Why you should move your legacy code to smart pointers.

Sébastien Gonzalve

MUREX

CODE RECKONS

Science to the CORE

Preamble

- This talk is about refactoring
- Good refactoring rely heavily on tests (any kind of)
- If you do not have test, your priority is to have them, not to “change” the code
- If your code is EOL, don’t change it

Ever seen this already?

```
struct SomeClass {
 std::vector<int *> collection;
 // no copy constructor ?!
 ~SomeClass {
     for (auto &e : collection) {
         delete(e);
     collection.clean();
 }
};
```

Problem

- C++ 11 introduces smart pointers, but many of use have huge code bases formerly written in C++03.
- This talk aims to explain why the right moment to move your legacy code to smart pointers is *now*.

- But.... Why one would not want to use modern solutions, after all?


Modern problems require modern solutions

Common motivation for not changing

01

FEAR OF LOSS/CONFUSION

02

MORE WORK

03

FEAR OF FAILURE

Fear of Loss/Confusion

- One need to understand the reason for the change
- Evidences:
  - “That was working well until now”
  - “We always did like this”
  - “Some of us don’t know C++11”

Why would we lose something?

- How do we know that it works?
  - Are there unit tests and/or validation tests? (with correct coverage)
  - Does it *always* work? (what about exceptions?)
  - Are there tools passed to check leaks and/or corruptions?

More Work

- Changing former code is unplanned *work*
- Evidences:
  - “We’ll do that next year”
  - “We need to take it into account for the next product”
  - “ROI is nul”

Preventing leaks means no code

```
SomeClass::~SomeClass {
      for (auto &e : collection) {
          delete(e);
      collection.clean();
  }

becomes:
```

More work? Cost of adding new features

- One need to be convinced that benefits of change will come fast
- Are there upcoming evolutions of the code
  - How will we make sure that new code is correct?
  - How will we make sure that new code does not break the former code?
  - What will be the price of having it work again?
- What about enforcing correct use of APIs?

Fear of Failure

- One need to feel he will be able to be successful
- Evidences:
  - “Compiler keeps giving me 6 pages of error bloat I don’t understand”
  - “Our code base is too large”
  - “I don’t like C++11”

Get help

- Training team is sometime needed prior to start
- You may try different compilers
- There are a lot of resources available on the internet especially the CppCoreGuidelines

When to pass smart pointer as function parameter

When not expressing ownership, don't pas them

- F.7: For general use, take T* or T& arguments rather than smart pointers
- R.3: A raw pointer (a T*) is non-owning

When ownership need to be expressed, pass by value

- R.32: Take a unique_ptr<widget> parameter to express that a function assumes ownership of a widget
- R.34: Take a shared_ptr<widget> parameter to express that a function is part owner

(ref CppCore Guidelines)

When to pass smart pointer as function parameter

When you need to be able to reseat content, pass references

- R.33: Take a unique_ptr<widget>& parameter to express that a function reseats the widget
- R.35: Take a shared_ptr<widget>& parameter to express that a function might reseat the shared pointer

(ref CppCore Guidelines)

You can be the one that bear the change!

Some usual patterns

Example: Evolving class

Say we have this class:

```
struct SomeClass {
   std::map<std::string, std::string> registery;
};
```

Someone simply add a new field:

```
struct SomeClass {
 std::map<std::string, std::string> registery;
 char *buffer = malloc(128);
 ~SomeClass() { delete buffer; }
};


int main() {
 SomeClass s1;
 { SomeClass s2; s1 = s2; }



}
```

Evolving class case

With:

```
struct SomeClass {
 std::map<std::string, std::string> registery;
 std::unique_ptr<char[]> = std::make_unique<char[]>(128);
 // destructor no more needed
};
```

