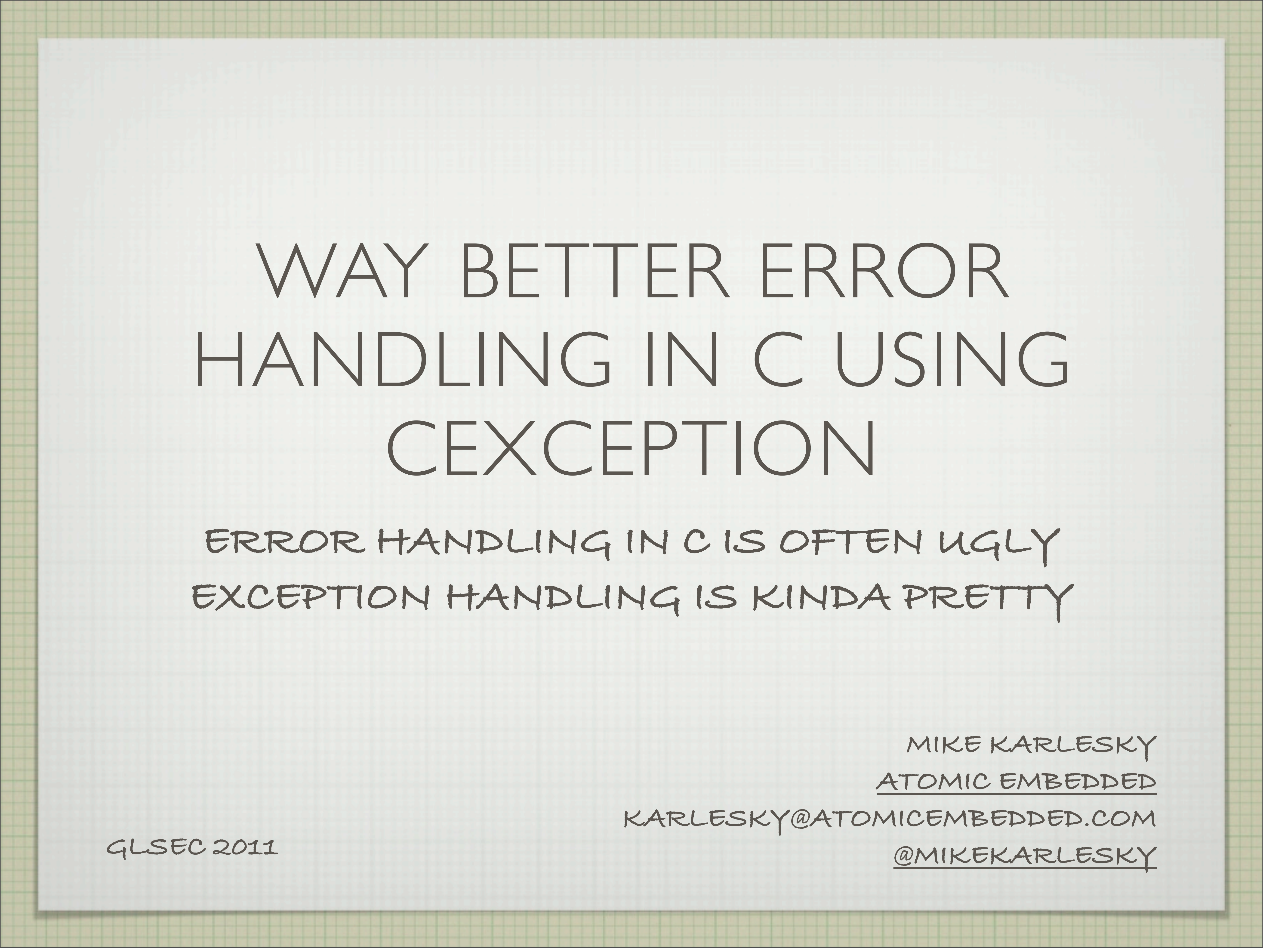


WAY BETTER ERROR HANDLING IN C USING CEXCEPTION

ERROR HANDLING IN C IS OFTEN UGLY
EXCEPTION HANDLING IS KINDA PRETTY

MIKE KARLESKY
ATOMIC EMBEDDED
KARLESKY@ATOMICEMBEDDED.COM
@MIKEKARLESKY

GLSEC 2011

* Using “error” and “exception” in a loose, semi-interchangeable way for purposes of this talk.
* Concepts apply to any programming language, concentrating here on C specifically.

