraints*** on the permitted operations.

21

Preventing Accidents

- Type information ***helps prevent accidents***:

```
int *p, *q;
double x, y;
~~~
p = q / 4;          // error: can't divide a pointer
x = y & 0xFF;       // error: can't bitwise-and a double
```

22

C++ is Stricter Than C

- In C, enumeration values are just integers.
- In C++, they have distinct types:

```
enum day { Sun, Mon, ~~~, Sat };
enum month { Jan, Feb, ~~~, Dec };
~~~
day d = Feb;        // OK in C; error in C++
month m = Tue;      // OK in C; error in C++
d = 3;              // OK in C; error in C++
~~~
```

23

C++ is Stricter Than C

- C++ is stricter about pointer conversions:

```
int *pi;
double *pd;
~~~
pi = pd;        // OK in C; error in C++
pd = pi;        // OK in C; error in C++
```

- C compilers don't have to issue warnings.
- Fortunately, most do.

24

Casts Are Hazardous

- Casts quell the complaints:

```
pi = (int *)pd;     // compiles quietly; still suspect
pd = (double *)pi;  // same here
```

- Silence doesn't mean assent:

```
int i;
pd = (double *)&i;
*pd = 3;            // compiles OK; undefined behavior
```

25

Use Casts Sparingly

✓ ***Use casts sparingly and cautiously.***

- Casts usually take your code in the wrong direction…
- They turn compile-time warnings and errors into potential run-time bugs.
- Avoiding casts can be hard in C.
- It's often much easier in C++.

26

memcpy Revisited

- memcpy copies arrays one character at a time, as if declared as:

```
void *memcpy(char *dst, char const *src, size_t n);
```

- That's the not real declaration.
- It's too inconvenient…

27

memcpy Revisited

- It requires casting to quell warnings:

```
int ix[10], iy[10];
~~~

memcpy((char *)ix, (char *)iy, sizeof(ix));
```

28

memcpy Revisited

- memcpy actually uses "pointer to void" parameters:

```
void *memcpy(void *dst, void const *src, size_t n);
```

- This avoids casting by sacrificing safety...

29

void * is a Weak Type

- A "pointer to void" is a ***generic data pointer***.
- You can assign any data pointer to it:

```
int *pi;
double *pd;
widget *pw;
void *pv;
~~~
pv = pi;        // OK in C and C++
pv = pd;        // this, too
pv = pw;        // this, too
```

30

void * is Hazardous

- Using "pointer to void" is akin to using a cast:

```
gadget *pg;
widget *pw;
void *pv = pg;       // copy gadget * to void *...
~~~
pw = pv;             // ...and then later to widget *
```

- *Danger!* pw is now a "pointer to widget" pointing to a gadget.

31

Use void * Sparingly

- As with casts…

✓ ***Use "pointer to void" sparingly and cautiously.***

- Very hard in C; much easier in C++.
- Using "pointer to void" takes your programs in the wrong direction…

32

The Most Important Design Guideline?

- "Make interfaces easy to use correctly and hard to use incorrectly."
  — Scott Meyers
- `memcpy` is neither ETUC[1] nor HTUI[2]…

1. Easy To Use Correctly
2. Hard To Use Incorrectly

33

`memcpy` Isn't All That ETUC

- Again, a typical call looks like:

```
memcpy(ix, iy, n);  // n is the size to copy
```

- This would be easier:

```
memcpy(ix, iy);     // easier: compiler supplies size
```

- This would be even easier:

```
ix = iy;            // even easier: familiar assignment
```

34

memcpy is Definitely Not HTUI

- Again, the real problem with memcpy is that it invites errors:

```
int ix[10], iy[10];
~~~

double dy[20];
~~~

memcpy(ix, dy, sizeof(ix));     // fills ix with garbage
memcpy(ix, dy, sizeof(dy));     // copies past ix
```

35

Little to Work With in C

- The C alternative is completely impractical…
- Write a copy function for each type and size of interest:

