Яндекс

Яндекс

Рефлексия настоящего и будущего

Руслан Манаев, разработчик

План

1. Немного про рефлексию
2. Сериализация агрегатов
3. Сериализация Json
4. Сериализация Protobuf
5. Замена GMock

Немного про рефлексию

~~Рефлексия~~ Reflection

Reflection

Reflection

Dynamic reflection
(Runtime)

Reflection

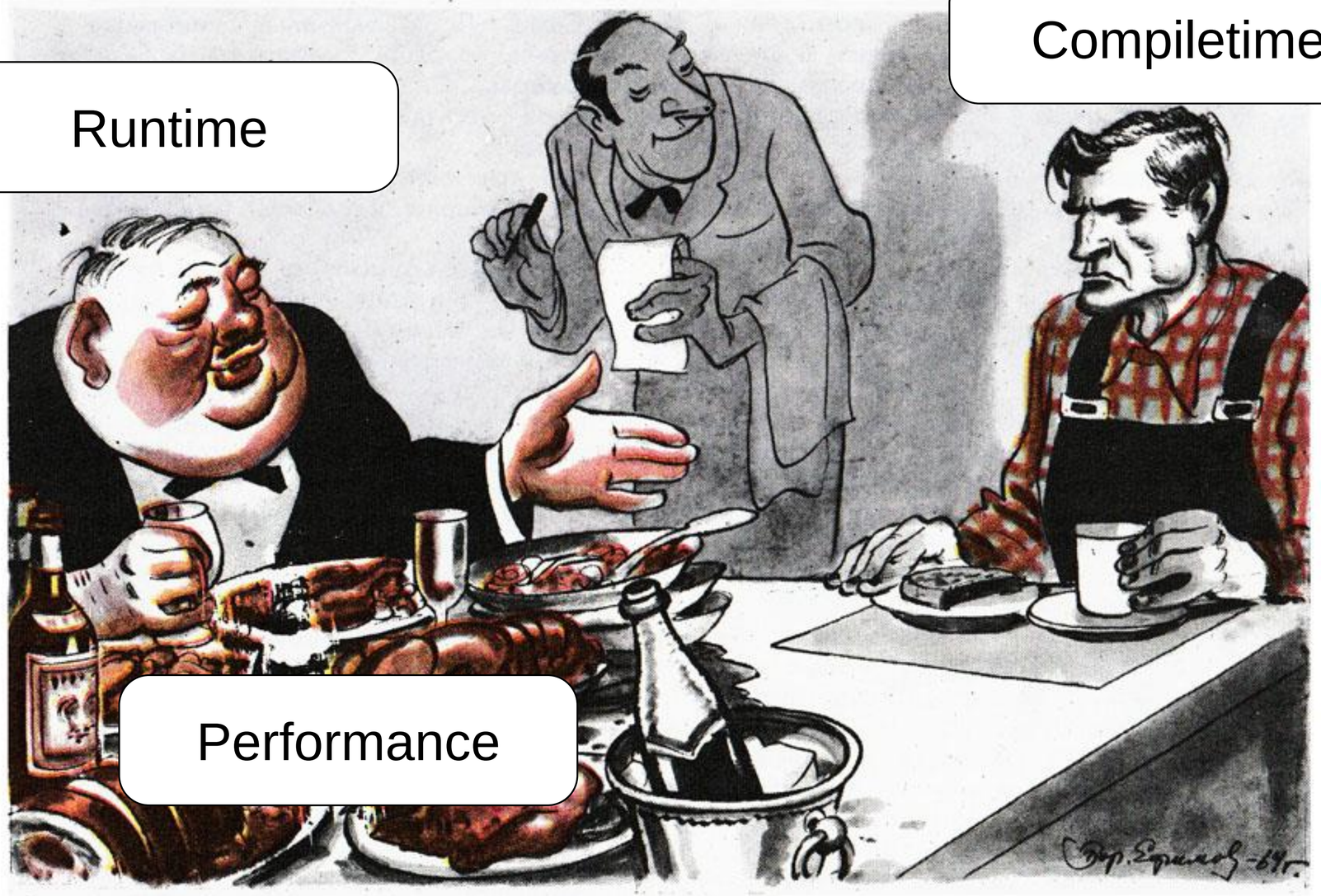
Dynamic reflection
(Runtime)

Static reflection
(Compiletime)

В РЕСТОРАНЕ «СОЮЗ GCC И CLANG »

Compiletime

Runtime



Performance

— Этот господин уплатит за всё!

