Sooner Rather Than Later

Dan Saks
Saks & Associates
www.dansaks.com

 1

Abstract

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. One of the best ways to do that is to code in a style that turns potential run-time errors into compile-time or link-time errors.

This talk explains how you can use the C++'s type system along with other semantic information to turn questionable constructs into code that doesn't build.

 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 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.

 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

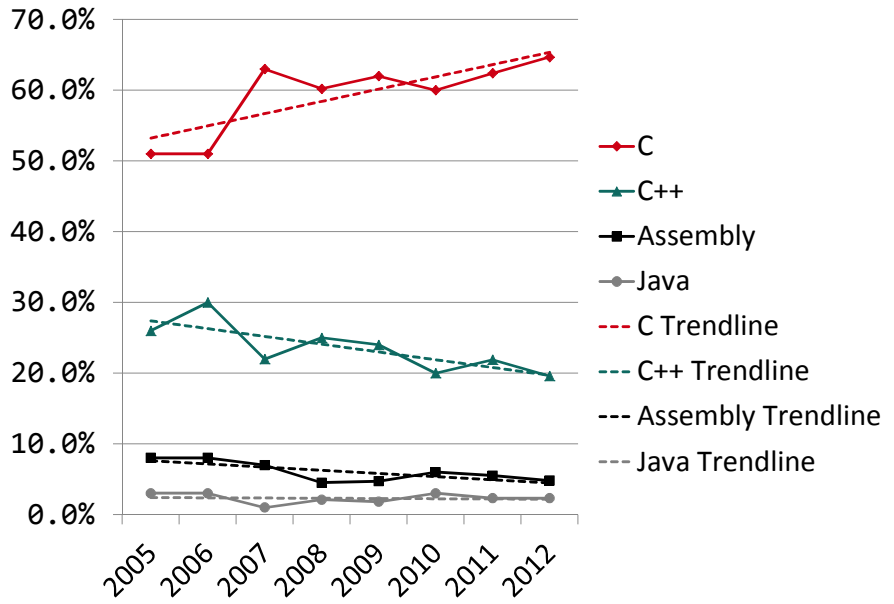
Ohio, USA

 5

Languages for Embedded Programming

- I wrote for:
  - ⇨ ***Embedded Systems Programming***
  - ⇨ ***Embedded Systems Design***
  - ⇨ ***embedded.com***
- Annual reader survey question…
- Complete this sentence: ***"My current embedded project is programmed mostly in…"***

 6



 7

What's Going On Here?

- In the embedded world…
- C's code base is large and growing.
- C++'s market share is shrinking.

 8

What's Going On Here?

- "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.
- The C++ community may be making change harder by setting the bar too high…

 9

A Bar Too High?

- Most students learn the "modern" approach to C++:
  - Use streams instead of FILEs.
  - Use vectors instead of arrays.
  - Use strings instead of null-terminated character sequences.
- C++ was once a "Better C".
- Now, it's a "new language".

 10

An Ironic Situation

- The irony is:
  - A key aspect of C++ that made it popular…
  - …is now often discounted.
- Some C projects stay with C to keep C++ programmers from running amok.
- The "best" may be the enemy of "better".

 11

An Ironic Situation

- Teaching C++ as a new language is appropriate in some situations.
- But one size doesn't fit all.
- At times, moving incrementally from C to C++ may be more practical.

 12

Changing Minds

- 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 very difficult in C…

 13

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
```

 14

memcpy is Flexible

- It can copy arrays of any type and any length:

```
double ix[20], iy[20];
~~~

memcpy(dx, dy, sizeof(dx));     // copy dy to dx
```

 15

memcpy is Very Flexible

- Really, ***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?

 16

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.

 17

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.

 18

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…

 19

An All-Too-Common C Mindset

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

 20

Sooner Rather Than Later

- ✓ ***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…

 21

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.

 22

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)
```

 23

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***

 24

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

 25

What's a Data Type?

- What a type can't do is also important.
- An `int` ***can't*** do…

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

i.m     // member

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

 26

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
```

 27

"Pop Quiz, Hot Shot"

```
int main() {
    int x[10];      // What is the type of x?
    ~~~
}
```

- It's an "array of `10` elements of type `int`."
- `x` is not a pointer, as this is:

```
int *p;             // really a pointer
```

 28

Arrays "Decay" into Pointers

```
int x[10];
int *p;
~~~
p = x;      // Why does this compile?
```

- Arrays "decay" into pointers.
- More precisely, the assignment performs an implicit ***array-to-pointer conversion***:

```
p = x;      // x "decays" to &x[0]
```

 29

Momentary Conversions

- Array "decay" is momentary, just like this conversion from `int` to `double`:

```
double d;
int i;
~~~
d = d + i;
```

```
double temp = i;
d = d + temp;
```

- The temporary vanishes soon thereafter.
- Object `i` remains an `int`.