```
void copy_char_10(char (*dst)[10], char (*src)[10]);
void copy_char_20(char (*dst)[20], char (*src)[20]);

void copy_int_10(int (*dst)[10], int (*src)[10]);
void copy_int_20(int (*dst)[20], int (*src)[20]);
```

- Yikes!

36

A Simple Alternative in C++

- In C++, you can write a ***function template*** that safely copies arrays of only the same type and size:

```
int ix[10], iy[10], iz[20];
double dx[10];
~~~
array_copy(ix, iy);     // OK: same type and size
array_copy(ix, iz);     // error: size mismatch
array_copy(ix, dx);     // error: type mismatch
```

37

A Simple Alternative in C++

- The implementation is remarkably simple:

```
template <typename t, size_t n>
void array_copy(t (&dst)[n], t (&src)[n]) {
    for (size_t i = 0; i < n; ++i)
        dst[i] = src[i];
}
```

- `array_copy` is ETUC and HTUI.

- `array_copy` runs no slower, and usually faster, than `memcpy`.

38

Another Alternative in C++

- What about arrays of different sizes?
- This seems like reasonable behavior:

```
int ix[10], iy[10], iz[20];
~~~
array_copy(ix, iy);     // OK: same size
array_copy(ix, iz);     // error: would overflow
array_copy(iz, ix);     // OK: would fit
```

- Implementing it is easy…

39

Another Alternative in C++

- Allow different-sized arrays, but…
- …make sure the source is no bigger than the destination:

```
template <typename t, size_t m, size_t n>
void array_copy(t (&dst)[m], T (&src)[n]) {
    if (m < n)
        throw "destination too small";
    for (size_t i = 0; i < n; ++i)
        dst[i] = src[i];
}
```

40

Why Wait?

- This is a runtime check:

```
if (m < n)
    throw "destination too small";
```

- But it tests values known at compile time:

```
int ix[10], iy[10], iz[20];
~~~
array_copy(ix, iy);     // m = 10, n = 10
array_copy(ix, iz);     // m = 10, n = 20
array_copy(iz, ix);     // m = 20, n = 10
```

41

Why Wait?

- You can test the condition at compile time:

```
template <typename t, size_t m, size_t n>
void array_copy(t (&dst)[m], t (&src)[n]) {
    static_assert(m >= n, "destination too small");
    for (size_t i = 0; i < n; ++i)
        dst[i] = src[i];
}
```

- This `array_copy` is also ETUC and HTUI.

42

More Alternatives in C++

- C++ offers a standard `vector` class template.
- A `vector` is a variable-length array that knows its size.
- It resizes itself on demand:

```
vector<int> vi = { 8, 6, 7, 5, 3, 0, 9 };
vector<int> wi;     // initially empty
~~~
wi = vi;            // even EerTUC than array_copy
```

43

More Alternatives in C++

- The standard `vector` class uses dynamic memory.
  - It might be verboten in some embedded applications.
- The standard `vector` class isn't HTUI.
- You can write a vector-like class template that:
  - avoids dynamic memory, and
  - is ETUC.

44

Sooner Rather Than Later

- ✓ ***Program in a style that turns potential run-time errors into compile-time errors.***
- Compile-time checks generate no code.
- They use no run time.
- You can ship a program that fails a run-time check.
- You can't ship a program that fails a compile-time check.

45

ETUC + HTUI

- ✓ ***Make interfaces easy to use correctly and hard to use incorrectly.***
- All of this is much easier to do in C++ than it is in C.

46

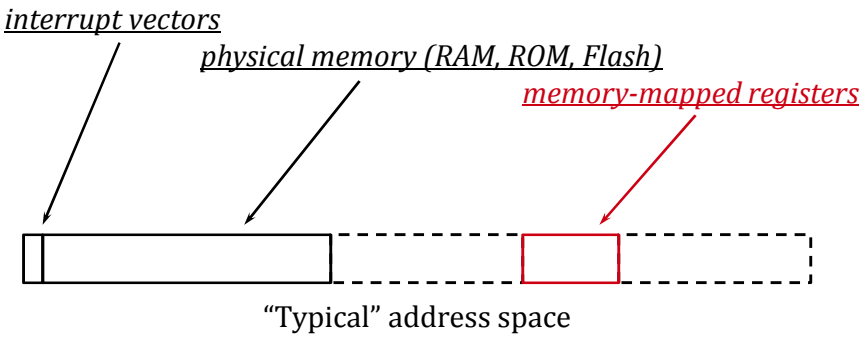
And Now For Something a Bit Different

- Drivers communicate with hardware via ***device registers***.
- Most modern computer architectures use ***memory-mapped addressing***…

47

Memory-Mapped Devices

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



48

Traditional Register Representation

- Hardware vendor libraries often define device register addresses as clusters of related macros.
- The registers often have the same type, such as:
- `typedef uint32_t volatile dev_reg;`
- OK. Maybe there are two or three register types:

