Adventures in Updating a ~~Legacy~~ Vintage Codebase

Billy Baker

billy.baker@flightsafety.com

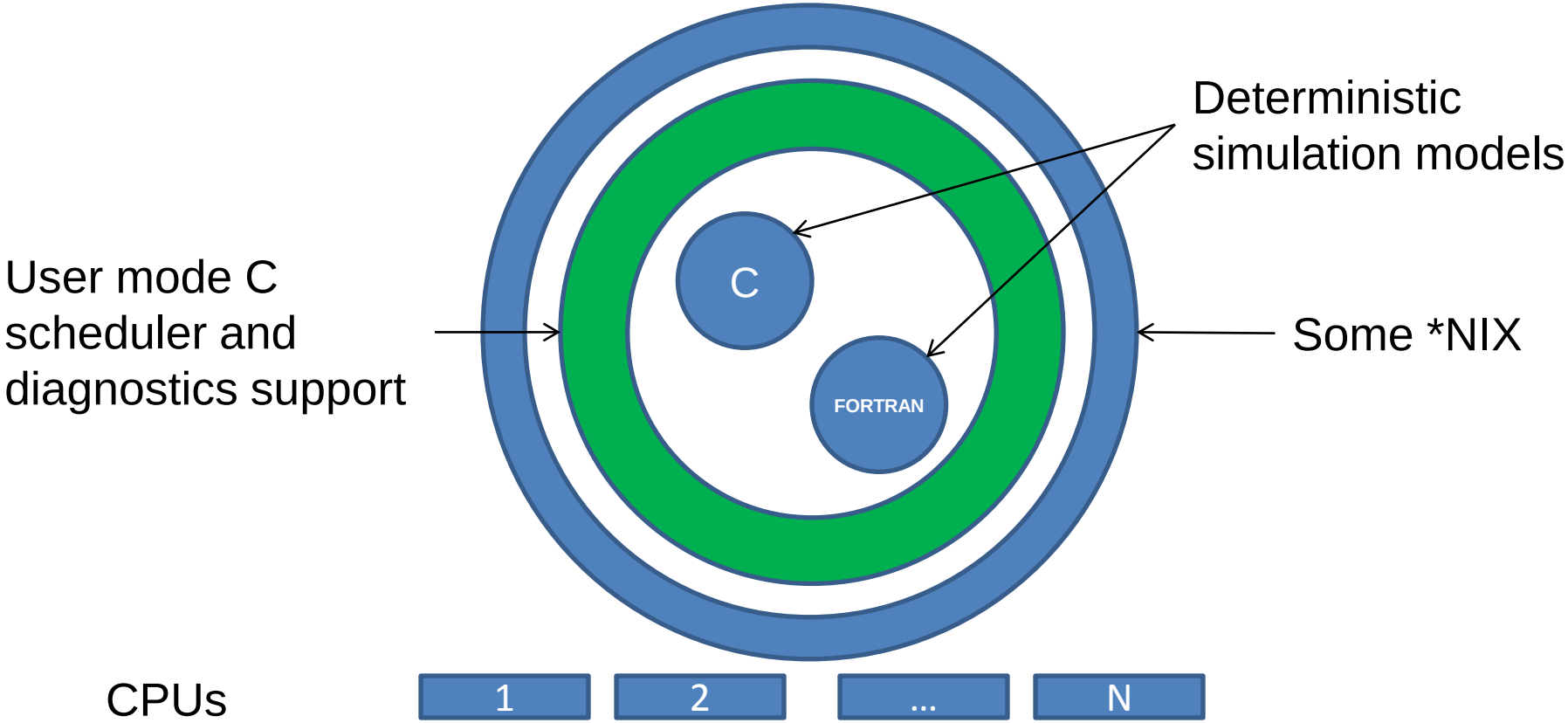
FlightSafety International Simulation Systems

Photos courtesy of Dan Littmann

VINTAGE

- Flight simulation must recreate diverse environments
  - Ada, C, C++, Fortran, Jovial, PLM, Pascal
  - PowerPC, Motorola 68k, AMD 29050, 16/32bit x86, 1750A
- The simulation must either stimulate, simulate or emulate real instruments