Рисунок Бор. ЕФИМОВА

C++ ???

Если очень хотим reflection сегодня

Пишем обработчик AST дерева

Модим компилятор

Dodge Greenley

Компилятор, который смог

lock3’s clang

Сериализация агрегатов

Сериализация агрегатов

- Хотим читать и писать в поток
- 10’000 структур
- std::is_aggregate_v<T> || has_io_operator_v<T>

```
struct User {
   std::string name;
   std::string status;
   uint64_t age;
};
```

```
User user = {"Ruslan", "I am okay.", 23, };
std::cout << user;
```

```
User user = {"Ruslan", "I am okay.", 23, };
std::cout << user;
```

```
$> {"Ruslan", "I am okay.", 23, }
```

```
template <typename Char,
          typename Traits,
          typename T>
auto& operator<<(
    std::basic_ostream<Char, Traits>& out,
    const T& value) {
  // code
}
```

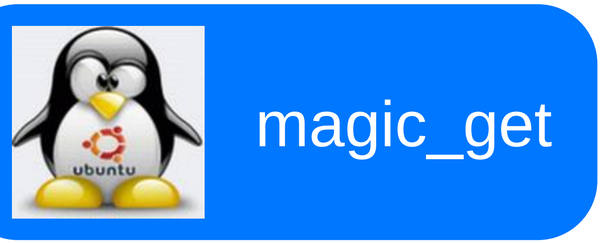
```
auto& operator<<(/*code*/, const T& value){
  out << '{';




}
```

```
auto& operator<<(/*code*/, const T& value){
  out << '{';
  auto tie = magic::tie_as_tuple(value);


}
```



```
auto& operator<<(/*code*/, const T& value){
  out << '{';
  auto tie = magic::tie_as_tuple(value);
  detail::for_each(tie, [&](auto&& arg) {
    out << arg << ", ";
  });


}
```

magic_get

```
auto& operator<<(/*code*/, const T& value){
  out << '{';
  auto tie = magic::tie_as_tuple(value);
  detail::for_each(tie, [&](auto&& arg) {
    out << arg << ", ";
  });
  out << '}';
  return out;
}
```

magic_get

```
template <typename T, typename Operator>
constexpr void for_each(T&& t,
                        Operator&& op) {
  for_each_impl(
    std::forward<T>(t),
    std::forward<Operator>(op),
    std::make_index_sequence<T::size_v>{});
}
```

```
template <typename T,
          typename Operator,
          size_t ... Is>
constexpr void for_each_impl(
              T&& t, Operator&& op,
              std::index_sequence<Is...>) {
(..., op(magic::get<Is>(std::forward<T>(t))));


}
```

```
template <typename T,
          typename Operator>

constexpr void for_each_impl(
                T&& t, Operator&& op,
                std::index_sequence<0,1,2>) {
  op(magic::get<0>(std::forward<T>(t));
  op(magic::get<1>(std::forward<T>(t));
  op(magic::get<2>(std::forward<T>(t));
}
```

WARNING!

```
auto& operator<<(/*code*/, const T& value){
  out << '{';
  auto tie = magic::tie_as_tuple(value);
  detail::for_each(tie, [&](auto&& arg) {
    out << arg << ", ";
  });
  out << '}';
  return out;
}
```