And the following wont **compile**

```
int main() {
 SomeClass s1;
 { SomeClass s2; s1 = s2; }

 for (size_t i = 0; i < 128; i++)
    s1.buffer[i] = 0xEE;
}
```

```
In function 'int main()':
error: use of deleted function 'SomeClass& SomeClass::operator=(const SomeClass&)'
   17 |         s1 = s2;
note: 'SomeClass& SomeClass::operator=(const SomeClass&)' is implicitly deleted because the
default definition would be ill-formed:
    5 | struct SomeClass {
error: use of deleted function 'std::unique_ptr<…>::operator=(const std::unique_ptr<…>&)'
unique_ptr& operator=(const unique_ptr&) = delete;
```

Callback use case (1)

- Say you have an object taking a callback function

```
struct Observee {
  void registerCb(std::function<void()> cb) {
   _cb = cb;
  }
  void call() {
   _cb();
  }
 std::function<void()> _cb;
};
struct SomeClass {
 void *x = this;
 void doSomething(Observee &o) {
   o.registerCb([this] {// this is captured but how to know if callback won't be called after this is destroyed
         std::cout << "Hello " << x << '\n';
   });
 }
};
int main() {
 Observee o;
 {
   SomeClass s;
   s.doSomething(o);
   o.call(); // Ok
 }
 o.call(); // BUG !! s was destroyed
}
```

Callback use case (2)

https://godbolt.org/z/Ke5n4P5b3

```
struct Observee {
  void registerCb(std::function<void()> cb) {
   _cb = cb;
  }
  void call() {
   _cb();
  }
 std::function<void()> _cb;
};
struct SomeClass {
 std::shared_ptr<std::any> _cookie = std::make_shared<std::any>(); // use a shared cookie to track object
 void *x = this;
 void doSomething(Observee &o) {
   std::weak_ptr<std::any> cookie = _cookie;
   o.registerCb([cookie, this] {
      if (auto s = cookie.lock()) { // when cookie cannot be accessed, object was destroyed
         std::cout << "Hello " << x << '\n';
      } else {
        std::cout << "CB discarded because object was destroyed...." << '\n';
      }
   });
 }
};
int main() {
 Observee o;
 {
   SomeClass s;
   s.doSomething(o);
   o.call(); // Ok
 }
 o.call(); // s was destroyed but no crash
}
```

Using C APIs

We use low level C libraries that do not take smart pointers.

- How do you handle resources of those libraries?
- Are you sure there is no misuse of APIs using pointers?

Smart pointers allow to use RAII over C api.

FFmpeg (C api) example (1).

```
/* frame containing input raw audio */
auto frame = av_frame_alloc();
if (!frame) {
 fprintf(stderr, "Could not allocate audio frame\n");
 return 1;
}

// [...]

uint16_t *samples;
/* setup the data pointers in the AVFrame */
ret = avcodec_fill_audio_frame(frame, c->channels, c->sample_fmt,
                               (const uint8_t*)samples, buffer_size, 0);
if (ret < 0) {
 fprintf(stderr, "Could not setup audio frame\n");
 return 1; // <= leak!
}

// use samples
av_frame_free(&frame);
```

FFmpeg (C api) example (2).

```
/* frame containing input raw audio */
auto frame = av_frame_alloc();
if (!frame) {
 fprintf(stderr, "Could not allocate audio frame\n");
 return 1;
}

// [...]

uint16_t *samples;
/* setup the data pointers in the AVFrame */
ret = avcodec_fill_audio_frame(frame, c->channels, c->sample_fmt,
                               (const uint8_t*)samples, buffer_size, 0);
if (ret < 0) {
 fprintf(stderr, "Could not setup audio frame\n");
 av_frame_free(&frame); // C style resource handling
 return 1;
}

// use samples
av_frame_free(&frame);
```

FFmpeg (C api) example (3).

```
/* frame containing input raw audio */
std::unique_ptr<AVFrame, void(*)(AVFrame*)> frame(av_frame_alloc(), av_frame_free);
if (!frame) {
 fprintf(stderr, "Could not allocate audio frame\n");
 return 1;
}

// [...]

uint16_t *samples;
/* setup the data pointers in the AVFrame */
ret = avcodec_fill_audio_frame(frame.get(), c->channels, c->sample_fmt,
                               (const uint8_t*)samples, buffer_size, 0);
if (ret < 0) {
 fprintf(stderr, "Could not setup audio frame\n");
 // No more av_frame_free call needed
 return 1; // leak!
}

// use samples
// No “av_frame_free” call needed anymore thanks to RAII
```