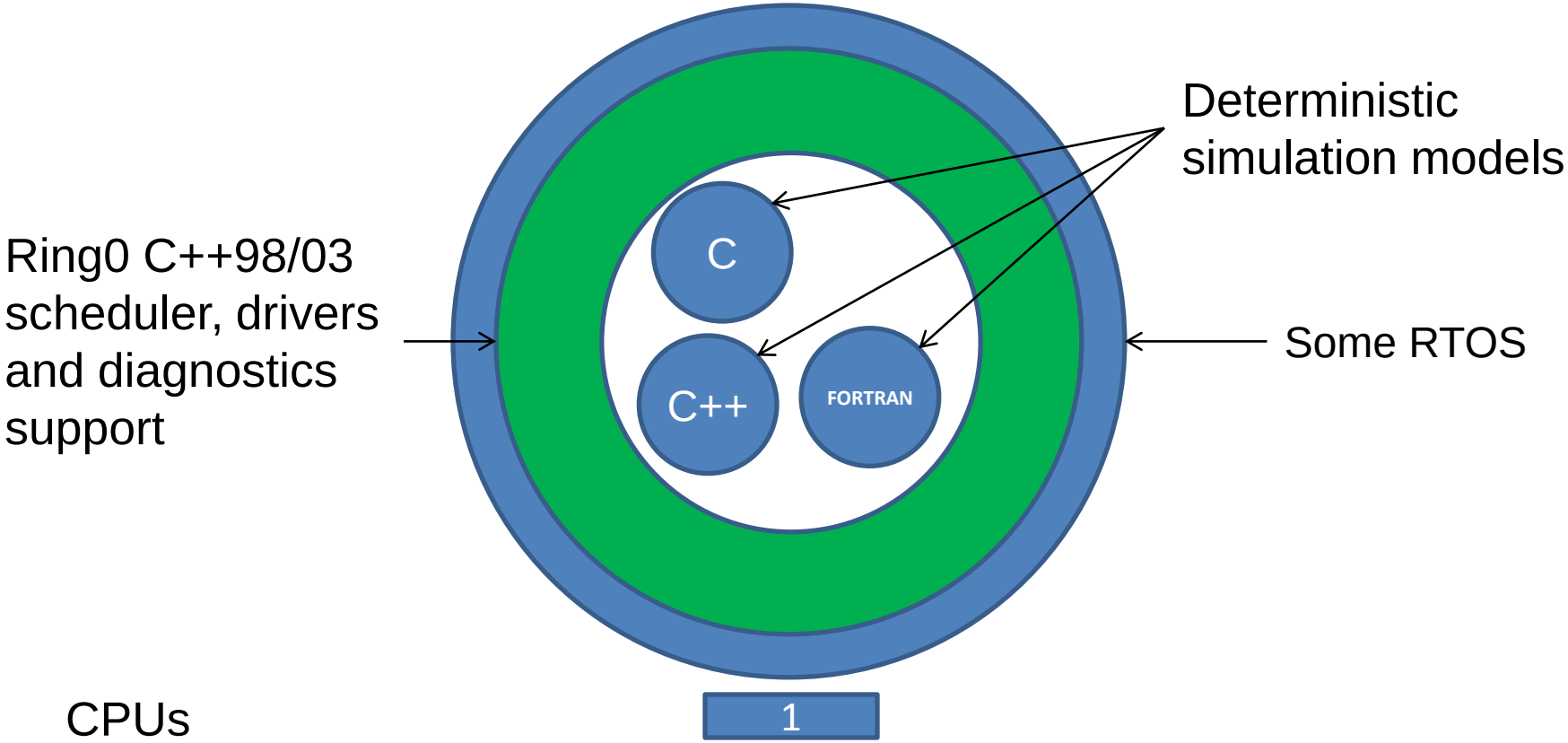
SIMULATION OF OLD



SIMULATION OF YESTERDAY

- CPUs became fast enough that only one CPU was needed
- Simulation framework switched from C to C++
- A deterministic real-time flight simulation could run on a desktop/laptop OS by linking with different libraries

