Hello!

- As a former German speaker, I am very grateful for the invitation to travel here to Berlin to take part in Meeting C++ 2019.
- Ladies and gentlemen, I am very pleased to be able to address you today as the closing keynote speaker.
- Alas, I’m a bit out of practice at speaking German, so the remainder of this talk will be held in English.
- Thank you again.

Let’s keep this in mind

- “A computer is a stupid machine with the ability to do incredibly smart things, …
- “…while computer programmers are smart people with the ability to do incredibly stupid things.”
  — Bill Bryson, 1998

Let’s also keep this in mind

- “Sometimes we discover unpleasant truths.
- “Whenever we do so, we are in difficulties:
  - suppressing them is scientifically dishonest, so we must tell them,
  - “but telling them, however, will fire back on us.
- “[W]e will be written off as
  - “totally unrealistic,
  - “dangerously revolutionary,
  - “hopelessly idealistic,
  - “foolishly gullible …
  - “(Besides that, telling such truths … is not without personal risks. Vide Galileo Galilei…… [sic])”
  — Edgar W. Dijkstra, How do we tell truths that might hurt?, 1975

How to become expert?

- “The answer is the same in all the fields I’ve seen:
  1. “Learn the basics.
  2. “Study the same material again but this time, concentrate on the details you didn’t realize were important the first time around.”
And there's always more to learn

- "Learning is cumulative."
- "It's revisionist."
- "It's iterative."
- "It's incremental. ..."
- "[So] your knowledge always increases.
- "[But] your appreciation of what you don't know increases [as well]."

— Kevlin Henney, What Do You Mean?, 2019

How to start?

- "One of the best ways to learn is the study of examples."
- "It is useful to examine both "good style to be emulated as well as "poor practice to be avoided."
- "The skillful critique of imperfect art — critical analysis — is a powerful technique to improve the quality of one's own work."

— Marc F. Paterno, Defective C++, 2003

In other words …

- "Study others' code."
- "Learn from past successes."
- "Learn even more from past failures."

— Howard E. Hinnant, Design Rationale for <chrono>, 2019

So I will show many small examples

- I chose most of these C++ excerpts because ...
  - I've seen them firsthand in production code ...
  - (or they have reliably been reported to me) ... often enough to annoy/irk/peeve/irritate/provoke me.
- I will discuss a few of them in considerable detail, but we'll simply look (and shake our heads) at others.
- To avoid embarrassments and legal entanglements:
  - I kept the essence of each code snippet, ...
  - But did sanitize (reformat/recode/restyle) each one, ...
  - While laundering (disguising) identifiers.

— Kevlin Henney, What Do You Mean?, 2019

I also have for you today a mix of ...

- Examples of unfortunate outcomes:
  - Some silly, some humorous, some just wrong, but also some truly horrific.
  - Selected advice from other experts:
    - Some quite recent, but also some rather older.
  - Some cultural influences that can make our jobs hard.
  - Some new, useful C++17 and C++20 features.
  - And a few bits of fun along the way.
  - (Coincidentally, some of my topics overlap talks by other speakers; do view those for additional depth.)

As I've often said, please be forewarned

- Based on my training and extensive experience, I do hold some rather strong opinions about computer software and programming methodology.
- I know that all these opinions are not yet shared by all programmers.
- But they should be! 😊
You, too, will discover this

- "[W]hen you need to pay tuition and a mortgage, you are willing to put up with a certain amount of stupidity so that you can take care of your family.
- "Once those bills are paid, your tolerance for idiocy shrinks quite a bit."

— Pseudonymous Blogger, *It’s No Big Deal*, 2017

Binary water? For programmers?

Makes sense to me

(apologies to Tolkien)

Once upon a time …

- A C++17 programmer walked into a local pub and ordered 1.000000119F beers.
- The proprietor said, "I’ll have to charge you extra; that’s a root beer float."
- "In that case," replied the programmer, "make it a double."

And now …

“‘The time has come,’ the Walrus said, ‘to speak of many things:’”

Through the Looking-Glass and What Alice Found There, 1872
Unprofessional Results
That Shouldn’t Have Reached Production

A number of years ago, we bought a GPS

• We don’t use “Home.”
• Instead, we entered our home address in the gen’l directory under my name: “Brown, W E.”
• Here’s what the GPS speaks as we arrive home: “Arriving at Brown, West East, on right.”