FFmpeg (C api) example (4).

```
/* This node not compile ☹ */
std::unique_ptr<AVFrame, void(*)(AVFrame*)>(av_frame_alloc(), av_frame_free);

/* delete takes a AVFrame** not AVFrame* ☹ */
void av_frame_free(AVFrame **frame)
```

FFmpeg (C api) example (5).

```
/* We can take a lambda */
std::unique_ptr<AVFrame, void(*)(AVFrame*)>(av_frame_alloc(),
                                            [] (AVFrame *f) { av_frame_free(&f);} );
```

But the syntax remains quite clumsy and you need to specify the lambda each time you use it.

FFmpeg (C api) example (6).

```
struct FrameDeleter {
  void operator()(AVFrame *f) { if (f) av_frame_free(&f); }
 };

using UniqueFrame = std::unique_ptr<AVFrame, FrameDeleter>;
```

Then usage becomes:

```
UniqueFrame frame{av_frame_alloc()};
```

FFmpeg (C api) use case (7).

```
/* frame containing input raw audio */
UniqueFrame frame{av_frame_alloc()};
if (!frame) {
 fprintf(stderr, "Could not allocate audio frame\n");
 return 1;
}

// [...]

uint16_t *samples;
/* setup the data pointers in the AVFrame */
ret = avcodec_fill_audio_frame(frame.get(), c->channels, c->sample_fmt,
                               (const uint8_t*)samples, buffer_size, 0);
if (ret < 0) {
 fprintf(stderr, "Could not setup audio frame\n");
 return 1; // leak!
}

// use samples
```

Use shared_ptr wisely

- Smart pointers are really helpful to express ownership
- Refactor to use smart pointers should be a chance to challenge design and implementation
- Resist the “let’s put shared_ptr everywhere because is don’t understand who owns this”

Advices

- Easier to start from upper layer and go down your class hierarchy
- Migrate one thing at the time (one variable, one function, etc...)
- Work with little iterations and test as soon as you get something to compile
- Be ready to put on hold some part when discovering more priority work
- Be aware that you will likely experience crashes when encountering ownership problems in your design => fix the design first

This is *really* better

- Eliminate dangling pointers (a very common source of bugs)
- Enforce design + Make the code self documented about ownership of data
- Reduce mental load when debugging
- Make sure the is no misuse of new[]/delete or malloc/delete

- **Using smart pointers does not only make the code safe *now*, but also will help keep it safe in the *future***