SIMULATION OF YESTERDAY



EXAMPLE: BIGGER PROBLEMS

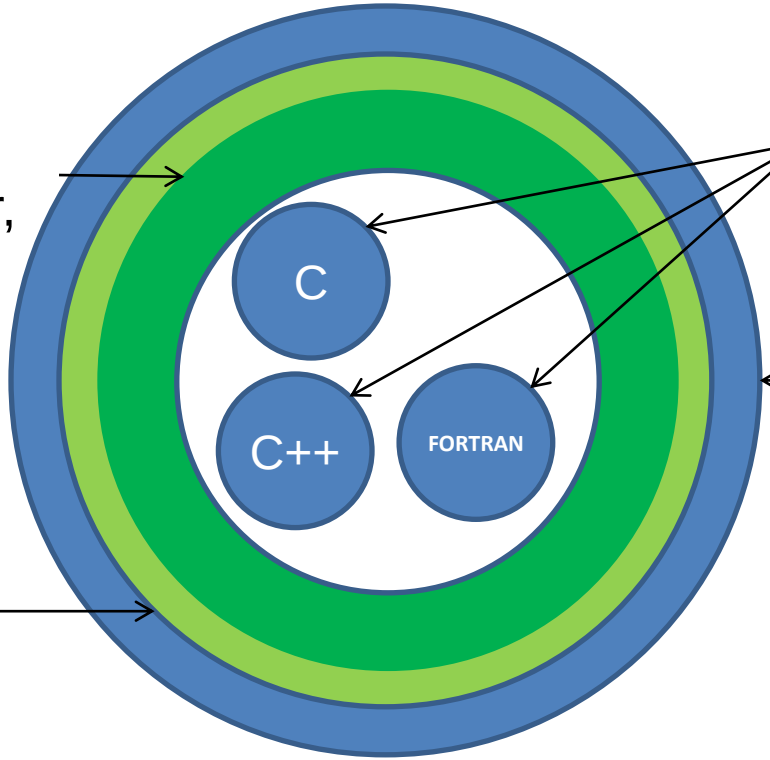
- Integrate binaries that run on
  - 6 little endian, 16bit CISC CPUs with 20bit segmented memory addressing and 60Hz scheduling
  - 4 big endian, 32bit CISC CPUs with 50Hz scheduling
  - Backplane shared memory
  - Serial, DMA, timers, Ethernet
- Solution developed with C++98/03 with some TR1 elements
- Return to multicore roots

SIMULATION OF TODAY

- Make it all work on a multicore system with C++11/14
  - without forgetting history of PDP-11s
- decltype, lambdas, rvalue-references/move semantics, nullptr, range-based for loops, auto, static_assert
- atomics, type traits, chrono, lock_guard, addressof, unique_ptr, regex, tuple

SIMULATION OF TODAY

Ring0 C++11 simulation framework (scheduler, diagnostics, …)



Deterministic simulation models

Some RTOS

Ring0 C++11 OS interface

CPUs

ACCEPTANCE

- FAA certified C++11 simulator currently in training
- More are on the way
- All FlightSafety simulators in 2015 will probably use C++11

C++: A FOREIGN VOCABULARY

- Developers may not be programmers first
  - Flight simulation modeler
    - VOR, ILS, DME, EFIS, HOT, PFD, APU, EOM, …
    - DI, DO, AI, AO, ARINC 429/629, MIL-STD1553, ARINC 664/AFDX, CAN, DR-11w, …
  - C++ developer
    - RAII, SFINAE, NSDMI, RTTI, CRTP, …