 30

Array Parameters are Pointers

- An array declaration in a parameter list really declares a pointer.
- These are equivalent:

```
int f(t *x);        // x is a "pointer to t"
int f(t x[N]);      // x is a "pointer to t"
int f(t x[]);       // x is a "pointer to t"
```

 31

But Who Uses Arrays Anymore?

- Again, C++ as a "new language" avoids arrays.
- However, you probably can't avoid them if you work with:
  - other embedded developers
  - on existing embedded software.

 32

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.

 33

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
```

- Compilers use type information to turn potential run-time errors into compile-time errors.
- You can, too.

 34

C++ is Stricter Than C

- C++ is stricter than C 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.

 35

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
```

 36

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++.

 37

Use "New Style" Casts

✓ ***If you must use a cast, use a "new style" cast:***

- `dynamic_cast`
- `const_cast`
- `reinterpret_cast`
- `static_cast`

 38

memcpy Revisited

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

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

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

 39

memcpy Revisited

- It requires casting to quell warnings or errors:

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

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

 40

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...

 41

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
```

 42

void * is Very Hazardous in C

- Using "pointer to void" in C 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 a "pointer to widget" that actually points to a gadget.

 43

void * is a More Overt Hazard in C++

- C++ is more restrictive:

```
gadget *pg;
widget *pw;
void *pv = pg;                    // OK in C and C++
~~~
pw = pv;                          // compiles only in C
pw = (widget *)pv;                // compiles in C and C++
pw = static_cast<widget *>(pv); // compiles only in C++
```

- *Same danger!*

 44

Use `void *` Sparingly

- As with casts…

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

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

 45

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

 46

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
```

 47

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
```

 48

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!

 49

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
```

 50

A Simple Alternative in C++

- The implementation is remarkably simple:

```
template <typename t, size_t n>
inline 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`'s parameters have the form ***`t (&r)[n]`***...

 51

Reference-to-Array Parameters

- An array declaration in a parameter list declares a pointer:

```
int f(T *x);        // x is a "pointer to T"
int f(T x[N]);      // x is a "pointer to T"
int f(T x[]);       // x is a "pointer to T"
```

- This is different:

```
int f(T (&r)[N]);   // x is a "reference to array of T"
```

- The array dimension is part of the parameter type.

 52

Keeping Up with memcpy

- If, for some reason, memcpy is faster than array_copy, you can do this:

```
template <typename t, size_t n>
inline void array_copy(t (&dst)[n], t (&src)[n]) {
    memcpy(dst, src, n);
}
```

- array_copy can copy arrays of any type, but…
- memcpy works only with arrays whose elements are trivially assignable…

 53

Restricting What You Copy

- You can use a type trait to restrict the copied type:

```
template <typename t, size_t n>
inline void array_copy(t (&dst)[n], t (&src)[n]) {
    static_assert(
        is_trivially_copy_assignable<t>::value,
        "element type has non-trivial copy assignment"
    );
    memcpy(dst, src, n);
}
```

- is_pod is a more restrictive alternative.

 54

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...

 55

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>
inline 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];
}
```

 56

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
```

 57

Sooner Rather Than Later

- You can test the condition at compile time:

```
template <typename t, size_t m, size_t n>
inline 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.

 58

More Alternatives in C++

- Modern C++ offers a standard `array` class template.
- An `array<t, n>` is an "array with `n` elements of type `t`" with a vector-like interface:

```
array<int, 10> ax;  // default initialized
array<int, 10> ay = { 8, 6, 7, 5, 3, 0, 9 };
array<int, 20> az;  // default initialized
~~~
ax = ay;            // even EerTUC than array_copy
az = ax;            // compile error: size mismatch
```

 59

More Alternatives in C++

- Using the standard `vector` is often a laudable aspiration.
- However, the standard `vector` class uses dynamic memory.
- It might be verboten in some embedded applications.