```
typedef uint32_t volatile dev_reg32;
typedef uint16_t volatile dev_reg16;
```

49

Traditional Register Representation

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

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

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

50

Free Software You Can't Afford

- These macros cripple compile-time error detection:

```
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
```

51

Using Structures

- Clustering registers into C structures is better:

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

void timer_enable(timer *t);
```

52

Using Classes

- Clustering registers into C++ classes is even better:

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

- A class restricts the permitted operations more rigorously.

53

ETUC

- Using classes simplifies driver interfaces.
- You simply apply member functions to objects representing memory-mapped devices:

```
UART *const com0 = (UART *)0x3FFD000;
~~~

com0->put(c);       // put c to a UART object
```

- You need not know exactly which registers to pass.

54

HTUI

- Each class is a distinct type.
- Type checking can catch accidents such as:

```
UART *const com0 = (UART *)0x3FFD000;
timer *const timer0 = (timer *)0x3FF6000;
~~~

timer0->put(c);      // error: can't put to a timer
com0->put(c);        // OK: can put to a UART
```

55

Preventing Accidents

- Most device registers support both read and write operations.
- Not all UART registers are read/write:
  - USTAT and URXBUF are ***read-only***.
  - UTXBUF is ***write-only***.

56

Preventing Accidents

- Modeling read-only registers is easy—use const:

```
class UART {
    ~~~
private:
    dev_reg ULCON;
    dev_reg UCON;
    dev_reg const USTAT;
    dev_reg UTXBUF;
    dev_reg const URXBUF;
    dev_reg UBRDIV;
};
```

57

Preventing Accidents

- What about write-only registers?
- Use a class template to enforce write-only semantics:

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

58

A Class Template for Write-Only Types

- Just note how short the complete template implementation is:

```
template <typename T>
class write_only {
public:
    write_only() { }
    write_only(T const &v): m (v) { }
    void operator =(T const &v) { m = v; }
    write_only(write_only const &) = deleted;
    write_only &operator =(write_only const &) = deleted;
private:
    T m;
};
```

59

ETUC and HTUI

- Using the `write_only` template turns spurious run-time errors into compile-time errors.
- It costs nothing in code or data space.
- It costs nothing in run time.

60

Sooner Rather Than Later

✓ ***Program in a style that turns potential run-time errors into compile-time errors.***

- Compile-time checks generate no code.
- They use no run time.
- You can ship a program that fails a run-time check.
- You can't ship a program that fails a compile-time check.

61

ETUC + HTUI

✓ ***Make interfaces easy to use correctly and hard to use incorrectly.***

- All of this is much easier to do in C++ than it is in C.

62

Ta Da

63

Copyright and Permission

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

64

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 currently on leave 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.

65

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.

66