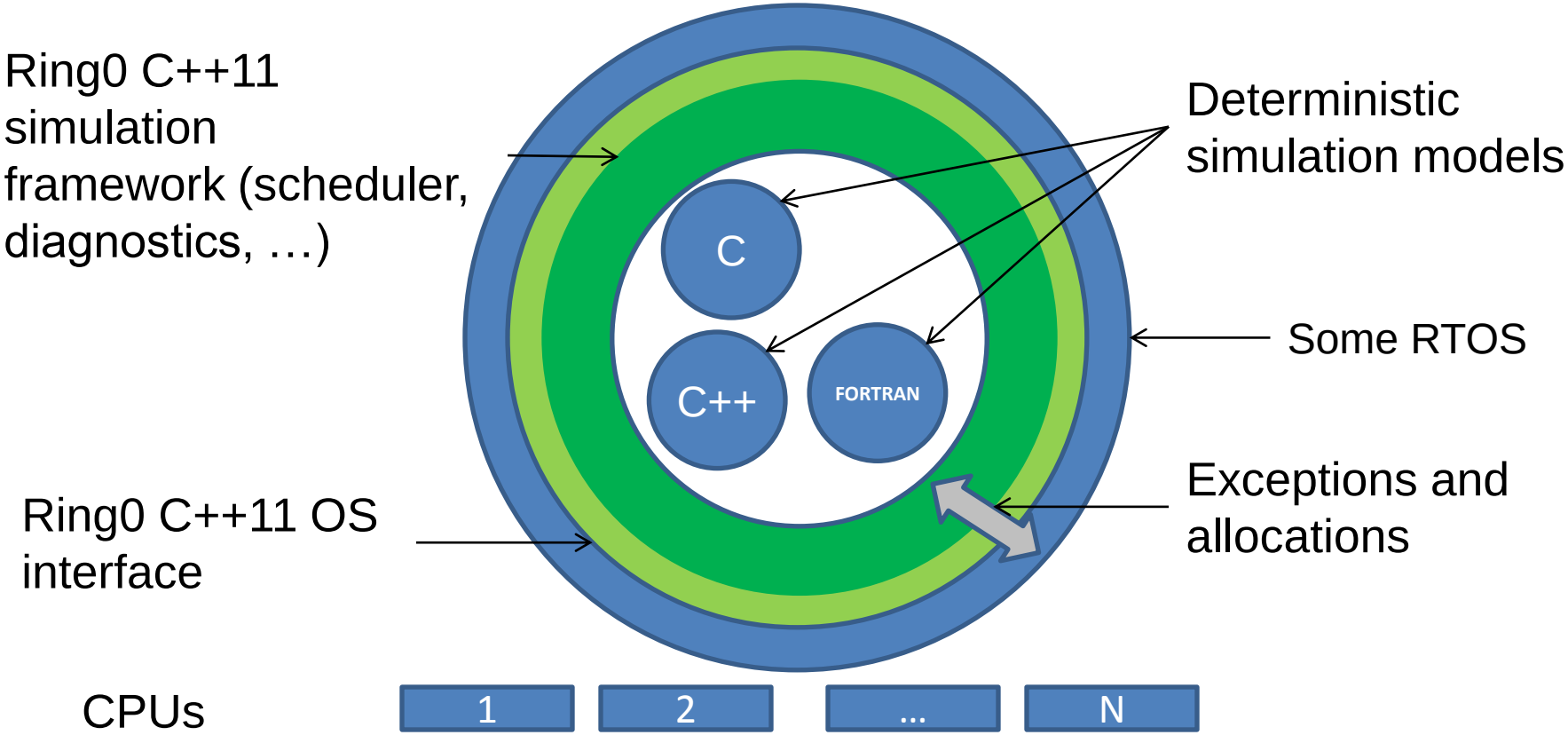
Late homework will not be excepted

- Tuesday: The Joint Strike Fighter Coding Standard: Using C++ on Mission and Safety Critical Platforms
- See keynotes for more on not using exceptions with low latency systems

EXCEPTIONS AND MEMORY

- At present, exceptions and memory allocations should be outside of our simulation models
- We aren’t even close to saturating a mid-range CPU
- If problems get larger to the point of requiring more processing, our exception usage may be reevaluated

EXCEPTIONS AND MEMORY



GETTING STARTED

- Starting an update development cycle can be huge
  - Why would you want to update a working codebase?
  - What parts of modern C++ can you use?
  - How can you update the working codebase?
  - Who is going to perform the work?
  - What should be done first?