magic_get

```
auto& operator<<(/*code*/, const T& value){
  out << '{';
  out << '}';
  return out;
}
```

```
auto& operator<<(/*code*/, const T& value){
  out << '{';
  detail::for_each_by_members(value,
    [&](auto&& arg) {
      out << arg << ", ";
    });
  out << '}';
  return out;
}
```

```
template<typename T, typename Operator>
void for_each_by_members(T&& t,
                         Operator&& op) {



}
```

```
void for_each_by_members(T&& t, /*code*/) {



}
```

```
void for_each_by_members(T&& t, /*code*/) {
  consteval {



  }
}
```

```
void for_each_by_members(T&& t, /*code*/) {
  consteval {
    meta::info info = reflexpr(T);









  }
}
```

```
void for_each_by_members(T&& t, /*code*/) {
  consteval {
    meta::info info = reflexpr(T);
    auto range =
            meta::data_member_range(info);




  }
}
```

```
void for_each_by_members(T&& t, /*code*/) {
  consteval {
    auto range =
      meta::data_member_range(reflexpr(T));









  }
}
```

```
void for_each_by_members(T&& t, /*code*/) {
  consteval {
    auto range =
      meta::data_member_range(reflexpr(T))|
      meta::is_nonstatic_data_member;









  }
}
```

```
void for_each_by_members(T&& t, /*code*/) {
  consteval {
    auto range =
      meta::data_member_range(reflexpr(T))|
      meta::is_nonstatic_data_member;
    for (meta::info member : range)


        ;
  }
}
```

```
void for_each_by_members(T&& t, /*code*/) {
  consteval {
    auto range =
      meta::data_member_range(reflexpr(T))|
      meta::is_nonstatic_data_member;
    for (meta::info member : range)
      -> __fragment {

      };
  }
}
```

```
void for_each_by_members(T&& t, /*code*/) {
  consteval {
    auto range =
      meta::data_member_range(reflexpr(T))|
      meta::is_nonstatic_data_member;
    for (meta::info member : range)
      -> __fragment {
        op(t.*valueof(member));
      };
  }
}
```

```
void for_each_by_members(T&& t, /*code*/) {
  consteval {
    auto range =
      meta::data_member_range(reflexpr(T))|
      meta::is_nonstatic_data_member;
    for (meta::info member : range)
      -> __fragment {
        op(t.*valueof(member));
      };
  }
}
```

```
void for_each_by_members(T&& t, /*code*/) {

  op(t.*(&User::name));
  op(t.*(&User::status));
  op(t.*(&User::age));

}
```

```
constexpr auto fragment = __fragment {
   // code
};

consteval {
   -> fragment;
}
// equivalent:
consteval -> fragment;
```

Вопросы?

Сериализация Json

Сериализация агрегатов

- Хотим кастить в Json и обратно
- Не хотим руками писать каждый раз преобразование
- Хотим использовать библиотеку “JSON for Modern C++” от Niels Lohmann

```
User user = {"Ruslan", "I am okay.", 23, };
    std::cout << nlohmann::json(user);
```

```
$> [{"name":"Ruslan","status":
"I am okay.","age":23}]
```

```
using nlohmann::json;


void to_json(json& j, const User& u) {


}
```

```
using nlohmann::json;



void to_json(json& j, const User& u) {
    j["name"] = u.name;
    j["status"] = u.status;
    j["age"] = u.age;
}
```

```
using nlohmann::json;


template <typename T>
void to_json(json& j, const T& t) {
  detail::for_each_by_members(t,
    [&](auto&& arg, auto name) {
      j[name] = arg;
    });
}
```

```
using nlohmann::json;


void from_json(const json& j, User& u) {


}
```

```
using nlohmann::json;



void from_json(const json& j, User& u) {
    j.at("name").get_to(u.name);
    j.at("status").get_to(u.status);
    j.at("age").get_to(u.age);
}
```

```
using nlohmann::json;


template <typename T>
void from_json(const json& j, T& t) {
  detail::for_each_by_members(t,
    [&](auto&& arg, auto name) {
      j.at(name).get_to(arg);
    });
}
```

```
void for_each_by_members(T&& t, /*code*/) {
  consteval {
    /* code */
    for (meta::info member : range)
      -> __fragment {
        op(t.*valueof(member));
      };
  }


}
```

```
void for_each_by_members(T&& t, /*code*/) {
  consteval {
    /* code */
    for (meta::info member : range)
      -> __fragment {
        op(t.*valueof(member),
           meta::name_of(member));
      };
  }

}
```

```
void for_each_by_members(T&& t, /*code*/) {

    op(t.*(&User::name), "name");
    op(t.*(&User::status), "status");
    op(t.*(&User::age), "age");

}
```

Вопросы?