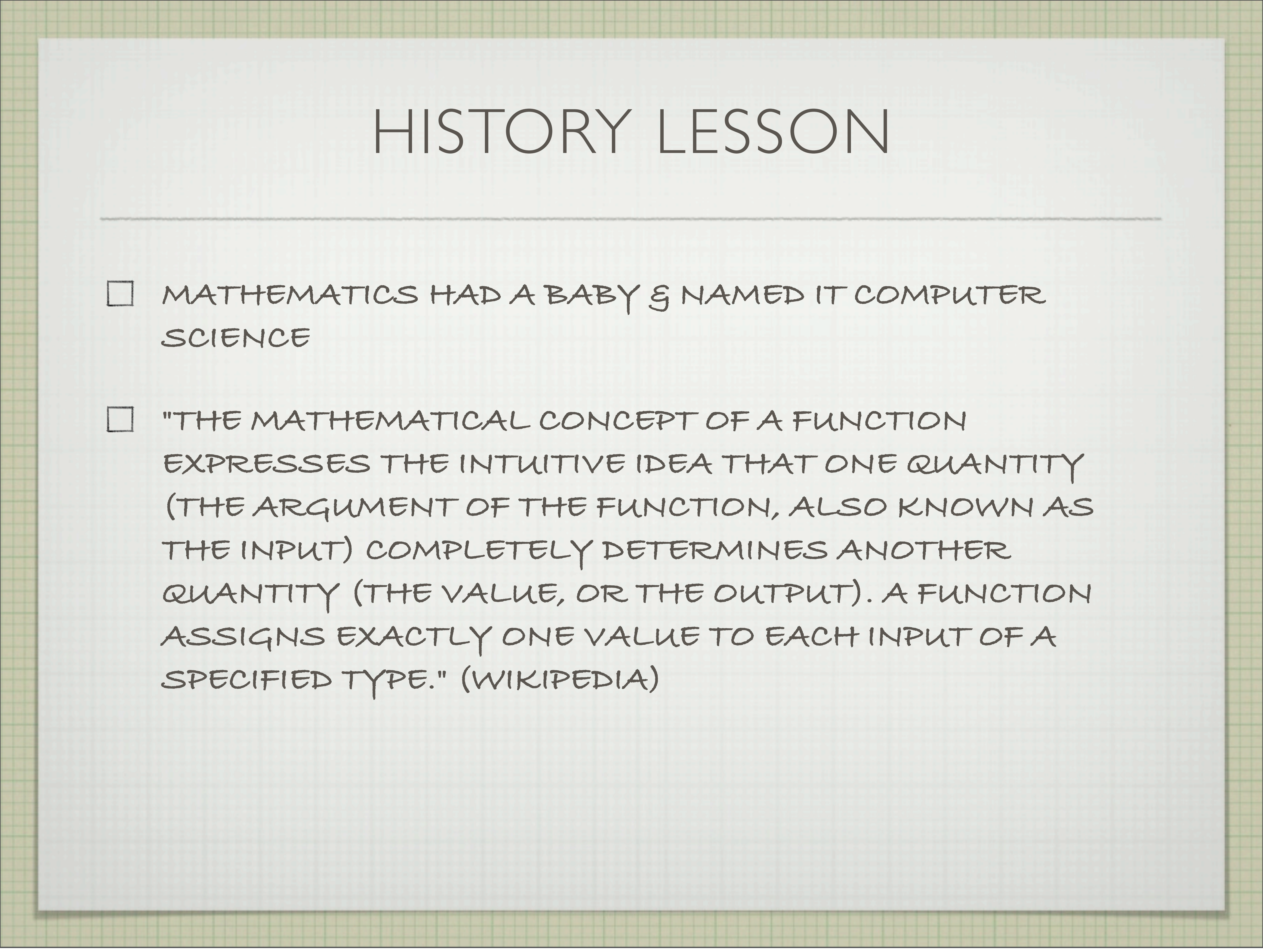
If you’re using C regularly, you’re likely doing low level system software or embedded systems work. This talk is especially for you.

