GENERATING PROTOCOL TESTS AT RUNTIME

Edouard Alligand
quasardb

TALK LAYOUT

- Few words about template meta-programming
- The use case
- The approach
- The solution

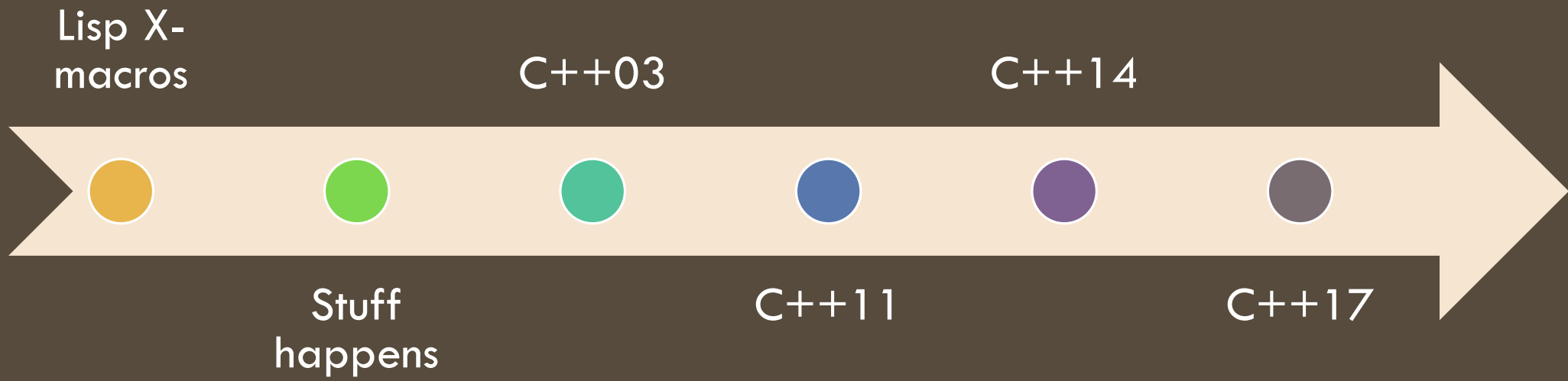
TEMPLATE METAPROGRAMMING

It is awesome and if you disagree you are wrong

WHY SHOULD YOU CARE?

- Some problems are just code generation problems
  - Code generation requires automation
  - Meta-programming is type safe automation
- Compile-time techniques keep gaining traction in the language
  - The usefulness of `static_assert` and friends
  - The elegance of `constexpr`
- Metaprogramming is a good mental exercise
  - Force you to work in a completely different mind frame
  - Improve your other C++ related skills

A BRIEF HISTORY




LIGHTSPEED IS TOO SLOW. WE'LL HAVE
TO GO RIGHT TO LUDICROUS SPEED.

BOOST.MPL

```
typedef vector<int, char, long, short, char, short, double, long> types;

typedef count<types, short>::type n;


BOOST_MPL_ASSERT_RELATION( n::value, ==, 2 );
```

C++ 11 TMP – BY HAND

```
template <class... T> struct list {};

template <class... T>
using count = std::integral_constant<unsigned int, sizeof...(T)>;

template <class A, template<class...> class B> struct wrap;

template<template<class...> class A,
class... T, template<class...> class B>
struct wrap<A<T...>, B>
{
    using type = B<T...>;
};

template<class L> using size = typename wrap<L, count>::type;

using types = list<int, char, long, short, char, short, double, long>;

static_assert(size<types>::value == 8, ":o");
```

BRIGAND

```
using types = brigand::list<int, char, long, short, char,
short, double, long>;

static_assert(brigand::size<types>::value == 8, ":o");
```

WHAT IS BRIGAND?

- Brigand is a C++ 11 meta-programming library
  - Boost software license
  - Light-weight
    - Roughly 2,000 lines
  - Fully functional
    - All the features of Boost.MPL and more
    - Already used in production
  - Instant-compile time
    - More than 200 tests
    - ~10s compile time on MSVC 2013

https://github.com/edouarda/brigand

CONSTEXPR?

