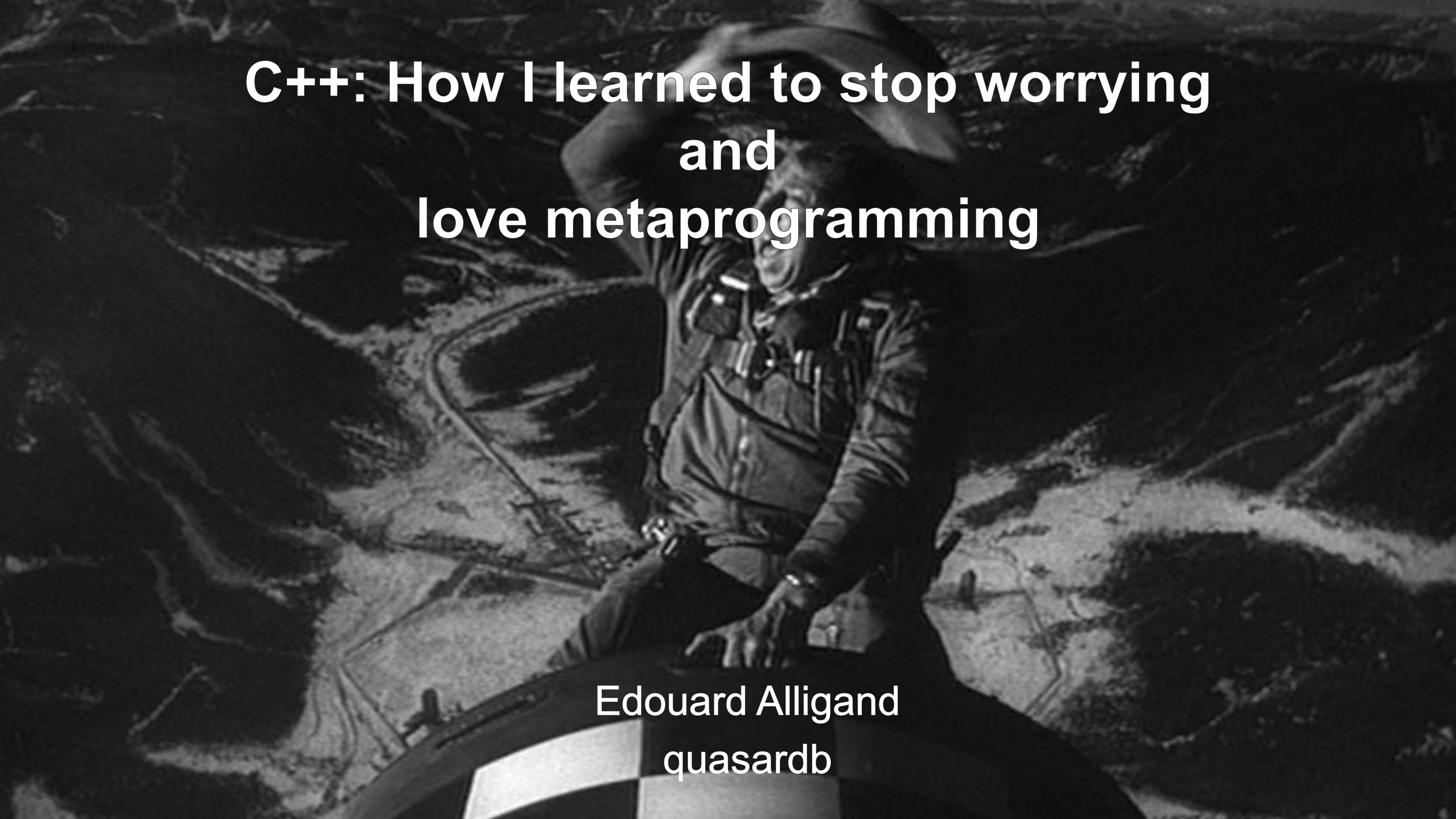
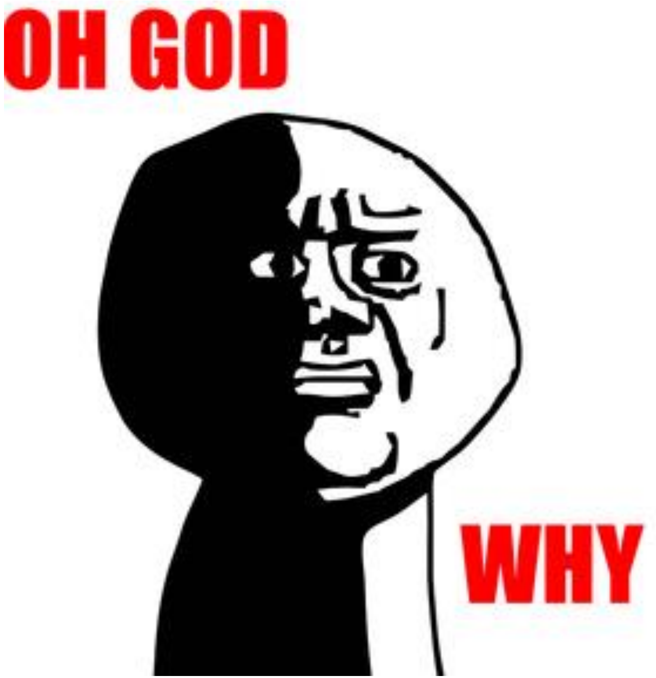
C++: How I learned to stop worrying and love metaprogramming