http://atomicembedded.com
http://twitter.com/mikekarlesky

AND AWAY WE GO

- WE WILL COVER:
    - SOME HISTORY & A LITTLE MATH
    - UGLY ERROR HANDLING IS UGLY
    - CONCEPT OF EXCEPTIONS & BENEFITS OF EXCEPTION HANDLING
    - ADDING EXCEPTION HANDLING TO C PROJECTS WITH CEXCEPTION
    - CODE EXAMPLES
- WE WILL NOT COVER:
    - IN-DEPTH MECHANICS OF CEXCEPTION (UNDER THE HOOD)

* There's plenty of great documentation available with CException. It's also only a single source and two header files (one of which is optional) — very easy to digest.



HISTORY LESSON

- MATHEMATICS HAD A BABY & NAMED IT COMPUTER SCIENCE
- "THE MATHEMATICAL CONCEPT OF A FUNCTION EXPRESSES THE INTUITIVE IDEA THAT ONE QUANTITY (THE ARGUMENT OF THE FUNCTION, ALSO KNOWN AS THE INPUT) COMPLETELY DETERMINES ANOTHER QUANTITY (THE VALUE, OR THE OUTPUT). A FUNCTION ASSIGNS EXACTLY ONE VALUE TO EACH INPUT OF A SPECIFIED TYPE." (WIKIPEDIA)

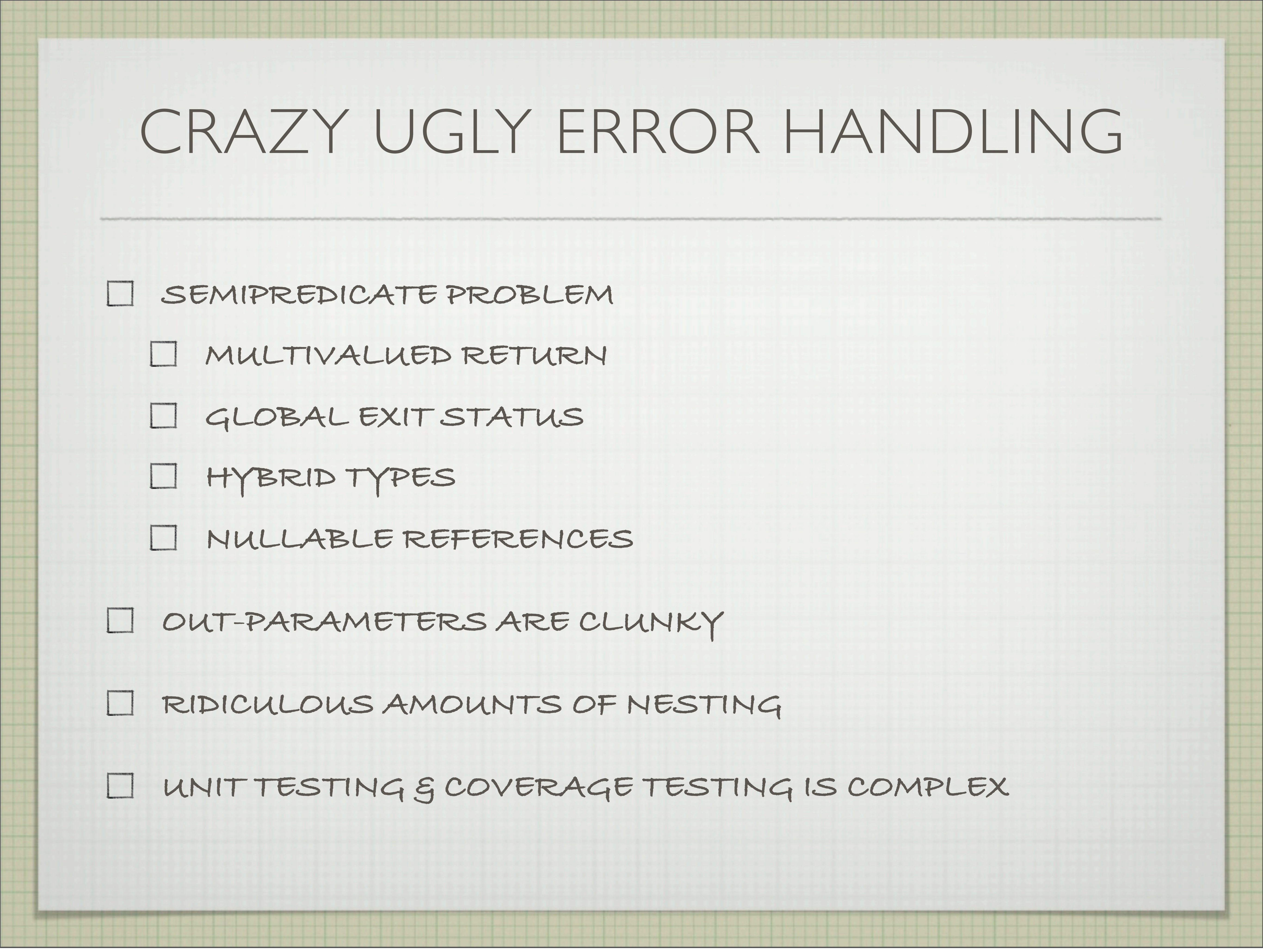
In pure mathematics, functions are great for dealing with the abstract concepts of numbers and transformations thereof. However, in the physical world of processors, sensors, and limited resources, functions are not capable of cleanly accommodating error conditions that prevent input to output transformations.