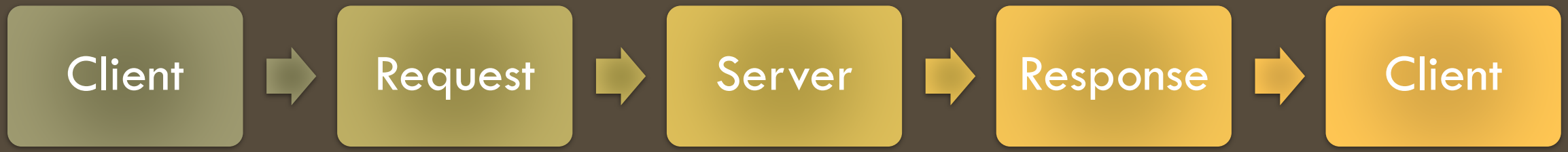
We can now write:

```
constexpr int add(int a, int b) { return a + b}
static constexpr int val = add(2, 3);
```

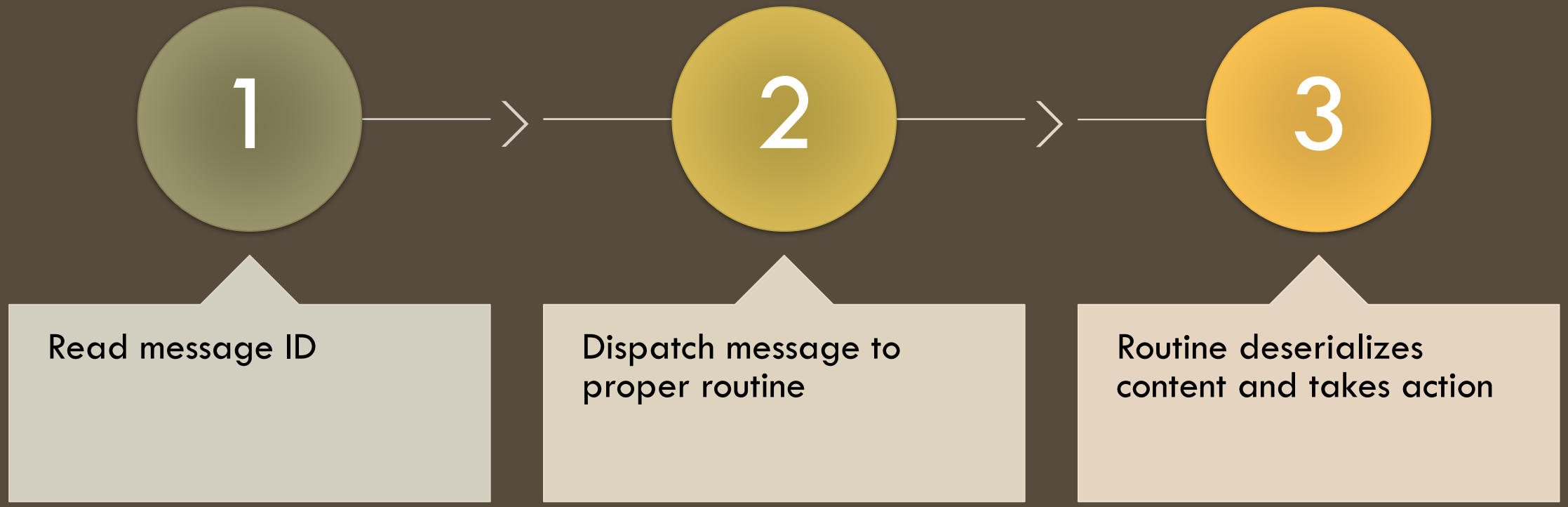
For more info: http://blog.quasardb.net/demystifying-constexpr/