WHY UPDATE

- APIs may be bloated and easy to misuse
  - Standard C++ may have a replacement
- Toolchain may no longer supported
- Hardware obsolete
- Need to update to 64bit
- Software maintenance costs may already be 60%
  - Robert Glass, Facts and Fallacies of Software Engineering

WHY: THE PLANETS ALIGNED

- Our APIs were bloated and easily misused
- Our toolchain was no longer supported
- Our hardware was end of life
- We needed to update to 64bit
  - New realtime operating system
- Our version control system was no longer supported

WHAT CAN YOU ACTUALLY USE

- Depends on compiler support
  - www.italiancpp.org/wp-content/uploads/2014/03/CppISO-Feb2014-r1.pdf
  - www.cpprocks.com
  - https://developer.mozilla.org/en-US/docs/Using_CXX_in_Mozilla_code
  - https://wiki.apache.org/stdcxx/C%2B%2B0xCompilerSupport
  - …
- Additional considerations
  - C++11 support in 3rd party libraries
  - Limited to compiler supplied with operating system

EXAMPLE: LLVM/CLANG/LLD

- llvm/clang/lld 3.4 last version built with C++98/03
- Builds using C++11 as of Feb 28, 2014
- “The set of features supported for use in LLVM is the intersection of those supported in MSVC 2012, GCC 4.7, and Clang 3.1.”

HOW TO UPDATE

- Brute force
- Write your own tools
  - Possibly using clang's LibTooling
- Use existing tools
  - Clang-modernize
  - Cevelop (www.cevelop.com)
  - More being developed every day

EXAMPLE: clang-modernize

- Transformations
  - Use nullptr
  - Use pass by value
  - Use range based for loops
  - Replace auto_ptr, use auto, and add override

EXAMPLE: clang-modernize

- Our experience
  - On Windows, might need to use headers from a different compiler
    - MinGW/MinGW-w64 works well (STL: www.nuwen.net)
  - Using nullptr was a common transformation
  - Using pass by value was not a common transformation

RISK: clang-modernize

- Important command line options
  - -risk
    - safe, reasonable (default), risky
  - -final-syntax-check

RISK: clang-modernize

```
#include <cstddef>
class Bar {
public:
        typedef int** iterator;
        iterator begin();
        iterator end();
        size_t size();
        int* operator[](size_t i);
};
int main(int, char**) {
        Bar ct;
        for (size_t i = 0; i < ct.size(); ++i) {
                int* f = ct[i];
        }
}
```

RISK: clang-modernize

- risk=safe
  - Transform: LoopConvert - Accepted: 0 - Rejected: 1 - Deferred: 0
- risk=reasonable
  - Transform: LoopConvert - Accepted: 1

```
int main(int, char**) {
    Bar ct;
    for (size_t i = 0; i < ct.size(); ++i) {
        int* f = ct[i];
    }
}
int main(int, char**) {
    Bar ct;
    for (auto & elem : ct) {            // <-----
        int* f = elem;                  // <-----
    }
}
```

RISK 2: clang-modernize

```
#include <cstddef>
class Bar {
public:
        typedef int** iterator;
        iterator begin();
        iterator end();
        size_t size();
        int& operator[](size_t i);            // <-----
};
int main(int, char**) {
        Bar ct;
        for (size_t i = 0; i < ct.size(); ++i) {
                int* f = &ct[i];              // <-----
        }
}
```

RISK 2: clang-modernize

- risk=safe
  - Transform: LoopConvert - Accepted: 0 - Rejected: 1 - Deferred: 0
- risk=reasonable
  - Transform: LoopConvert - Accepted: 1

```
int main(int, char**) {
    Bar ct;
    for (size_t i = 0; i < ct.size(); ++i) {
        int* f = &ct[i];
    }
}
int main(int, char**) {
    Bar ct;
    for (auto & elem : ct) { // <-----
        int* f = &elem;      // <----- Compile error: cannot convert from int** to int*
    }
}
```

RISK 3: clang-modernize

```
#include <cstddef>
class Bar {
public:
        typedef int** iterator;
        iterator begin();
        iterator end();
        size_t size();
        int& operator[](size_t i);
};
int main(int, char**) {
        Bar ct;
        for (size_t i = 0; i < ct.size(); ++i) {
                int* f = (int*)&ct[i];        // <----- danger!
        }
}
```

RISK 3: clang-modernize

- risk=safe
  - Transform: LoopConvert - Accepted: 0 - Rejected: 1 - Deferred: 0
- risk=reasonable
  - Transform: LoopConvert - Accepted: 1

```
int main(int, char**) {
    Bar ct;
    for (size_t i = 0; i < ct.size(); ++i) {
        int* f = &ct[i];
    }
}
int main(int, char**) {
    Bar ct;
    for (auto & elem : ct) {            // <-----
        int* f = (int*)&elem;           // <----- Runtime errors most likely
    }
}
```