BIG PROBLEM

SO WHAT TO DO WITH ERROR CONDITIONS THAT PREVENT THE MAPPING OF C FUNCTIONS' INPUTS TO OUTPUTS?

Examples:
- Your heap runs out of a memory as you allocate a data structure within a function
- A function's input parameters create a divide-by-zero condition in a later division operation
- A sensor is damaged and produces nonsense values

Low-level and embedded systems are particularly susceptible to dealing with all manner of specialized and even unforeseen error cases (i.e. at times difficult to even plan for them all).



CRAZY UGLY ERROR HANDLING

- ☐ SEMIPREDICATE PROBLEM
  - ☐ MULTIVALUED RETURN
  - ☐ GLOBAL EXIT STATUS
  - ☐ HYBRID TYPES
  - ☐ NULLABLE REFERENCES
- ☐ OUT-PARAMETERS ARE CLUNKY
- ☐ RIDICULOUS AMOUNTS OF NESTING
- ☐ UNIT TESTING & COVERAGE TESTING IS COMPLEX

“A semipredicate problem occurs when a subroutine intended to return a useful value can fail, but the signaling of failure uses an otherwise valid return value. The problem is that the caller of the subroutine cannot tell what the result means in this case.” (Wikipedia)

Multivalued returns: A data structure of some sort containing both a success/fail status as well as the meaningful output of the function itself.

Global exit status: Just like it sounds. Probably a bad idea — even more so in multi-process / multi-threaded systems.

Hybrid types: A data type wider than range of values to be returned by function and/or embodying some ad hoc scheme of flags, masks, etc to segregate success/failure information from meaningful function output. This becomes especially painful to deal with when passing codes up through multiple layers of software / function calls.

Nullable references: If everything is a pointer, NULL can signify error cases. However, we all know pointers are delicate things to work with responsibly; an entire system of pointers is a bad idea. Further, a NULL pointer is often itself a useful value; overloading its meaning leads to complication. Further still, only a single error value is not particularly useful in distinguishing what to do with error cases.

Out-parameters (filling a “return” value by reference passing) are just ugly and violate the simplicity of the construct of a function. Out-parameters usually reveal poor design and/or the very issue we’re addressing in this talk.

Any of the preceding solutions to the semipredicate problem cause complex code that’s difficult to understand, maintain, and test.

LOVELY EXCEPTION HANDLING

- EXCEPTION: ERROR CASE THAT INTERRUPTS/PREVENTS NORMAL EXECUTION
- ORTHOGONAL OR OUT-OF-BAND TO FUNCTIONS
- "RAISING" OR "THROWING" AN EXCEPTION IS LIKE TELEPORTING OUT OF FUNCTION SCOPE

Errors that muck up clean execution of our functions are... exceptional cases.