Looking for work? Maybe not here …

When you want 8 English-only ports?

I’d likely lose this on my desk

Seems a bit much to remember
So what should the user do now?

Error in error-handling. Sorry, no handling possible.

Another example

- “A software glitch forced 12 ... stealth fighters to ... turn back.... The problem seems to have arisen ... from the change in longitude from W179.99 degrees to E180 which occurs on the International Date Line.”
  — News report, 2007

- “At the international date line, whoops, all systems [failed, including] their navigation, part of their communications, [and] their fuel systems.”
  — Maj. Gen. Don Sheppard (ret.)

Our embarrassments seem unending

- “Today, researchers ... are detailing ... vulnerabilities in a popular operating system that runs on more than 2 billion [sic] devices worldwide....
- “VxWorks is [a] real-time operating system for ... devices, like medical equipment, elevator controllers, or satellite modems. ...
- “Roughly 200 million devices appear to be vulnerable; the bugs have been present [since] 2006. [T]he patching process will be long and difficult ....”
Crazy Code, Crazy Coders

- Please take note of the Forum on Risks to the Public in Computers and Related Systems:
  - Inaugurated in 1985, moderated by Peter G. Neumann on behalf of the ACM Committee on Computers and Public Policy.
  - Current and all previous digests are online at http://catless.ncl.ac.uk/risks.
  - Note also a monthly column, Inside Risks, in CACM:
    - “Edited and distilled highlights from the columns appear bimonthly in ACM SIGSOFT Software Engineering Notes.”
    - Recent and “important” columns are online at http://www.csl.sri.com/users/neumann/insiderisks.html.

- FYI

  • Please take note of the Forum on Risks to the Public in Computers and Related Systems:
    - Inaugurated in 1985, moderated by Peter G. Neumann on behalf of the ACM Committee on Computers and Public Policy.
    - Current and all previous digests are online at http://catless.ncl.ac.uk/risks.
    - Note also a monthly column, Inside Risks, in CACM:
      - “Edited and distilled highlights from the columns appear bimonthly in ACM SIGSOFT Software Engineering Notes.”
      - Recent and “important” columns are online at http://www.csl.sri.com/users/neumann/insiderisks.html.

- Anything wrong here?

  • void calculate() {
      double x; d = 1.0; // zero-init thusly, or copy-init via = 0.0
      // Or maybe copy-init via = std::nan("")
    }

  • Uninitialized local variables are usually problematic:
    - Especially when of native (e.g., arithmetic) type — no ‘ctors!
    - And even more problematic when of floating-point type.
  
  • Why is uninitialized floating point so dire?
    - Because the uninitialized residue in the above variable d is a bit pattern that could denote any floating point value, ...
    - Even one of the bit patterns denoting a signalling NaN!
      (When did your team last code with an sNaN in mind?)

- The “let me change my mind” idiom

  • if (k > max) k = max; // … and correct the guess if necessary
  
  • Why not simply set k’s value correctly ab initio, thus allowing for const-correctness?

- Scope rules matter, too

  • int x = 42;
    void act() {
      float x = x; // wait, what?
    }

  • Each name has a point of declaration after which that name is in scope (decl is findable via appropriate lookup):
    - “The point of declaration for a name is immediately after its complete declarator...” (See [basic.scope.decl][1].)
    - So “the second x is initialized with its own (indeterminate) value” — whose bit pattern might be that of an sNaN!

Examples:

```cpp
#include <iostream>
#include <cassert>
using namespace std;

int x = 42;

int main() {
  x = 0;
  x = x; // wait, what?
  return 0;
}
```

- Let’s Rethink Some Common C++ Coding Practices

  “Practice yourself ... in little things, and thence proceed to greater.”

  — Epictetus, ca. 55-135

---

[1]: https://www.open-std.org/jtc1/sc22/wg21/docs/papers/2021/p2749r0.html
Idiomatic counted loops

- void eval() {
  auto b[N] = {...};
  for(int k = 0; k < N; ++k) {
    b[k] = 0;
  }
}

Why use post-increment here?
- It's previous value is unused, so why copy and return it?
- Instead, avoid an implicit copy by coding such loops with a pre-increment operator: ++k.
- “But the compiler will optimize it for me.”
- Maybe, but post-increment here misstates the intent.
- It’s a poor coding habit that hinders comprehension.