 60

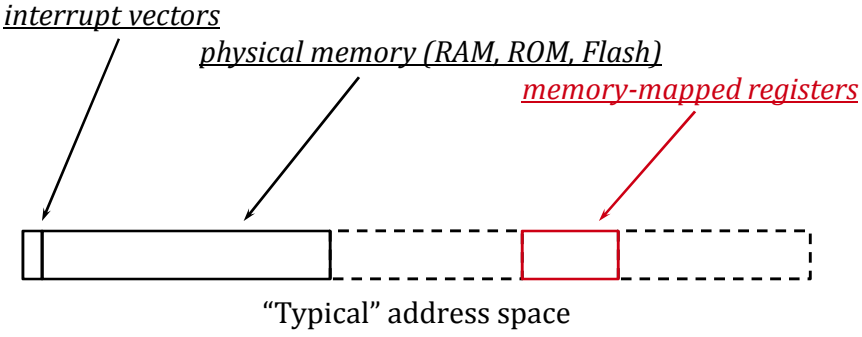
And Now For Something a Bit Different

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

 61

Memory-Mapped Devices

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



 62

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;
```

 63

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)
~~~
```

 64

Accessing Device Registers

- You can use these macros to access the registers:

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

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

 65

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
```

 66

Using Structures

- Clustering registers into C structures is better:

```
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 concerns about storage layout in my other lecture.

 67

Using Structures

- This, too:

```
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);
```

 68

ETUC

- Using structures simplifies driver interfaces.
- You pass all the registers for a device as a unit:

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

 69

Somewhat HTUI

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

```
UART *const com0 = (UART *)0x3FFD000;
timer *const timer0 = (timer *)0x3FF6000;
~~~
UART_put(timer0, c);    // error: can't put to a timer
UART_put(com0, c);      // OK: can put to a UART
```

- A structure leaves the registers exposed to improper accesses…

 70

Using Classes

- Using a class with private members cuts down on improper registers accesses:

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

 71

Enforcing Restricted Access

- 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***.

 72

Enforcing Read-Only

- const provides read-only semantics:

```
class UART {
    ~~~
private:
    dev_reg ULCON;
    dev_reg UCON;
    dev_reg const USTAT;        // read only!
    dev_reg UTXBUF;             // write only?
    dev_reg const URXBUF;       // read only!
    dev_reg UBRDIV;
};
```

 73

Enforcing Write-Only

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

```
class UART {
    ~~~
private:
    ~~~
    dev_reg const USTAT;            // read-only!
    write_only<dev_reg> UTXBUF;     // write-only!
    dev_reg const URXBUF;           // read-only!
    dev_reg UBRDIV;
};
```

 74

Enforcing Write-Only

- A write-only object should support a very limited set of operations.
- It prohibits all read operations:

```
write_only<int> m = 0;  // initialization
write_only<int> n;      // default initialization
n = 42;                 // assignment
m = n;                  // error: attempts to read from n
```

 75

A Write-Only Class Template

- The complete template implementation is very simple:

```
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;
};
```

 76

Rethinking Read-Only

- Using `const` to declare read-only registers is actually a problem.
- In C++, each const class member must be initialized:
  - via an initializer in the member declaration, or
  - via a member initializer in each class constructor.

 77

Rethinking Read-Only

- Member initialization involves a write operation:

```
class UART {
public:
    UART(): URXBUF (0) { }
    ~~~
private:
    ~~~
    dev_reg const USTAT = 0;     // not really read-only
    write_only<dev_reg> UTXBUF; // write-only!
    dev_reg const URXBUF;        // not really read-only!
    ~~~
};
```

 78

A Read-Only Class Template

- We need a read-only type without mandated initialization.
- Using a `read_only<t>` class template looks like:

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

 79

A Read-Only Class Template

- `read_only<t>` provides a ***default constructor***, plus…
  - A ***conversion operator*** that lets you read the object's value:

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

  - An ***operator &*** that yields the address as a "pointer to const":

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

 80

A Read-Only Class Template

- Not surprisingly, the template implementation is simple:

```
template <typename t>
class read_only {
public:
    read_only() { }
    operator t const &() const { return m; }
    t const *operator &() const {  return &m; }
    read_only(read_only const &) = deleted;
    read_only &operator =(read_only const &) = deleted;
private:
    t m;
};
```

 81

A Read-Only Class Template

- Using the `read_only` and `write_only` templates turns spurious run-time errors into compile-time errors.
- They cost nothing in code or data space.
- They cost nothing in run time.

 82

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.

 83

ETUC and HTUI

- ✓ ***Make interfaces easy to use correctly and hard to use incorrectly.***
- Use parameter and return types that restrict values and operations as narrowly as possible.
- Avoid casts and "pointer to void".

 84