Сериализация Protobuf

Немного про Protobuf

```
// user.proto file
message User {
  string name = 1;
  string status = 2;
  uint64 age = 3;
};
```

Немного про Protobuf

```
message User {
   string name = 1;
   string status = 2;
   uint64 age = 3;
};
```

```
$> name: "Ruslan"
status: "I am okay."
age: 23
```

Немного про Protobuf

Немного про Protobuf

protoc

libprotobuf

Немного про Protobuf

protoc

libprotobuf

Сериализация Protobuf

- Хотим сериализовать и десериализовать в protobuf
- Хотим поддерживать оригинальный API
- Хотим поддерживать C++ структуры

```
struct User {
   std::string name;
   std::string status;
   uint64_t age;
};
```

```
struct(proto) User {
  std::string name;
  std::string status;
  uint64_t age;
};
```

```
struct(proto) User {
  [[id(1)]] std::string name;
  [[id(2)]] std::string status;
  [[id(3)]] uint64_t age;
};
```

Атрибуты хороши, но не работают пока

```
User user;
user.set_name("Ruslan");
user.set_status("I am okay.");
user.set_age(23);

Std::cout << user.DebugString();
```

```
$> name: "Ruslan"
status: "I am okay."
age: 23
```

```
consteval void proto(meta::info info) {

}
```

```
struct User {

};
```

```
consteval void proto(meta::info info) {
  -> __fragment struct {
  };
}
```

```
struct User {
};
```

```
consteval void proto(meta::info info) {

  -> __fragment
  struct {
  };

}
```

```
struct User {

};
```

```
consteval void proto(meta::info info) {

  -> __fragment
  struct {
    consteval {


    }
  };









}
```

```
struct User {
  consteval {


  }


};
```

```
consteval void proto(meta::info info) {
  using protobuf::Message;
  -> __fragment
  struct {
    consteval {
      __inject_base(public Message);
    }
  };









}
```

```
struct User {
  consteval {
    __inject_base(
        public Message);
  }









};
```

```
consteval void proto(meta::info info) {
  using protobuf::Message;
  -> __fragment
  struct {
    consteval {
      __inject_base(public Message);
    }
  };




}
```

```
struct User :
public Message {




};
```

```
consteval void proto(meta::info info) {
  /* collapsed code */
  -> __fragment
  struct {
  };
}
```

```
struct User :
public Message {

};
```

```
consteval void proto(meta::info info) {
  /* collapsed code */
  -> __fragment
  struct T {
  };
}
```

```
struct User :
public Message {

};
```

```
consteval void proto(meta::info info) {
  /* collapsed code */
  -> __fragment
  struct T : public Message {
  };
}
```

```
struct User :
public Message {

};
```

```
consteval void proto(meta::info info) {
  /* collapsed code */
  -> __fragment
  struct T : public Message {
    consteval {
      auto r = meta::member_range(info))
        | meta::is_nonstatic_data_member;
      for (auto member : r) {


      }
    }
  };



}
```

```
struct User :
public Message {









};
```

```
consteval void proto(meta::info info) {
  /* collapsed code */
  -> __fragment
  struct T : public Message {
    consteval {
      auto r = meta::member_range(info))
        | meta::is_nonstatic_data_member;
      for (auto member : r) {
        inject(member);
      }
    }
  };


}
```

```
struct User :
public Message {
 private:
  std::string name_;
 public:
  void set_name(
       const std::string& arg) {
    name_ = arg;
  }


   const auto& name() const {
    return name_;
   }
   /* etc */


};
```

```
consteval void proto(meta::info info) {
  /* collapsed code */
  -> __fragment
  struct T : public Message {
    consteval {
      auto r = meta::member_range(info))
        | meta::is_nonstatic_data_member;
      for (auto member : r) {
        inject(member);
      }
    }
  };


}
```

```
struct User :
public Message {
 private:
  std::string name_;
 public:
 void set_name(
      const std::string& arg) {
   name_ = arg;
 }


  const auto& name() const {
   return name_;
 }
  /* etc */


};
```

```
consteval void inject(meta::info member){
}
```

```
struct User :
public Message {
};
```

```
consteval void inject(meta::info member){
  auto name = meta::name_of(member);
  auto field_name = __concat(name, "_");
  auto set_name = __concat("set_", name);
  auto get_name = name;









}
```

```
struct User :
public Message {









};
```

```
consteval void inject(meta::info member){
  auto name = meta::name_of(member);
  auto field_name = __concat(name, "_");
  auto set_name = __concat("set_", name);
  auto get_name = name;









  -> member;









}
```

```
struct User :
public Message {

  std::string name;
  std::string status;