RISK: THE ENGINES ARE BLEEDING

- Don’t be blinded by your tools
- Reviewing only the red rather than the whole line led to broken code

```
int* f = (int*)&elem;
int* f = (int*)&ct[i];
```

WHO WILL DO THE WORK: TRANSITIONS

WHO: C PROGRAMMER

```
/* no include statements */
int main(int argc, char** argv) {
        double r = pow(2.0, 3.0);
        printf("%f\n", r);
        return 0;
}
Result:
0.000000 (msvc)
8.000000 (clang, gcc)
```

- cl /W4 pow.c
  - warning C4013: 'pow' undefined; assuming extern returning int
  - warning C4013: 'printf' undefined; assuming extern returning int

WHO: C PROGRAMMER

- clang –c pow.c

```
pow.c:2:13: warning: implicitly declaring library function 'pow' with type
        'double (double, double)‘
        double r = pow(2.0, 3.0);
                   ^
pow.c:2:13: note: include the header <math.h> or explicitly provide a
        declaration for 'pow‘
pow.c:3:2: warning: implicitly declaring library function 'printf' with type
        'int (const char *, ...)‘
        printf("%f\n", r);
        ^
pow.c:3:2: note: include the header <stdio.h> or explicitly provide a
        declaration for 'printf'
```

WHO: C PROGRAMMER (TAKE 2)

```
extern void y(char*);
void do_y(char* p) {   // better name – do_y_and_modify_memory
        y(p);
        p[0] = '\0';
}

void x() {
        do_y("oops");
}
```

- C++03 deprecated string literal to char* conversion
- C++11, Annex C states that conversion is invalid
- g++: warning: deprecated conversion from string constant to 'char*'

WHO: C PROGRAMMER (TAKE 3)

```
class Foo {
public:
        ~Foo() {
                release(b1);
                ...
                release(bN);
        }
private:
        Bar b1;
        ...
        Bar bN;
};
```

- Bar probably needs a destructor calling release() so that adding another Bar does not require dtor changes

WHO: JAVA PROGRAMMER

```
void foo() {
    Bar b = *new Bar();
    ...
}
```

- No memory leak if copy constructor deals with the allocation (Richard Smith)
  - Don’t do this

```
void foo() {
    Bar b;              // the easy solution
    ...
}
```

WHO: ??? PROGRAMMER

```
class E {
public:
        virtual enum { NO, YES };     // virtual what? Wrong!
};
E e;
```

- Some versions of one vendor’s compiler have no problems with this
  - Future version: warning ‘virtual ‘: ignored on left of ’’ when no variable is declared
- g++: error: 'virtual' can only be specified for functions