Idiomatic counted loops

- void eval() {
  auto b[N] = {...};
  for(int k = 0; k < N; ++k) {
    b[k] = 0;
  }
}

Why use operator < here?
- As written, this loop’s exit condition is k >= N
  (i.e., “we’ve performed at least N iterations”).
- But the likely intended exit condition is k == N
  (i.e., “we’ve performed exactly N iterations”!!)
- Instead, code as k <= N in the predicate of such loops.

A range-based for is no panacea

- template<class T>
  void refill(std::vector<T> & vec, T const & value) {
    for(auto & e : vec) {
      e = value;
    }
  }

This won’t always compile; did you spot the bug?

Hint: when T is bool, e’s type can’t/mustn’t be bool &&:
- Iteration will involve a temporary (of a proxy type for the bit in question), which can bind to a const ref type only.
- Should declare auto && e to infer the correct type!
- (But try teaching this to inexperienced programmers.)
Analysis of range-based for ②  
(adapted from N3853)

• for(auto const & e : source) works in limited cases:
  • Does observe elements in situ, even for most proxies.
  • But obviously can’t mutate const elements in-place.
• Explicit (non-deduced) element types may be worse:
  • for(string e : source) still copies elements.
  • for(string & e : source) fails for const/value elements.
• for(Elem const & e : source) can be “actively harmful” when the type, Elem, is even slightly wrong:
  • E.g., for a source of type std::map<K, V>, an Elem type std::pair<K, V> will convert-copy each element …
  • Because the element type is actually std::pair<K const, V>.

Analysis of range-based for ③

• Further, keep in mind that code such as this:
  • std::vector<std::string> make_strings(…); // factory fctn
  • for(char c : make_strings(…)[0]) c = ‘c’;
  • … by definition behaves as if the loop were written:
    • auto & & s = make_strings(…)[0]; // ref to 0th string
      for(auto & b = s.begin(); e = s.end();)
      ; b != e;
      ++b ;
  • But s refers to the leading (0th) member of a temporary vector, now gone (nothing extended the vector’s lifetime).
  • Thus s is a dangling reference, unusable in the loop!

Analysis of range-based for ④  
(C++20)

• Will be able to rewrite this incorrect code:
  • std::vector<std::string> make_strings(…); // factory fctn
    for(auto & & c : make_strings(…)[0]) c = ‘c’;
  • ...to use the new optional init-statement feature:
    • for(auto & v = make_strings(…); auto & & c : v[0])
      c = ‘c’;
  • This now extends the returned vector<string>’s lifetime to the end of the loop’s body.
  • (Could of course instead copy the vector if so desired.)

Out of sight, often out of mind?  
(adapted from D. S. Hollman)

• template< typename Container, typename UnaryPredicate >
  void replicate_if(Container & cont, Predicate & pred)
  { for(auto const & e : cont)
    if(pred(e))
      cont.push_back(e); // append a copy of this element
  }

  • Would the programmer have invalidated the iterator had it been not concealed by the range-based for?

Assume the declaration bool it_works(…);

1. if(it_works(…)) return true;
2. if(it_works(…)) return false;
3. else if(it_works(…)) return true;
4. else if(it_works(…)) return false;
5. return true: false;

"[The Analytical Engine] might act upon other things besides number..."  
— Ada Lovelace, 1815–1852
Crazy Code, Crazy Coders

But I've seen these variations, too

• if (it works(⋯) ? true : false) { return true; }
  return false;
• if (bool b = it works(⋯)) return b; else return false;
• switch (it works(⋯)) {
  case true:
    return true;
  break;
  case false:
    return false;
  break;
  default:
  }

More bool-related foolishness

• if (to be or not to be) {
  // same code here
} else {
  // exact same code here!
}
• bool isValid() { return true; }
• template <class Value> Value return value (Value value) {
  return value (false);
}

And still more bool illogic

• while (true) { bool flag = true; } // really want a true flag
• bool isAlive() {
  try {
    return true;
  } catch (myException ex) { return false; }
}
• for (⋯) {
  if (condition1(⋯)) break;
  if (condition2(⋯)) break;
  // code goes here
  break; // always break out of the loop at the end
}

And, when bool isn't good enough, ...