A function can "raise" or "throw" an exception. Conceptually, the exception leaves the scope of the function (processing stops) and can then be caught and handled elsewhere in software.

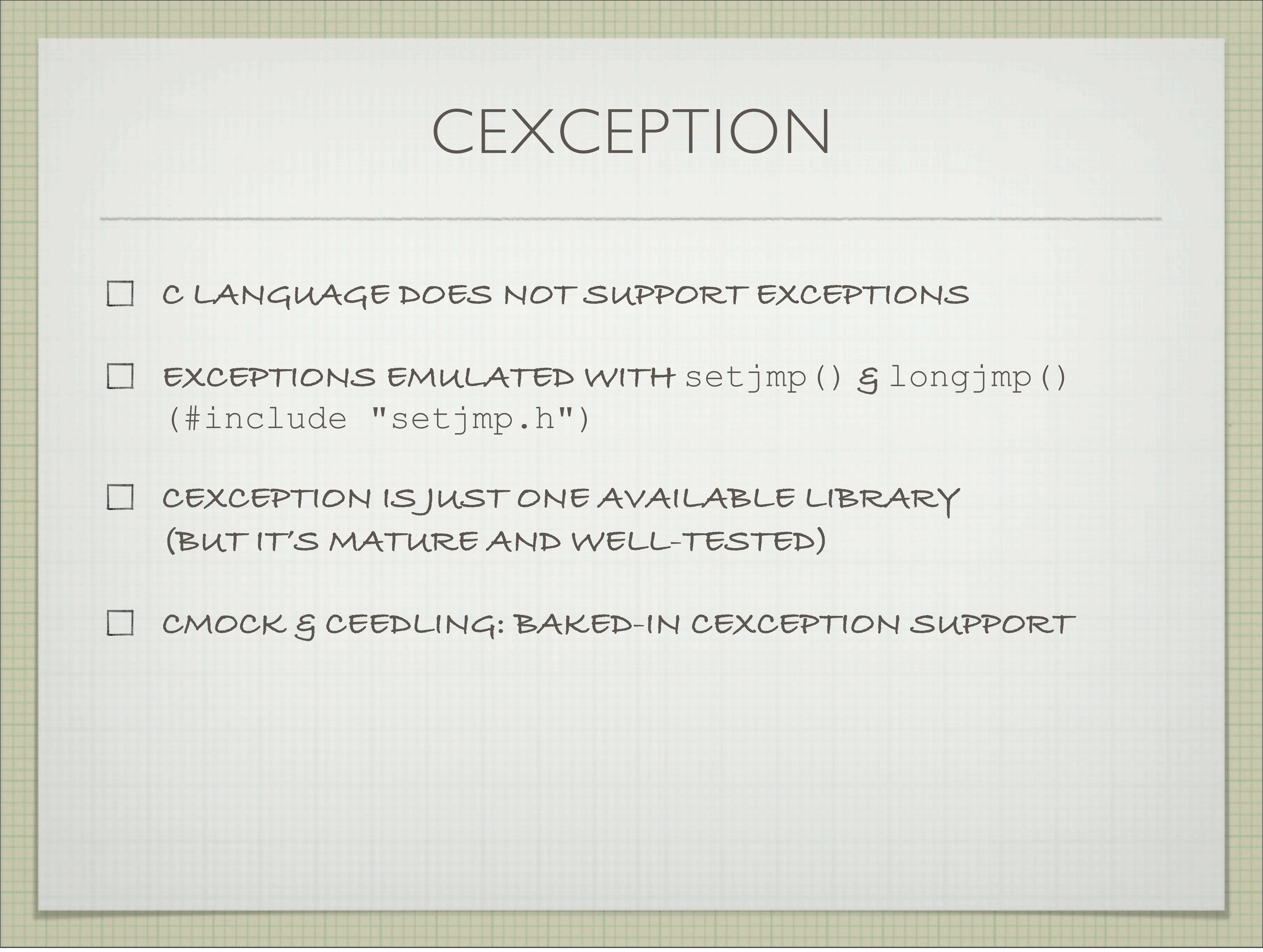
In general, an exception is handled (resolved) by saving the current state of execution in a predefined place and switching the execution to a specific subroutine known as an exception handler. Depending on the situation, the handler may later resume the execution at the original location using the saved information. (Wikipedia)