Edouard Alligand
quasardb

Metaprogramming



My first C++ metaprogram, ever

```
int main(int argc, char ** argv)
{
        static_assert(sizeof(int) == 4, “I want 32-bit integers!”);
        static_assert(LITTLE_ENDIAN, “I want little Endian!”);
}
```

Life of a C++ metaprogrammer

quasardb®

AN UNLIMITED KEY-VALUE STORE

WWW.QUASARDB.NET

What we use metaprogramming for

- Checks
- Constant generation
- Parser generator
- Protocol generation
- Serializer generator

BUT IT IS HARD/IMPOSSIBLE/ILLEGAL/DANGEROUS AND MY DOG ATE MY HOMEWORK

5 reasons to not do template-metaprogramming

1. **Compilation time**
2. **Code complexity**
3. **Compiler support**
4. **Skills**
5. **Non-rational reasons**

But, what is metaprogramming, really?

Programming

Code → Compiler → Binary

Metaprogramming

Why we do it

Have the compiler generate the correct program for you

- It knows things you don’t
- It will not be afraid to do tedious work
- It makes less mistakes than you
- Why write code when you could be playing Baldur’s Gate 2 for the 10th time?

Also: it’s awesome

A gentle introduction

Checking types, again

```
#include <type_traits>

template <typename T>
T factorial(T v)
{
        using threshold = std::integral_constant<T, 1>;
        static_assert(std::is_integral<T>::value, "I need an integral type");
        return (v <= threshold::value) ? v : (v * factorial(v - 1));
}
```

On the usefulness of compile-time computations

Why write

```
static const long magic = 120; // magic value is 5!
```

Which one year later will be

```
static const long magic = 720; // magic value is 5!
```

On the usefulness of compile-time computations

When you can write

```
static const long magic = factorial<5>::value;
```

Which one year later will be

```
static const long magic = factorial<6>::value;
```

Computing factorial at compile time

```
template <long v>
struct factorial :
    std::integral_constant<long, v * factorial<v - 1>::value> {};

factorial<2>     ⬄ std::integral_constant<long,
                        2 * std::integral_constant<long, 1>::value>
                 ⬄ std::integral_constant<long, 2 * 1>
                 ⬄ std::integral_constant<long, 2>
```

Writing the termination with template specialization

```
template <>
struct factorial<0> : std::integral_constant<long, 1> {};
```

The whole program

```
#include <type_traits>

template <long v>
struct factorial :
    std::integral_constant<long,
                           v * factorial<v - 1>::value> {};

template <>
struct factorial<0> : std::integral_constant<long, 1> {};
```

Brigand – A C++ 11 metaprogramming library

A **brigand** is a person who usually lives in a gang and lives by pillage and robbery.

A lightweight C++ 11 meta-programming library

https://github.com/edouarda/brigand

Lightning talk tomorrow!

Compile time lists

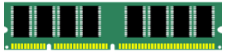
```
namespace brigand
{
    template <class... T> struct list {};
}
```

Example:

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

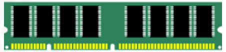
List vs list

```
char my_list[4] =
{ ‘a’, ‘b’, ‘c’, ‘d’};
```



“a” “b” “c” “d”

```
brigand::list<char, int,
bool, int>;
```



∅

Why a collection of types?

```
using allowed_types = brigand::set<char, int, bool>;

template <typename T>
void my_func(T t)
{
    static_assert(brigand::contains<allowed_types, T>::value,
      "Pain au chocolat >> chocolatine");
}
```

Last element of a 2 elements list

```
// if we have only two elements…

template <class H, class T>
struct last_element
{
    using type = T;
};
```

Last element of a 3 elements list

```cpp
// if we have only three elements…

template <class H, class M, class T>
struct last_element
{
    using type = T;
};
```

Last element of a list

```
template <class H, class… R>
struct last_element
{
        // when not a trivial case
        // make it a trivial case
};
```