THE USE CASE

Where the presenter will brag and show off

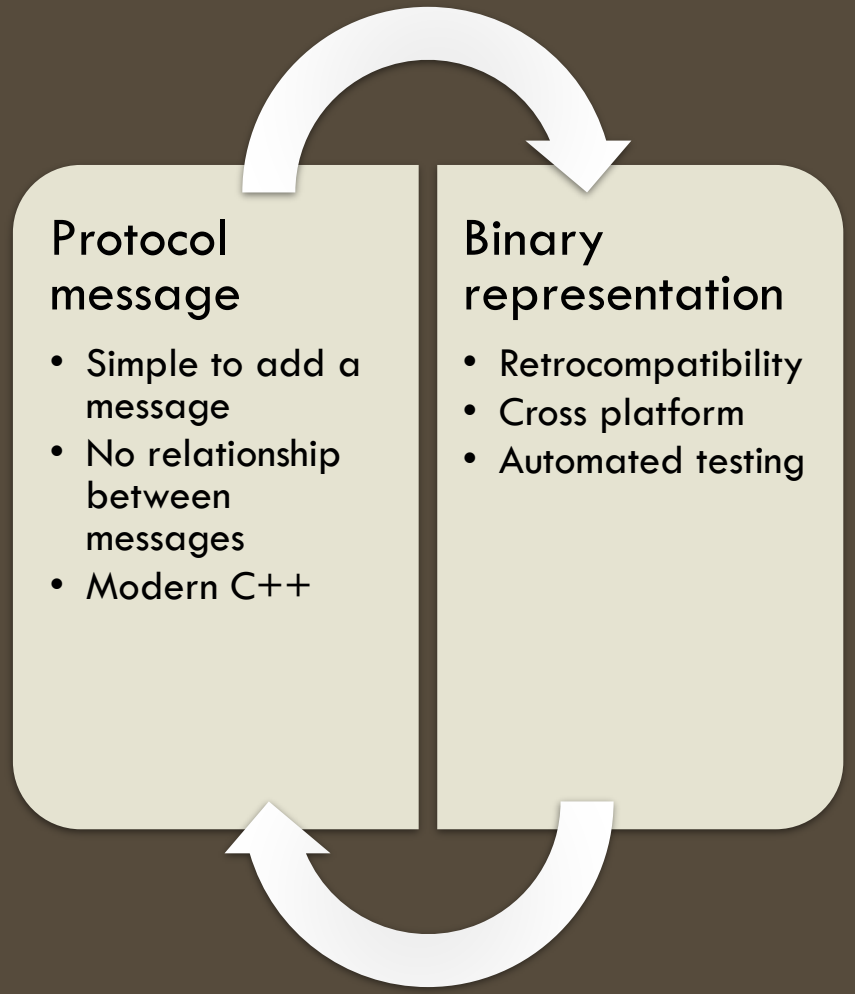


QUASARDB: TIME SERIES AT SCALE


1
2
3
Read message ID
Dispatch message to proper routine
Routine deserializes content and takes action

A MESSAGE

THE PROBLEM


Protocol message
• Simple to add a message
• No relationship between messages
• Modern C++
Binary representation
• Retrocompatibility
• Cross platform
• Automated testing

Backward compatibility

Forward compatibility

Resilience

Behavior

WHAT WE WANTED TO TEST

A MESSAGE

```
struct put
{
    using id = id_type<30>;
    using batchable = std::true_type;
    using replicable = std::true_type;

    put() {}

    put(boost::asio::const_buffer a, boost::asio::const_buffer c,
qdb::timespec e) : alias(a), content(c), expiry_time(e)
    {
    }

    boost::asio::const_buffer alias;
    boost::asio::const_buffer content;
    qdb::timespec expiry_time;
};
```

HOW SERIALIZATION WORK

```
BOOST_FUSION_ADAPT_STRUCT(qdb::protocol::blob::put,
      alias,
      content,
      expiry_time);
```

Then

```
protocol::blob::put blah;
serialization::marshal(id_encoder(blah), buffer);
```

Serialization/deserialization

Proper execution

Proper parameters passing

THIS MEANS WE NEED TO WRITE TESTS TO CHECK

VERY BRIEF BRIGAND OVERVIEW

Shameless plug

CONCEPT OF COMPILE-TIME OPERATIONS

- Types are first-class values inside compile-time programs
- Any template class accepting a variable number of type parameters can be considered a type container

```
// void is supported
using my_list = brigand::list<int, void>;

// tuple is variadic, hence a list
using foo = std::tuple<char, double, int[4]>;
```

OPERATIONS ON TYPE LISTS

```
// Add a type to an existing list
using list2 = brigand::push_back<list1, char>;

// Pop an element from a list
using list3 = brigand::pop_front<list2>;

// Concatenate two lists
using list1_and_2 = brigand::append<list1, list2>;
```