BENEFITS OF EXCEPTION HANDLING

- REDUCES ERROR HANDLING CONDITIONAL LOGIC & NESTING
- ALLOWS RETURN VALUES TO BE RETURN VALUES
- STREAMLINES & CENTRALIZES ERROR HANDLING

With exception handling, only one “master” handler is necessary for any specific exception no matter how many times it’s thrown in your code. Additionally, there is no need to check for a particular error in each place it may occur all along a call trace path. Throw an exception in one place. Catch it in one place. All production code in-between can be oblivious to exceptional error cases.

Catching and rethrowing allows some flexibility in smartly handling retries and error logging / reporting.



CEXCEPTION

- C LANGUAGE DOES NOT SUPPORT EXCEPTIONS
- EXCEPTIONS EMULATED WITH `setjmp()` & `longjmp()` (`#include "setjmp.h"`)
- CEXCEPTION IS JUST ONE AVAILABLE LIBRARY (BUT IT'S MATURE AND WELL-TESTED)
- CMOCK & CEEDLING: BAKED-IN CEXCEPTION SUPPORT

C++, Ruby, Java, other modern languages support exceptions. C does not.

`setjmp()` and `longjmp()` are standard C library calls that provide non-local jumps: flow control outside usual subroutines & return sequence paradigm. `setjmp()` saves current execution environment in a platform-specific data structure frame on the stack. `longjmp()` restores program state to that frame stored by `setjmp()`.

CException wraps up `setjmp()` and `longjmp()` calls with a little bit of code and some macros so that a "Try/Catch" (i.e. exception handler) saves execution state to which a "Throw" jumps back to upon exceptional error cases.

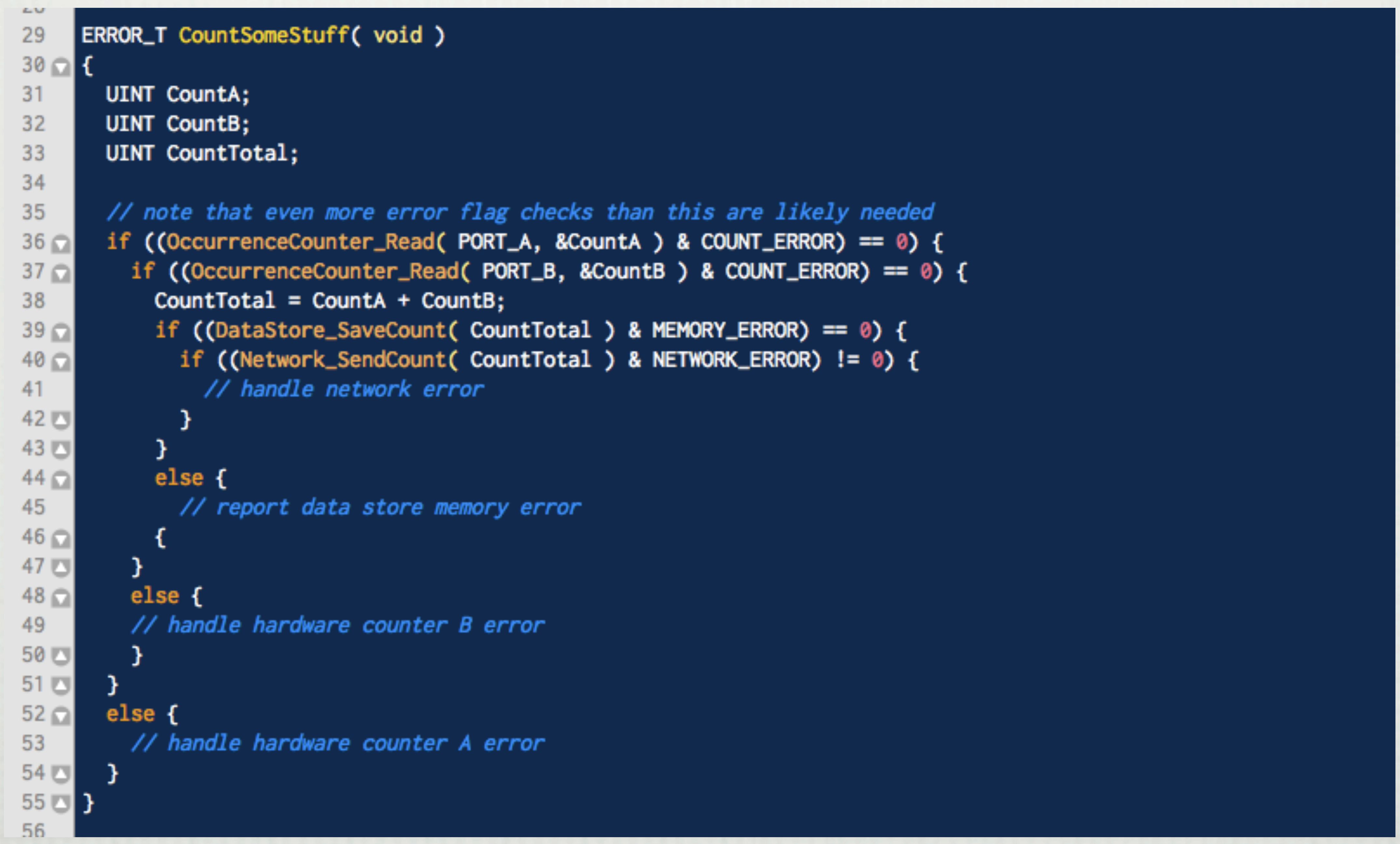
CException works "out of the box" (i.e. compiles) with minor requirements. All you need to do is define a unique list / enumeration of exception IDs for use in your project and compile / link in CException. With just a touch more configuration, it can also work with multi-threaded or multi-tasking systems: connect up routine to supply unique ID per thread or task (i.e. per stack).