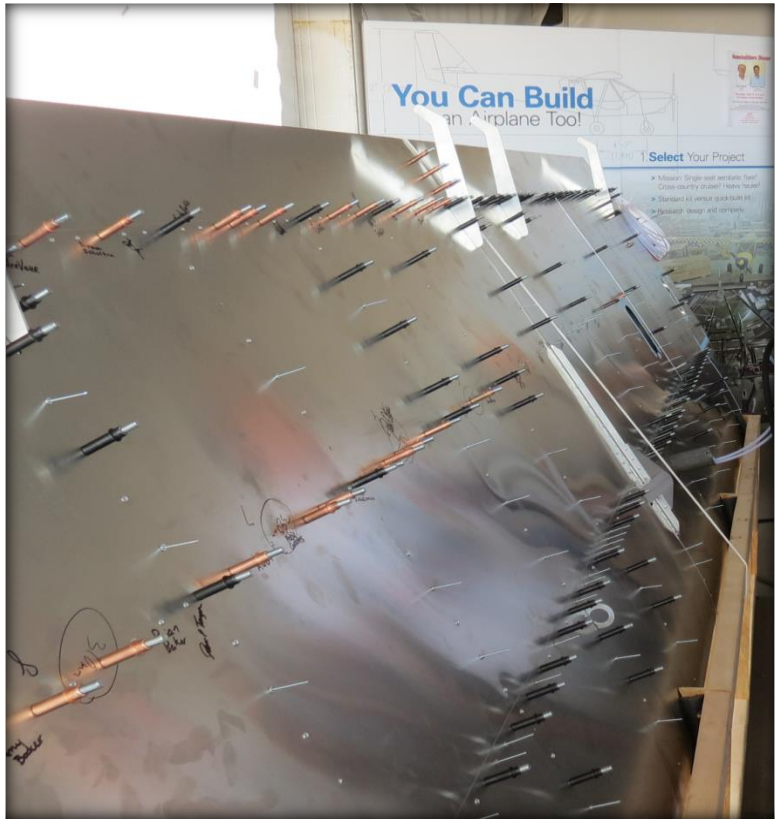
WHAT SHOULD BE DONE FIRST

- Internal changes may be easier
- Start with something you know
  - Lock/wait free data structures may not be the best place to start
- Replace boost:: with std:: or ???

FIRST: boost:: to std:: to ???

- What if your platform isn't an officially supported boost platform?
  - Linux, MacOS, QNX, Windows are listed on boost.org
- What if compiler's std::thread and std::mutex won't work on your platform?
  - As well as most other operating system interfaces
    - filesystem
    - networking?

BUILD IT YOURSELF



- Match the standard interfaces as much as possible for easy documentation

DO IT YOURSELF: CHRONO

```
#include <chrono>
#include <cstdint>
struct high_resolution_clock {
    typedef std::chrono::duration<int64_t, std::ratio<1, 10000000>>
duration;
    typedef duration::rep rep;
    typedef duration::period period;
    typedef std::chrono::time_point<high_resolution_clock> time_point;
    static const bool is_steady = true;

    static time_point now();
};
```

- now() built on top of IEEE1588, GPS hardware, or other sources

DO IT YOURSELF: THREAD

- A std::thread interface that, like boost, adds attributes to constructor. This makes Rate Monotonic or Earliest Deadline First scheduling easier
- A std::mutex interface allows usage with std::lock_guard and std::unique_lock

NONCOPYABLE TO MOVABLE

- Noncopyable probably means pointers in containers
- Double indirection via container may be bad for cache

NONCOPYABLE: UGH

```
// suppose Foo is not copyable
std::vector<Foo*> foos;

// initialize could also be a constructor
void initialize() {
        // for each foo in configuration file
        foos.push_back(new Foo(arg1, arg2));
}
```

- Would have required explicit traversal of foos to delete each element
- No modern C++ here

NONCOPYABLE: HO-HUM

```
// suppose Foo is not copyable
std::vector<std::unique_ptr<Foo>> foos;

void initialize() {
        // for each foo in configuration file
        foos.push_back(std::unique_ptr<Foo>(new Foo(arg1, arg2)));
}
```

- Automatic cleanup on vector destruction
- C++11 standard library usage

NONCOPYABLE: DOH

```
// suppose Foo is not copyable
std::vector<std::unique_ptr<Foo>> foos;

void initialize() {
        // for each foo in configuration file
        foos.push_back(std::make_unique<Foo>(arg1, arg2));
}
```

- Automatic cleanup on vector destruction
- C++14, now with std::make_unique, standard library usage

NONCOPYABLE: MOVABLE!

```
// suppose Foo is not copyable but movable
// i.e. Foo has Foo(Foo&&) constructor
std::vector<Foo> foos;

void initialize() {
        // for each foo in configuration file
        foos.emplace_back(arg1, arg2);
}
```

- C++11 standard library and core language usage

BONUS: NETWORKING

- Network byte order moved in Chicago to become first paper in TS working paper
- Network byte order moved to Library Fundamentals TS in Issaquah
- LEWG looked at N2175 in Rapperswil as potentially the starting point for the networking TS

BONUS: ISSUES FROM EMBEDDED

- Renewed interest at Rapperswil
- www.open-std.org/mailman/listinfo/embedded
- Early initialization function
- Main with noreturn attribute
  - Power cycle is only way to restart
  - No atexit() processing
    - Smaller footprint
- Removal of exception and RTTI overhead even when using the standard library

ENDING THE JOURNEY (FOR NOW)

Make sure you have test cases.
If you don’t try a feature, who will?
If you don’t report a bug, it won’t get fixed.