• enum logical { tautology, contradiction, http 404 not found };
• enum maybe so { always so, usually so, often so, sometimes so, rarely so, never so };

Summing is simple, right?

Let's Rethink Arithmetic

“I wish to God these calculations had been executed by steam.”
— Charles Babbage, 1791–1871

Assuming 3-digit decimal mantissas, let's add:

- 1.00 + .001 + .999 = (1.00 + .999) + .001 → 1.99
- But in reverse order: (.001 + .999) + 1.00 → 2.00

So, whenever possible, sum the smallest values first:

- Why? To reduce risk of loss of significance.
- double reordered_sum(this from, double* from, double* upto) {
  std::sort(from, upto);
  return std::accumulate(from, upto, 0);
}

- Oops: The above starting value, 0, should be double(0)
- Oops 2: What if some (or all) values are negative?
No, summing is often not so simple

- auto less_in_magnitude = [] (double x, double y)
  ( return std::abs(x) < std::abs(y); )
- auto reordered sum= double* from, double* upto )
  std::sort(from, upto, less_in_magnitude);
  return std::accumulate(from, upto, double{});

- This approach works for integral values, too, reducing
  (but not eliminating) risk of overflow.
- In general, we worry about summing values that:
  - Have very large or very small values, and/or ...
  - Have mixed signs.

For floating point

- template< floating_point F >
  F midpoint(F a, F b) { // see Hauser’s theorem 3.4.1a,
  // cited in F. Gouraud: “How do you
  // compute the midpoint of an
  return can_safely_sum(a, b)
    ? (a + b) / 2 :
      safe_half(a) + safe_half(b);
  }

- See also:
  - G. E. Forsythe: “Pitfalls in computation, or why a math

How about finding the midpoint (e.g., in binary search)?

- Possible approaches to implement midpoint(a, b):
  - return (a + b) / 2; ?
  - return a / 2 + b / 2; ?
  - return a + (b - a) / 2; // for uint and ptr types when a, b
  - Btw, consider also the more general algorithm:
    - Linear interpolation, a.k.a., lerp, a + t * (b - a).
    - (midpoint and lerps are planned for the C++20 std library.)

- Some common issues (here, and for most arithmetic):
  - Integer overflow, excessive truncation.
  - Floating point underflow, denormals, qNaNs, sNaNs, infs,
    signed zeroes, rounding (mode, frequency).

The helpers

- template< floating_point F >
  bool can_safely_sum(F x, F y) noexcept {
    constexpr F upper = numeric_limits<F>::max() / F(2);
    return abs(x) <= upper
      and abs(y) <= upper; }

- template< floating_point F >
  bool can_safely_halve(F x) noexcept {
    return F(2) * numeric_limits<F>::min() <= abs(x); }

- template< floating_point F >
  F safe_halve(F x) noexcept {
    return can_safely_halve(x) ? x / F(2) : x; }

In fact, no computer arithmetic is simple — it’s finite!

- Subtracting two nearly equal floating values may lead
  to loss of significance via catastrophic cancellation:
  - sqrt(x+1) - sqrt(x) // inaccurate when x > 1
  - 1 / sqrt(x+1) + sqrt(x) // equivalent, yet accurate for all x

- Multiplying even medium-valued ints risks overflow.

- Some summation algorithms that compensate for finite
  arithmetic’s pitfalls (e.g., Kahan-Neumaier, pairwise, ...).

- Older programs mimicked ~48-bit integers via long
  double, but C++11 gave us long long, typically 64 bits.

- Unless they’re such glorified ints, essentially never
  compare floating-point values for exact equality.

Is this really how best to ensure a non-positive n?

- if( n > 0 ) n = n - 1;
- if( n > 0 ) n = 0 - n;
- n = n > 0 ? n + 1 : n;
- n = -1 + abs(n);
- if( n > 0 ) n -= 2 + n;
- if( n > 0 || n < 0 ) {
  string str = "-"s + to_string(n);
  str = regex_replace( str, "-\", "-" );
  // same time later
  n = stoi( str );
}
How about division …

- ... with negative ints?
  - $-15 \div 2 \rightarrow ?$
  - $-15 \% 2 \rightarrow ?$

- By the way, what’s the proper name of operator %?
  - “The binary / operator yields the quotient, and the binary % operator yields the remainder from the division of the first expression by the second.
  - ‘... if the quotient $a/b$ is representable in the type of the result, $(a/b) \times b + a\%b$ is equal to $a$; otherwise, the behavior of both $a/b$ and $a\%b$ is undefined.’ [See (expr.mul)].