No explicit support for "finally" block at present though you can fake it with a bit of conditional code to catch an exception, execute "final" code, and rethrow exception.

CException only uses primitives for exception IDs. To keep things lean, it does not attempt to work with custom structures to report more detail like higher level languages usually do. The primitive used is configurable (to save memory if needed).

For those who've adopted unit testing, particularly interaction-based unit testing, the mocking tool CMock supports CException handling directly. In addition, the test & release build environment tool Ceedling supports CException (and CMock) as well.

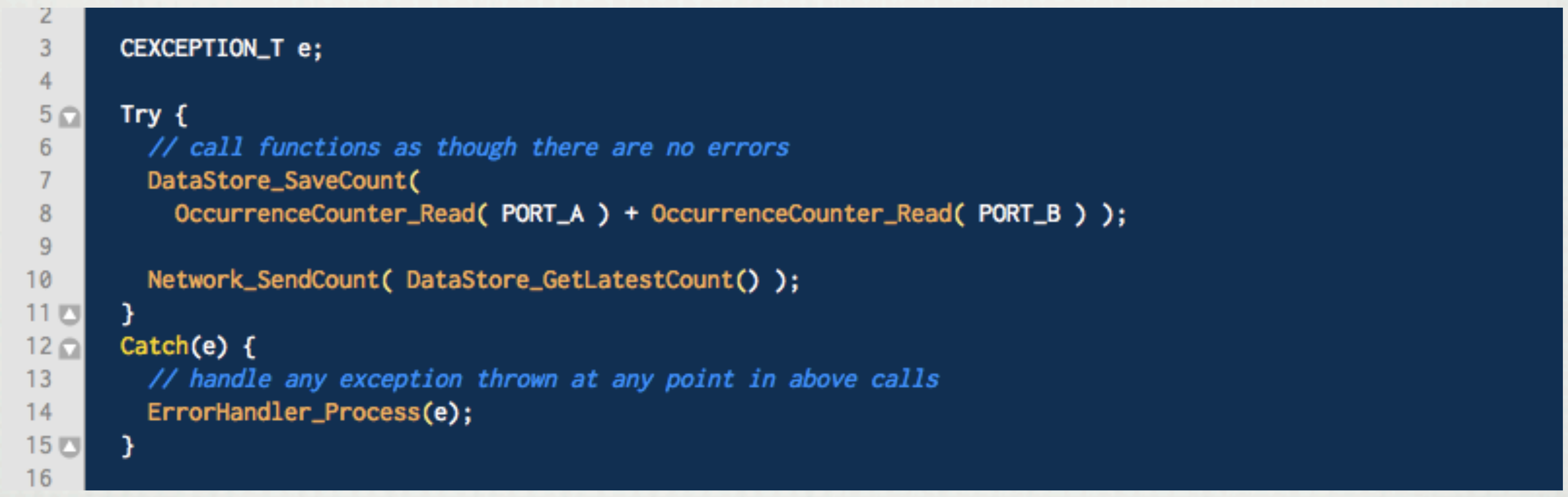
CONTRIVED EXAMPLE: UGLY ERROR HANDLING



```
ERROR_T CountSomeStuff( void )
{
  UINT CountA;
  UINT CountB;
  UINT CountTotal;

  // note that even more error flag checks than this are likely needed
  if ((OccurrenceCounter_Read( PORT_A, &CountA ) & COUNT_ERROR) == 0) {
    if ((OccurrenceCounter_Read( PORT_B, &CountB ) & COUNT_ERROR) == 0) {
      CountTotal = CountA + CountB;
      if ((DataStore_SaveCount( CountTotal ) & MEMORY_ERROR) == 0) {
        if ((Network_SendCount( CountTotal ) & NETWORK_ERROR) != 0) {
          // handle network error
        }
      }
      else {
        // report data store memory error
      {
    }
    else {
    // handle hardware counter B error
    }
  }
  else {
    // handle hardware counter A error
  }
}
```