  uint64_t age;









};
```

```
consteval void inject(meta::info member){
  auto name = meta::name_of(member);
  auto field_name = __concat(name, "_");
  auto set_name = __concat("set_", name);
  auto get_name = name;

  meta::set_new_name(member,
                     field_name);

  -> member;




}
```

```
struct User :
public Message {

  std::string name_;
  std::string status_;
  uint64_t age_;









};
```

```
consteval void inject(meta::info member){
  auto name = meta::name_of(member);
  auto field_name = __concat(name, "_");
  auto set_name = __concat("set_", name);
  auto get_name = name;

  meta::set_new_name(member,
                     field_name);
  meta::make_private(member);
  -> member;




}
```

```
struct User :
public Message {
 private:
  std::string name_;
  std::string status_;
  uint64_t age_;









};
```

```
consteval void inject(meta::info member){
  /* collapsed code */
}
```

```
struct User :
public Message {
 private:
  std::string name_;
  std::string status_;
  uint64_t age_;
};
```

```
consteval void inject(meta::info member){
  /* collapsed code */
  meta::info type = meta::type_of(member);
}
```

```
struct User :
public Message {
 private:
  std::string name_;
  std::string status_;
  uint64_t age_;
};
```

```
consteval void inject(meta::info member){
  /* collapsed code */
  meta::info type = meta::type_of(member);

  -> __fragment
  struct {
    void unqualid(set_name)(
                const typename(type)& arg) {
      unqualid(field_name) = arg;
    }
  };



}
```

```
struct User :
public Message {
 /* collapsed code */
public:
  void set_name(
       const std::string& arg) {
    name_ = arg;
  }
  void set_status(
       const std::string& arg) {
    status_ = arg;
  }
  void set_age(
       const uint64_t& arg) {
    age_ = arg;
  }


};
```

```
consteval void inject(meta::info member){
  /* collapsed code */

}
```

```
struct User :
public Message {
 /* collapsed code */

};
```

```
consteval void inject(meta::info member){
  /* collapsed code */

  -> __fragment
  struct {
    const auto& unqualid(get_name)()
                                    const {
      return unqualid(field_name);
    }
  };




}
```

```
struct User :
public Message {
 /* collapsed code */
public:
  const auto& get_name() {
    return name_;
  }
  const auto& get_status() {
    return status_;
  }
  const auto& get_age() {
    return age_;
  }




};
```

```
consteval void proto(meta::info info) {
  /* collapsed code */
}
```

```
struct User :
public Message {
  /* collapsed code */
};
```

```
consteval void proto(meta::info info) {
  /* collapsed code */
  using Metadata = protobuf::internal::
                   InternalMetadataWithArena;
  using Size = protobuf::internal::
                                CachedSize;
  -> __fragment
  struct T : public Message {
    Metadata _meta_;
    mutable Size _size_;
  };



}
```

```
struct User :
public Message {
  /* collapsed code */
  Metadata _meta_;
  mutable Size _size_;



};
```

```
consteval void proto(meta::info info) {
  /* collapsed code */
  -> __fragment
  struct T : public Message {
    T() : Message(), _meta_() {
      Clear();
    }
    T(const T& from) : _meta_() {
      _meta_.MergeFrom(from._meta_);
      CopyFrom(from);
    }
  };


}
```

```
struct User :
public Message {
 /* collapsed code */
 User() : Message(),
          _meta_() {
   Clear();
 }
 User(const User& from)
          : _meta_() {
   _meta_.MergeFrom(
          from._meta_);
   CopyFrom(from);
 }


};
```

```
consteval void proto(meta::info info) {
  /* collapsed code */
  -> __fragment
  struct T : public Message {
    void Clear() final {
      for_each_proto(*this,
                     [](auto&& member) {
        member = {};
      });
    }
  };



}
```

```
struct User :
public Message {
  /* collapsed code */

 void Clear() final {
   name_ = {};
   status_ = {};
   age_ = {};
 }



};
```

```
void for_each_proto(T&& t, /*code*/) {
  consteval {
    auto range =
      meta::data_member_range(reflexpr(T))|
      meta::is_nonstatic_data_member;
    for (meta::info member : range)
      -> __fragment {
        op(t.*valueof(member));
      };
  }
}
```

```
void for_each_proto(T&& t, /*code*/) {
  consteval {
    auto range =
      meta::data_member_range(reflexpr(T))|
meta::is_nonstatic_data_member | is_proto;
    for (meta::info member : range)
      -> __fragment {
        op(t.*valueof(member));
      };
  }
}
```

```
consteval void proto(meta::info info) {