Wrong or right?

- How to decide whether $n$ is odd?
  - bool is_odd( int n ) { return n % 2 == 1; }
  - bool is_odd( int n ) { return n & 0b1 == 0b1; }
  - bool is_odd( int n ) {
    if( n == 2'147'483'647 ) return true;
    else if( n == 0 ) return false;
    return is_odd( n += 2 );
  }
  - inline bool is_even( int n ) { return n % 2 == 0; }
  - inline bool is_odd( int n ) { return not is_even(n); }

Even simple arithmetic can be challenging

- $59 + 1 \rightarrow 0$ every minute, unless there happens to be a leap second, then $59 + 1 \rightarrow 60$ and $60 + 1 \rightarrow 0$.

- Consider tennis scoring:
  - LOVE $+ 1 \rightarrow 15 + 1 \rightarrow 30 + 1 \rightarrow 40 + 1 \rightarrow$ GAME.

- In music, going up by a third and then up by a fifth has the net effect of going up by a seventh!

- Increment your house number; is it your neighbor’s?

Consider arithmetic in pop culture

- “In the arithmetic of love, one plus one equals everything, and two minus one equals nothing.” — Mignon McLaughlin

- “Arithmetic is being able to count up to twenty without taking off your shoes.” — Mickey Mouse

- “Computer, compute to the last digit the value of pi.” — Mr. Spock

Toilet paper arithmetic???

- 12 MEGA PLUS = 54 REGULAR
- 30 DOUBLE PLUS = 68 REGULAR
- 12 SUPER MEGA = 72 REGULAR
- 18 PLUS = 82 REGULAR
Published misinformation (myth-information?) doesn’t help us

- Claim: “every operator has a precedence — a specified order in which the expressions are evaluated”:
  - This popular impression has never been true.
  - Precedence certainly influences order of evaluation, but (except in trivial cases) does not determine it.
  - It also doesn’t help that that book is in its 7th edition!
- So what is the correct role of precedence?
  - To determine which operands bind to which operators ...
  - Using left/right associativity to break any ties.
  - This enables a compiler to build the expression’s tree.

Expression trees: for analysis and for synthesis

<table>
<thead>
<tr>
<th>+</th>
<th>*</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>3</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>+</th>
<th>*</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>4</td>
</tr>
</tbody>
</table>

2 + 3 ≠ 4
(2 + 3) ≠ 4

Building the tree is one algorithm ...

- But expression evaluation uses a different algorithm:
  - To evaluate an expression is to traverse (systematically walk) the corresponding expression tree, ...
  - Along the way applying each operator to its operands (evaluated expression subtrees) ...
  - Resulting in both value computations and side effects.
- C++17 changed the order of evaluation for binary op’s:
  - Previously specified at sequence points only, now ...
    1. In assignment op’s, the right operand’s (subtree’s) evaluation is sequenced before that of the left.
    2. In all other binary op’s, the left operand’s (subtree’s) evaluation is sequenced before that of the right.

Some Thoughts about Our Craft

On clarity and craftsmanship

- “[I]n general terms it’s up to the artist programmer to use language that can be understood, not hide it in some private code ...
- “[O]bscurity is usually the refuge of incompetence.”
  — Robert A. Heinlein, Stranger in a Strange Land, 1961

On semantics

“The difficulty of literature software is not to write code, but to write what you mean; not to affect your reader computer, but to affect him precisely as you wish.”
  — Robert Louis Stevenson, The Truth of Intercourse, 1879
On correctness

- “As soon as we started programming, we found to our surprise that it wasn’t as easy to get programs right as we had thought.
- “Debugging had to be discovered.
- “I can remember ... when I realized that a large part of my life from then on was going to be spent in finding mistakes in my own programs.”

— Maurice V. Wilkes, 1949

On clever coding

- “Debugging is twice as hard as writing the code in the first place.
- “Therefore, if you write the code as cleverly as possible, you are, by definition, not smart enough to debug it.”


On compilers

“Even though the compiler is supposed to be helpful, it also treats you like an adult.”


On “non-local reasoning”

“The farther away [that] I need to look for an answer, the longer it takes to comprehend code.”