More information: https://github.com/edouarda/brigand/wiki/Introduction

TRANSFORM A LIST OF TYPES

```
using vanilla_list = brigand::list<char, bool, int>;

// ptr list will be 'char *, bool *, int *
using ptr_list = brigand::transform<vanilla_list,
                                    std::add_pointer<brigand::_1>>;
```

More information: https://github.com/edouarda/brigand/wiki/Algorithms

FIND A TYPE WITHIN A LIST

```
using my_list = brigand::list<int, bool, char>;

// find_result will be 'bool, char'
using find_result = brigand::find<my_list,
                                  std::is_same<brigand::_1, bool>>;

// bool_found will be std::true_type
using bool_found = brigand::found<brigand::find<my_list,
                                  std::is_same<brigand::_1, bool>>>;
```

More information: https://github.com/edouarda/brigand/wiki/Algorithms

RUNTIME ALGORITHMS – PRINT TYPES

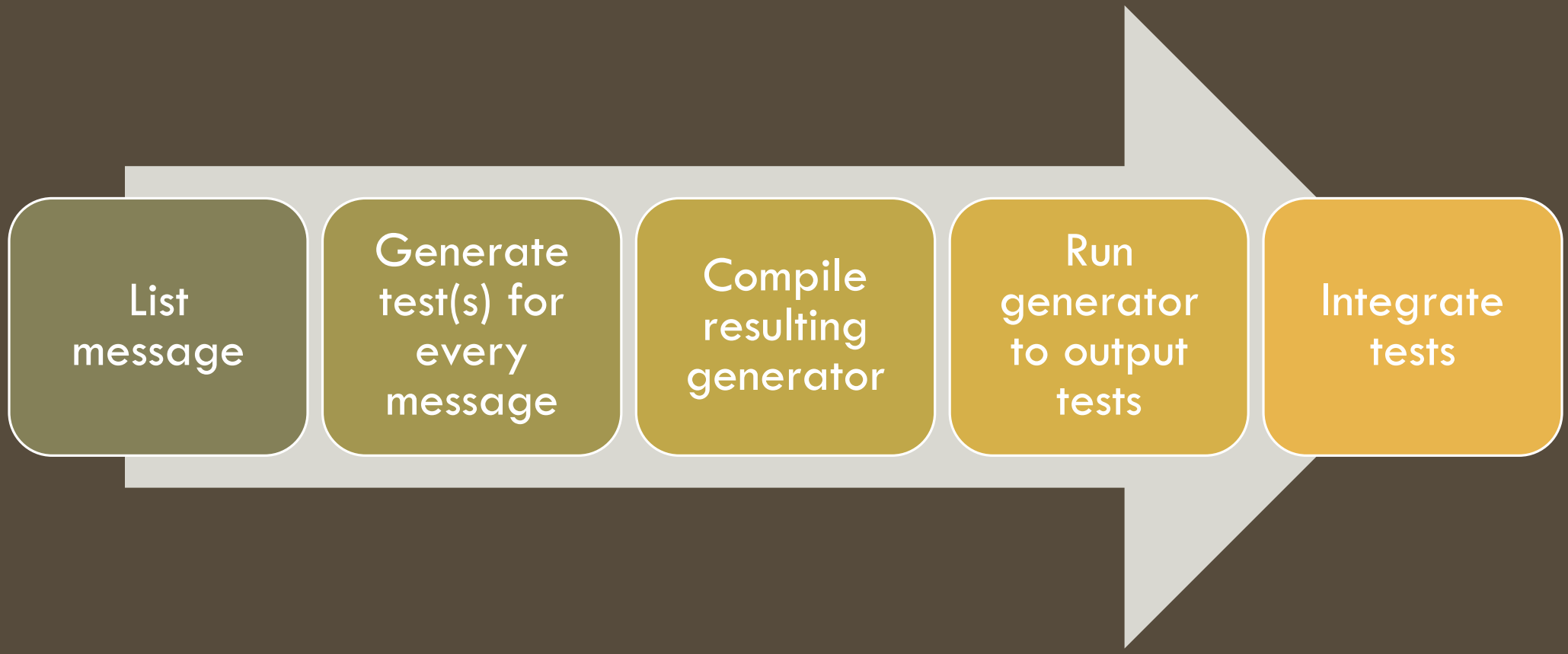
```
using my_list = brigand::list<char, int, bool>;

struct f
{
    template <typename T>
    void operator()(brigand::type_<T>)
    {
        std::cout << typeid(T).name() << std::endl;
    }
};

brigand::for_each<my_list>(f{});
```