  /* collapsed code */
  -> __fragment
  struct T : public Message {
    void MergeFrom(const T& from) {
      _meta_.MergeFrom(from._meta_);
      for_each_pairwise(
          *this, from,
          [](auto&& to, auto&& from) {
            to = from;
          });
    }
  };


}
```

```
struct User :
public Message {
  /* collapsed code */

 void MergeFrom(
     const User& from) {
   name_ = from.name_;
   status_ =
       from.status _;
   age_ = from.age_;
 }



};
```

```
void for_each_pairwise(T&& t1, T&& t2/**/){
  consteval {
    auto range =
      meta::data_member_range(reflexpr(T))|
meta::is_nonstatic_data_member | is_proto;
    for (meta::info m : range)
      -> __fragment {
        op(t.*valueof(m));
      };
  }
}
```

```
void for_each_pairwise(T&& t1, T&& t2/**/){
  consteval {
    auto range =
      meta::data_member_range(reflexpr(T))|
meta::is_nonstatic_data_member | is_proto;
    for (meta::info m : range)
      -> __fragment {
        op(t.*valueof(m), t.*valueof(m));
      };
  }
}
```

```
consteval void proto(meta::info info) {
  /* collapsed code */
  -> __fragment
  struct T : public Message {
    using protobuf::Metadata;
    Metadata GetMetadata() const final {
      return GetMetadataStatic();
    }
    static Metadata GetMetadataStatic() {
      /* code */
    }
  };



}
```

```
struct User :
public Message {
  /* collapsed code */
  Metadata GetMetadata()
                  const final {
    return GetMetadataStatic();
  }

  static Metadata
         GetMetadataStatic() {
    /* code */
  }




};
```

```
static Metadata GetMetadataStatic() {

}
```

```
static Metadata GetMetadataStatic() {
  /* collapsed code */
  static const DescriptorTable dt = {
    /* collapsed code */
    .descriptor = static_dt.c_str(),
    /* collapsed code */
    .offsets = offsets.data(),
    /* collapsed code */
  };
  /* collapsed code */
}
```

```
static Metadata GetMetadataStatic() {
  /* collapsed code */
  static const DescriptorTable dt = {
    /* collapsed code */
    .descriptor = static_dt.c_str(),
    /* collapsed code */
    .offsets = offsets.data(),
    /* collapsed code */
  };
  /* collapsed code */
}
```

```
static std::string static_dt;
```

```
static std::string static_dt;
constexpr auto message_name =
  meta::name_of(meta::definition_of(info));
constexpr auto file_name =
          __concat(message_name, ".proto");
```

```
static std::string static_dt;
/* collapsed code */
if (static_descriptor_table.empty()) {


}
```

```
static std::string static_dt;
/* collapsed code */
if (static_descriptor_table.empty()) {
  protobuf::FileDescriptorProto file_proto;
}
```

```
static std::string static_dt;
/* collapsed code */
if (static_descriptor_table.empty()) {
  protobuf::FileDescriptorProto file_proto;
  file_proto.set_syntax("proto3");
  file_proto.set_name(file_name.data());
  file_proto.set_package("cpp_generated");



}
```

```
static std::string static_dt;
/* collapsed code */
if (static_descriptor_table.empty()) {
  protobuf::FileDescriptorProto file_proto;
  file_proto.set_syntax("proto3");
  file_proto.set_name(file_name.data());
  file_proto.set_package("cpp_generated");
  auto& message_descriptor =
                file_proto.add_message_type();

}
```

```
static std::string static_dt;
/* collapsed code */
auto& message_descriptor =
                  file_proto.add_message_type();
```

```
static std::string static_dt;
/* collapsed code */
auto& message_descriptor =
                  file_proto.add_message_type();
for_each_by_proto_info<T>(
  [&](auto name, auto number, auto type) {



  });
```

```
static std::string static_dt;
/* collapsed code */
auto& message_descriptor =
                  file_proto.add_message_type();
for_each_by_proto_info<T>(
  [&](auto name, auto number, auto type) {
    auto& field_descriptor =
                *message_descriptor.add_field();
  });
```

```
static std::string static_dt;
/* collapsed code */
  [&](auto name, auto number, auto type) {
    auto& field_descriptor =
            *message_descriptor.add_field();
  });
```

```
static std::string static_dt;
/* collapsed code */
  [&](auto name, auto number, auto type) {
    auto& field_descriptor =
            *message_descriptor.add_field();
    field_descriptor.set_name(name);
    field_descriptor.set_number(number);
    field_descriptor.set_type(type);
    field_descriptor.set_type(OPTIONAL);


  });
```

```
static std::string static_dt;
/* collapsed code */
file_proto.SerializeToString(&static_dt);
```