— Matthew Fleming, *The Smart Pointers I Wish I Had*, 2019

On reading code

- “[...] and then there are those beautiful, snowflake-like cases of abuse,
- “those moments where you see the code,
- “you understand the code,
- “and you wish that, somehow, you could throttle the invisible person responsible for that code.”


On harsh code reviews

- “Your code is 100% bogus and should be taken out the back, lined up against a wall, and machine-gunned.
- “Then the bleeding corpse should be hung, drawn and quartered.
- “Then burnt.
- “Then the smouldering rubble should be jumped up and down on.
- “By a hippo.”

— Dave Korn, 2005
On maintainability

“Always code as if the [programmer] who ends up maintaining your code will be a violent psychopath who knows where you live.”


On life in general

“Life would be so much easier if we could just look at the source code.”

— Tom Parker (?)

How Bad Can Code Get?

Let’s abuse … almost everything

• Here’s some shockingly inane code, provided by a Highly Paid Consultant:

```cpp
std::string capitalize(std::string s) {
    std::string result;
    if (s.empty()) return "";
    std::for_each(s.begin(), s.end(), std::toupper);
    return result;
}
```

• This was clearly untested; how many mistakes and misunderstandings can you find?

More thoughtless code fragments

• `return (tot == 1 ? 1 : tot);`
• `if (not ok(⋯) std::runtime_error("oops!");`
• `bool not_provided(string const & str) {
    if (str.length() > 0) return true;
    return false;
}
• `while (busy) ; // wait until other thread says not busy`
• `do something(⋯, bool busy, ⋅) {
    while (busy) ( /* wait until it’s not busy */ }

And still more code abuse

• `#define SUCCESS 1
#define FAILURE 2`
• `bool badly_named_fcn(⋯) {
    return (succeeded(⋯) ? SUCCESS : FAILURE);
}
• `my_type* get_me() { return this; }
• `// bogosort (bogus sort; O(n•n) expected swaps)
    while( not std::is_sorted(b, e) )
        std::shuffle( b, e, std::default_random_engine );`

A tale of debugging hell

- I can’t show this code, so please imagine:
  - A copy assignment operator, whose body, ...
  - Through macro magic, has 2 implementations.
  - In debug mode, it performs a traditional deep copy.
  - But in production, to save time, it does a shallow copy:
    - So changing the original (off in another thread) also changes the copy, whereas …
    - In debug mode, they are independent objects, each with its own lifetime, so the bug (a race condition) never manifests!

How not to close a bug report

"I didn’t understand the diagnostic, but this fixed the problem:
#define EXTERN static"


/*
   *-----------------------------------------------------------------
   * When I wrote this code,
   * only God and I knew what’s going on.
   *-----------------------------------------------------------------
   * Now only God knows.
   *-----------------------------------------------------------------
*/

And So …

We have responsibilities

- “Programming is a profession.
- “It is an ethical obligation to work to improve our profession.
- “The more senior and talented you are, the more you owe to the community.
- “… Part of that obligation is to continue to study, to read papers and work through books.”
  — Sean Parent,
  "Modern" C++ Rumination, 2018

“The difference between amateurs and professionals” (excerpted)

- Amateurs:
  - have a goal.
  - think they are good at everything.
  - see feedback and coaching as a blow to their ego.
  - think knowledge is power.
  - blame others.
- Professionals:
  - have a process.
  - understand their circles of competence.
  - know they have weak spots and seek out thoughtful criticism.
  - pass on wisdom and advice.
  - accept responsibility.
Do I have a problem?

• I admit that I do not like all programmers equally.
• While I am biased, it’s not due to issues of ethnicity, religion, politics, etc.
• And it’s certainly not because some programmers use:
  • Ada, • APL, • C, • Cobol, • Fortran, • Go, • Haskell, • Java,
  • Javascript, • Lisp, • PL/I, • Python, • Ruby, • Rust, • Swift,
  or any other not-C++ programming language du jour.

No problem, just a programming zealot!

• I dislike professionally incompetent programmers:
  • Those who demonstrate inadequacy at our craft, ...
  • AND who refuse to learn to do better!
• Not only do such coders make our jobs more difficult:
  • But they don’t really care that their lack of skill and
    of good judgment causes others to suffer, ...
  • So long as they are paid.
• Who are those programmers?
  • Just read their code.
  • You will recognize them.