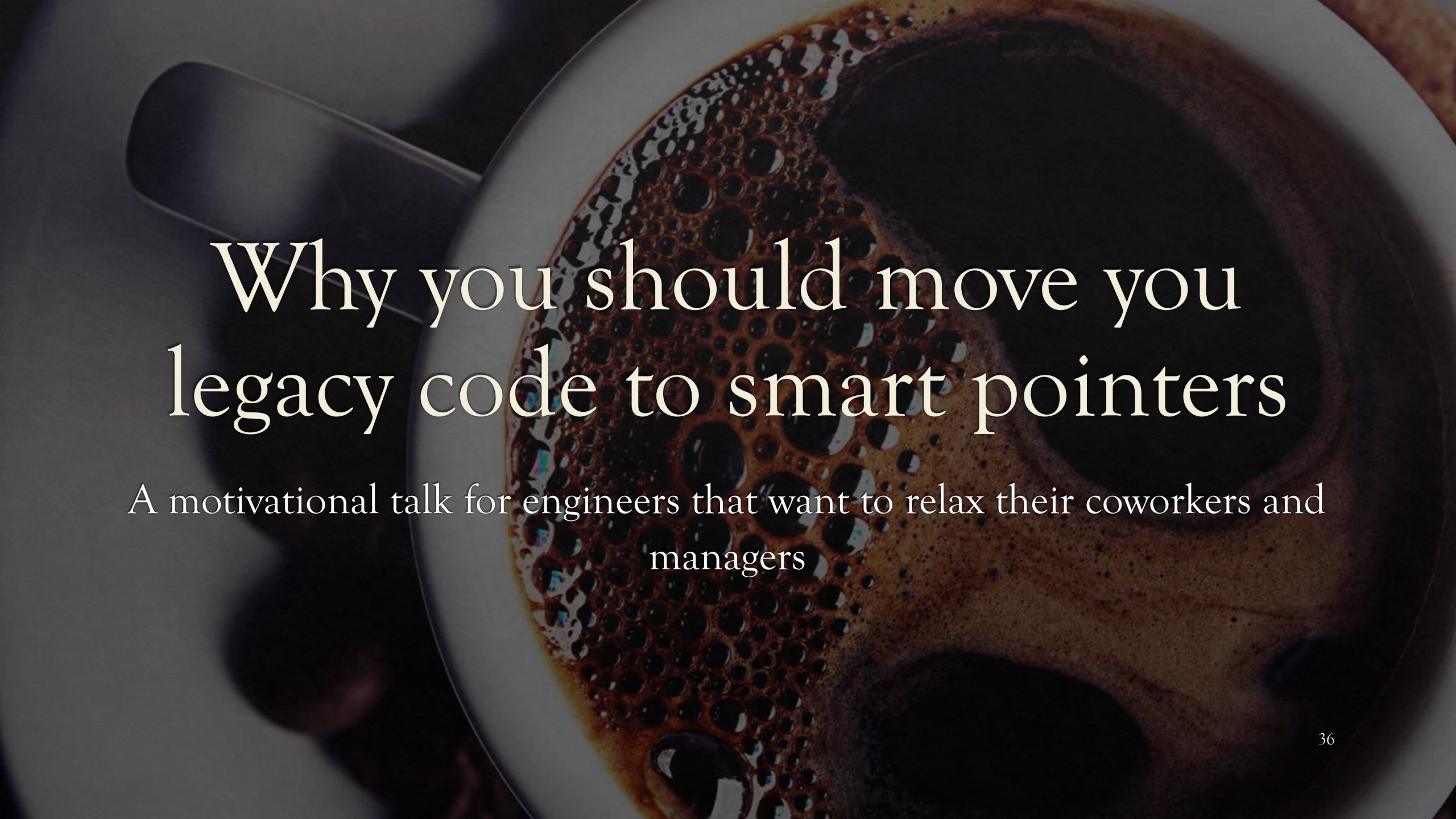
Questions

Thank you!

Any Feedback is welcome

sebastien.gonzalve@aliceadsl.fr

Backup slides



Why you should move you legacy code to smart pointers

A motivational talk for engineers that want to relax their coworkers and managers

Pimpl idom and smart pointers

```
struct Details; // Forward declaration
struct SomeClass {
 private:
   Details *detail; // pimpl idom
};
```

```
struct Details; // Forward declaration
struct SomeClass {
 ~SomeClass(); // Careful, Body *must* be in cpp file
               // SomeClass::~SomeClass() = default;
 private:
   unique_ptr<Details> detail;
};
```

Impact on performances

- Unique_ptr has no impact on performances
- Are there benches of measurements of those performances?
- Shared ptr have little overhead that may be measured, and balanced with the risk of not using them or using cunstom reference counting

How to proceed

- Begin with quick wins: private members
  - First focus on unique_ptr<>
  - Pass references when no pointer was needed
  - Keep pointer for third party interfaces that don't take ownership
  - Remove all code that is useless (cleanups loops, initializations, destructors…)
- Identify factories and migrate them
- Replace *new T[X]* by vector<T> or *make_unique<T[]>(X)*
- Find C-like function and wrap them (see next section)
- Whenever a raw pointer was passed through an interface, question:
  - Is a pointer really needed?
  - Who is owner of the object; is it eventually shared?