for_each_by_proto_info

```
static Metadata GetMetadataStatic() {
  /* collapsed code */
  static const DescriptorTable dt = {
    /* collapsed code */
    .descriptor = static_dt.c_str(),
    /* collapsed code */
    .offsets = offsets.data(),
    /* collapsed code */
  };
  /* collapsed code */
}
```

```
static Metadata GetMetadataStatic() {
  /* collapsed code */
  static const DescriptorTable dt = {
    /* collapsed code */
    .descriptor = static_dt.c_str(),
    /* collapsed code */
    .offsets = offsets.data(),
    /* collapsed code */
  };
  /* collapsed code */
}
```

```
static std::vector<uint32_t> offsets;
```

```
static std::vector<uint32_t> offsets;
if (offsets.empty()) {



}
```

```
static std::vector<uint32_t> offsets;
if (offsets.empty()) {
  offsets.emplace_back(~0u);
  offsets.emplace_back(
          PROTOBUF_FIELD_OFFSET(T, _meta_));
  offsets.emplace_back(~0u);
  offsets.emplace_back(~0u);
  offsets.emplace_back(~0u);


}
```

```
static std::vector<uint32_t> offsets;
if (offsets.empty()) {
  offsets.emplace_back(~0u);
  offsets.emplace_back(
          PROTOBUF_FIELD_OFFSET(T, _meta_));
  offsets.emplace_back(~0u);
  offsets.emplace_back(~0u);
  offsets.emplace_back(~0u);
  fill_offsets<T>(offsets);

}
```

```
void fill_offsets<T>(std::vector& offsets){
  consteval {
    auto range =
      meta::data_member_range(reflexpr(T))|
      meta::is_nonstatic_data_member;
    for (meta::info m : range)
      -> __fragment {

      };
  }
}
```

```
void fill_offsets<T>(std::vector& offsets){
  consteval {
    auto range =
      meta::data_member_range(reflexpr(T))|
      meta::is_nonstatic_data_member;
    for (meta::info m : range)
      -> __fragment {
  offsets.push_back(offset_of(valueof(m)));
      };
  }
}
```

```
template <typename T, typename U>
size_t offset_of(U T::*member) {
  return (char*)&((T*)nullptr->*member) -
         (char*)nullptr;
}
```

```
User user;
user.set_name("Ruslan");
user.set_status("I am okay.");
user.set_age(23);

Std::cout << user.DebugString();
```

```
$> name: "Ruslan"
status: "I am okay."
age: 23
```