More information https://github.com/edouarda/brigand/wiki/Runtime

THE SOLUTION

Where we see how we can use all we learned today to make something actually useful

THE SOLUTION



FIRST, MAKE A LIST

Regroup all messages in list:

```
using command_messages = brigand:list<command_one, command_two>;
using blob_messages = brigand::list<blob_get, blob_put>;
```

Then make a list of lists:

```
using messages = brigand::append<command_messages, blob_messages>;
```

FILTER THE LIST

Let's imagine we don't want to include deprecated messages. We could mark the deprecated messages as such:

```
struct my_message
{
    using id = std::integral_constant<std::uint16_t, 2>;
    using deprecated = std::true_type;
    std::string blah;
    std::uint64_t flags;
};
```

FILTER THE LIST

And we filter them out as such:

```
BOOST_TTI_HAS_TYPE(deprecated);

using to_test_list = brigand::remove_if<messages_list,
has_type_deprecated<brigand::_1>>;
```

SORT THE LIST

For convenience, we'd like to sort the list by IDs:

```
using unordered_messages = // complete_requests_list;

template <typename Left, typename Right>
using compare_ids = brigand::less<typename Left::id,
                                  typename Right::id>;

using ordered_messages = brigand::sort<unordered_messages,

brigand::quote<compare_ids>>;
```

GENERATE CODE

Now, for each message in the list « generate the code »

```
struct my_functor
{
    template <typename Message>
    void operator()(brigand::type_<Message>) const
    {
        // for each message, generate the test code
        // and output it on the console
    }
};

brigand::for_each<messages_list>(my_functor{});
```