Last element of a list

```
template <class… R> struct last_element;

template <class T>
struct last_element<T> { using type = T; };

template <class T, class… R>
struct last_element<T, R…>
{
    using type = typename last_element<R…>::type;
};
```

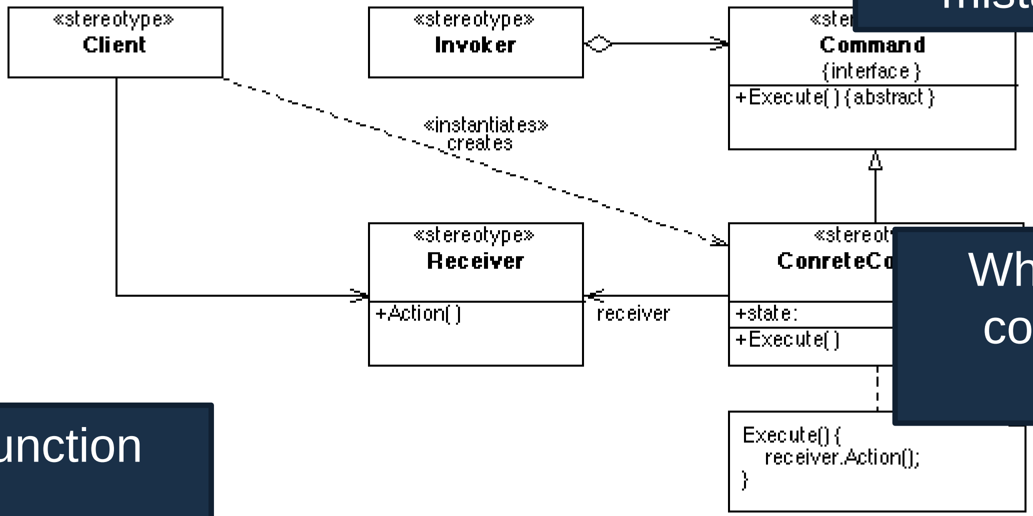
Top 5 game changers

1. Variadic templates
2. Decltype
3. Alias templates
4. Better type traits
5. Improved compilers

A *“useful”* metaprogram*

**may not be useful at all*

The command pattern



Compiler let many mistakes « pass »

Why should the commands be related?

pure virtual function call?

How can I list commands?

The command pattern, revisited

```
struct do_this { void execute(); std::string help() const; };
struct do_that { void execute(); std::string help() const; };

// you could create categories without changing the commands
using commands_list = brigand::list<do_this, do_that>;

// you might want to put them in a variant
using command = brigand::as_variant<commands_list>::value;
```

Execute a command

```
template <typename Command>
void execute(Command & cmd)
{
    static_assert(brigand::contains<commands_list,
        Command>::value,
       "you forgot to add the command in the list");

    cmd.execute();
}
```

Print help of all commands

```
struct help_caller
{
  template <typename U>
  void operator()(const brigand::type_<U> & u)
  {
      u.help();
  }
};

brigand::for_each(commands_list, help_caller);
```

C++ Metaprogramming :

Awesomeness overflow time

Also: typename.

Wouldn't it be cool to...

```
struct my_struct
{
    int alpha;
    std::string omega;
};
```

```
{
  "my_struct" :
  {
      "alpha": 2,
      "omega": "boom"
   }
}
```

```
my_struct blah = { 2, ”boom” };
auto boom = json::generate(json::from_fusion(blah));
```

Introducing Boost.Fusion

```
BOOST_FUSION_ADAPT_STRUCT(my_struct, (int, alpha)(std::string, omega))
```

- **Compile time introspection**
- **Bridge between compile time and runtime**

Example, set of various types:

```
boost::fusion::set<struct1, struct2> blah;
boost::fusion::at_key<struct1>(blah);
```

”Perfect” serialization

Goals

- Fast
  - Generate highly optimized code
  - No ”if mess”
- Frugal
  - Allocate memory only if needed
- Correct
  - Code is generated, not written
- Convenient
  - Unintrusive

“LEFT AS AN EXERCISE TO THE READER”

Conclusion

When and why

- Reliability
- Performance
- “Scalability”

- Needs a modern toolchain
- Time investment
- Complexity trap

Tips to get started

C++ 11	Compiles faster
	Better libraries – like Brigand ☺
	More flexibility
Meta != complex	Start with simple checks – it's safe!
	Move to the next level when you don't event think about it anymore
	Metaprograms must be maintained like normal programs!

The Metaprogrammer toolbox

- **Clang 3.5+**
- **<type_traits>**
- **<tuples>**
- **Boost.MPL**
- **Boost.Fusion**
- **Boost.Hana**
- **Brigand – https://github.com/edouarda/brigand/**

Thank you!

Questions & Answers