source: https://knowyourmeme.com/memes/blinking-white-guy

Мы только что получили рабочий dynamic reflection для полей протобафа воспользовавшись static reflection’ом и metaclass’ми

Вопросы?

Замена GMock

Немного про GMock

```
struct Tab {
  void Init();
};
```

```
struct Browser {
 void AddTab(Tab* t);
};
```

Немного про GMock

```
struct Tab {
  void Init();
};
```

```
struct Browser {
 void AddTab(Tab* t);
};

TEST(Test, Test) {
 Browser browser;
 Tab tab;


 browser.AddTab(&tab);
}
```

Немного про GMock

```
struct ITab {
  virtual void Init() = 0;
};

struct Tab : public ITab {
  void Init() final;
};
```

```
struct Browser {
 void AddTab(ITab* t);
};

TEST(Test, Test) {
 Browser browser;
 Tab tab;


 browser.AddTab(&tab);
}
```

Немного про GMock

```
struct ITab {
  virtual void Init() = 0;
};

struct Tab : public ITab {
  void Init() final;
};

struct MockTab : public ITab

};
```

```
struct Browser {
 void AddTab(ITab* t);
};

TEST(Test, Test) {
 Browser browser;
 MockTab tab;


 browser.AddTab(&tab);
}
```

Немного про GMock

```
struct ITab {
  virtual void Init() = 0;
};

struct Tab : public ITab {
  void Init() final;
};

struct MockTab : public ITab
  MOCK_METHOD(void, Init,
              (), (override));
};
```

```
struct Browser {
 void AddTab(ITab* t);
};

TEST(Test, Test) {
 Browser browser;
 MockTab tab;


 browser.AddTab(&tab);
}
```

Немного про GMock

```
struct ITab {
  virtual void Init() = 0;
};

struct Tab : public ITab {
  void Init() final;
};

struct MockTab : public ITab
  MOCK_METHOD(void, Init,
              (), (override));
};
```

```
struct Browser {
 void AddTab(ITab* t);
};

TEST(Test, Test) {
 Browser browser;
 MockTab tab;
 EXPECT_CALL(tab, Init())
   .Times(AtLeast(1));
 browser.AddTab(&tab);
}
```

Замена GMock

- Хотим избавиться от виртуальности и указателей
- Хотим избавиться от макросов
- Хотим упростить код

```
struct ITab {
  virtual void Init() = 0;
};

struct Tab : public ITab {
  void Init() final;
};

struct MockTab : public ITab
  MOCK_METHOD(void, Init,
              (), (override));
};
```

```
struct Browser {
 void AddTab(ITab* t);
};

TEST(Test, Test) {
 Browser browser;
 MockTab tab;
 EXPECT_CALL(tab, Init())
   .Times(AtLeast(1));
 browser.AddTab(&tab);
}
```

```
struct Tab {
  void Init();
};
```

```
struct Browser {
 void AddTab(ITab* t);
};

TEST(Test, Test) {
 Browser browser;
 MockTab tab;
 EXPECT_CALL(tab, Init())
   .Times(AtLeast(1));
 browser.AddTab(&tab);
}
```

```
struct Tab {
  void Init();
};
```

```
struct Browser {
 void AddTab(Tab t);
};

TEST(Test, Test) {
 Browser browser;
 MockTab tab;
 EXPECT_CALL(tab, Init())
   .Times(AtLeast(1));
 browser.AddTab(&tab);
}
```

```
struct Tab {
  void Init();
};
```

```
struct Browser {
 Tab& AddTab(Tab t);
};

TEST(Test, Test) {
 Browser browser;
 MockTab tab;
 EXPECT_CALL(tab, Init())
   .Times(AtLeast(1));
 browser.AddTab(&tab);
}
```

```
struct(mockable) Tab {
  void Init();
};
```

```
struct Browser {
 Tab& AddTab(Tab t);
};

TEST(Test, Test) {
 Browser browser;
 Tab tab;
 EXPECT_CALL(tab, Init())
   .Times(AtLeast(1));
 browser.AddTab(&tab);
}
```

```
struct(mockable) Tab {
  void Init();
};
```

```
struct Browser {
 Tab& AddTab(Tab t);
};

TEST(Test, Test) {
 Browser browser;
 Tab tab(InitAsMock);
 tab.ExpectCall(&Tab::Init
   .Times(AtLeast(1));
 browser.AddTab(&tab);
}
```

```
consteval void mockable(meta::info info){
  auto range = meta::member_range(info);



  for (meta::info member : ) {
  }
```

```
struct Tab {









};
```

TODO

Яндекс

Спасибо за внимание!

Руслан Манаев

разработчик

manavrion@yandex-team.ru

manavrion