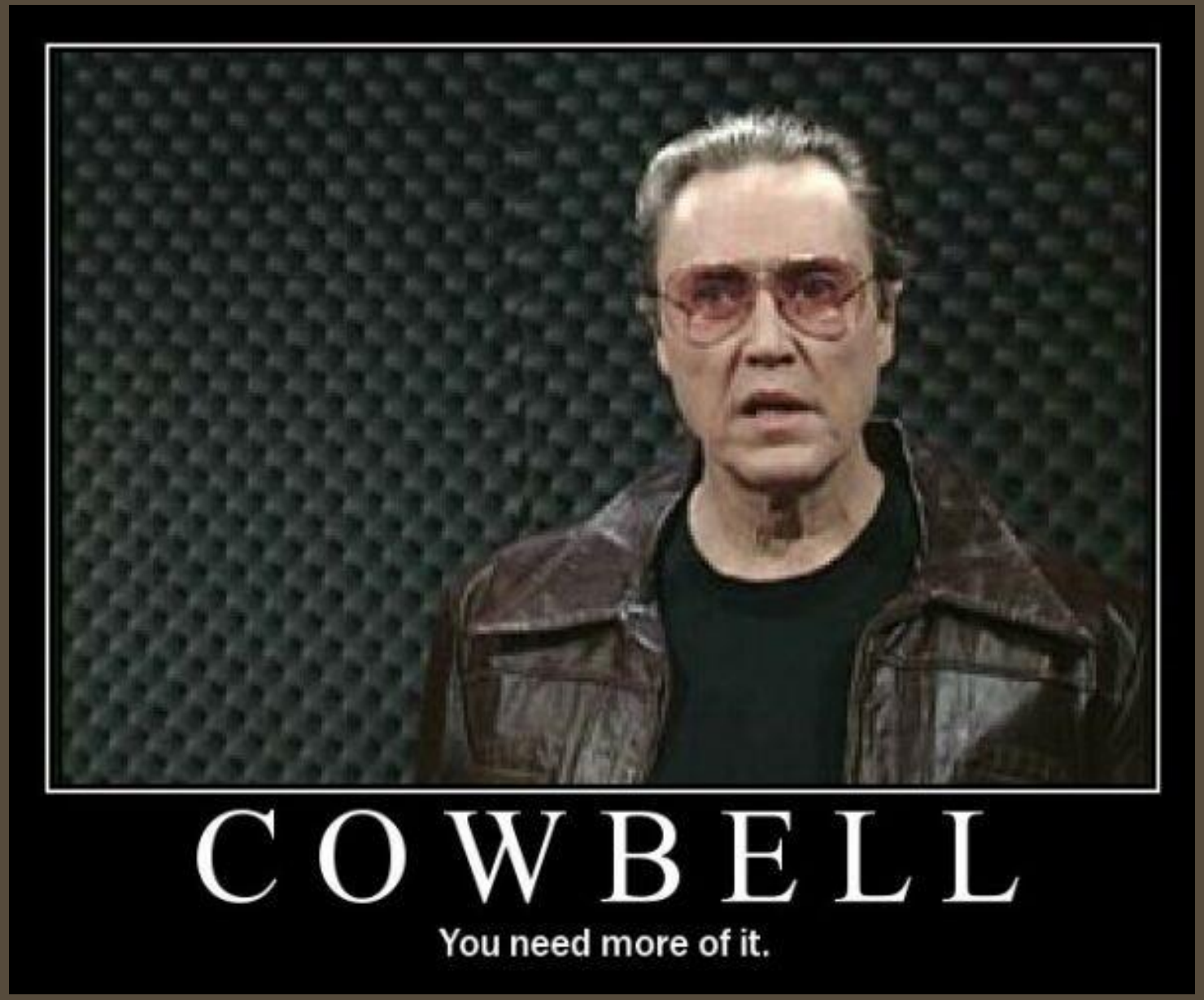
Just run the generated code

against predetermined

binary strings

and run that in continuous integration

THE SOLUTION ?


COWBELL
You need more of it.

EXTRA! EXTRA!

How to make sure at compile time I don't have duplicate IDs and that no ID is zero?

```
// Red alert! Shields up!
struct message_one
{
    using id = std::integral_constant<std::uint16_t, 1>;
};

struct message_two
{
    using id = std::integral_constant<std::uint16_t, 1>;
};
```

NO ZERO ID AT COMPILE TIME

```
struct get_id
{
    template <typename T>
    struct apply
    {
        using type = typename T::id;
    };
};

using id_list = brigand::transform<requests_list, brigand::bind<get_id, brigand::_1>>;

static_assert(brigand::not_found<brigand::find<id_list, brigand::equal_to<brigand::_1,
id_zero>>>::value, "id of a message cannot be zero");
```

NO DUPLICATE ID AT COMPILE TIME

```
struct inserter
{
    template <typename A, typename B>
    struct apply
    {
        using type = typename brigand::insert<A, B>;
    };
};

using id_set = brigand::fold<id_list, brigand::set<>, brigand::bind<inserter,
brigand::_state, brigand::_element>>;

static_assert(brigand::equal_to<brigand::size<id_set>,
brigand::size<id_list>>::value, "duplicate message id found!");
```

CONCLUSION

Template metaprogramming became much easier since C++ 11

- It's even easier in C++ 14, and things will continue to improve

Template metaprogramming is not just a mental exercise, it's actually useful for production code

CONCLUSION CONTINUED

Leveraging TMP techniques greatly increases our productivity

- A lot of mistakes are avoided at compilation time
- We don't waste time writing tests: they are generated for us
- Code is super efficient

LINKS AND REFERENCES

Tiny Metaprogramming Library by Eric Niebler

- http://ericniebler.com/2014/11/13/tiny-metaprogramming-library/

Simple C++ 11 Metaprogramming by Peter Dimov

- http://www.pdimov.com/cpp2/simple_cxx11_metaprogramming.html

The Brigand TMP library

- https://github.com/edouarda/brigand

Practical C++ Metaprogramming by Edouard Alligand and Joel Falcou

- http://www.oreilly.com/programming/free/practical-c-plus-plus-metaprogramming.csp

THANKS !

https://github.com/edouarda/brigand