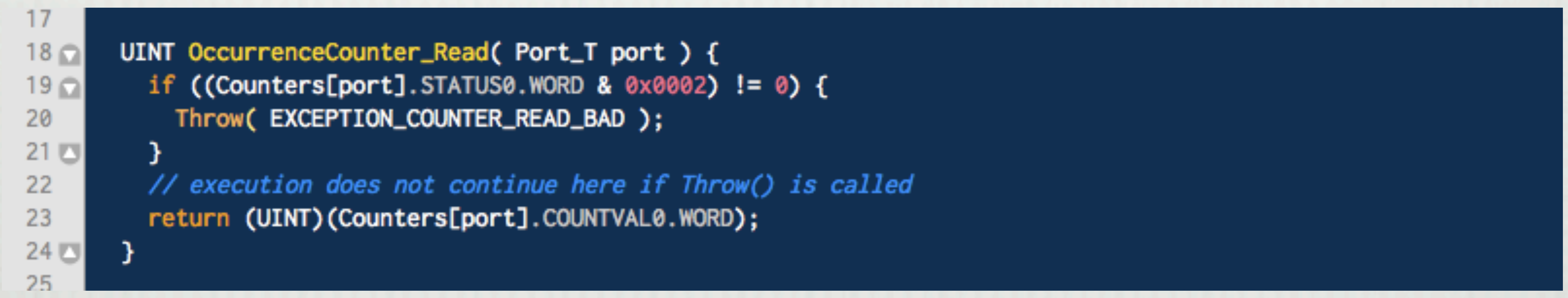
CONTRIVED EXAMPLE: LOVELY EXCEPTION HANDLING



```
CEXCEPTION_T e;

Try {
  // call functions as though there are no errors
  DataStore_SaveCount(
    OccurrenceCounter_Read( PORT_A ) + OccurrenceCounter_Read( PORT_B ) );

  Network_SendCount( DataStore_GetLatestCount() );
}
Catch(e) {
  // handle any exception thrown at any point in above calls
  ErrorHandler_Process(e);
}
```



```
UINT OccurrenceCounter_Read( Port_T port ) {
  if ((Counters[port].STATUS0.WORD & 0x0002) != 0) {
    Throw( EXCEPTION_COUNTER_READ_BAD );
  }
  // execution does not continue here if Throw() is called
  return (UINT)(Counters[port].COUNTVAL0.WORD);
}
```

CEXCEPTION.SOURCEFORGE.NET

THROWTHESWITCH.ORG

CException is offered under the MIT License. You should have no problem incorporating it into your personal or commercial development project.

http://cexception.sourceforge.net
http://throwtheswitch.org