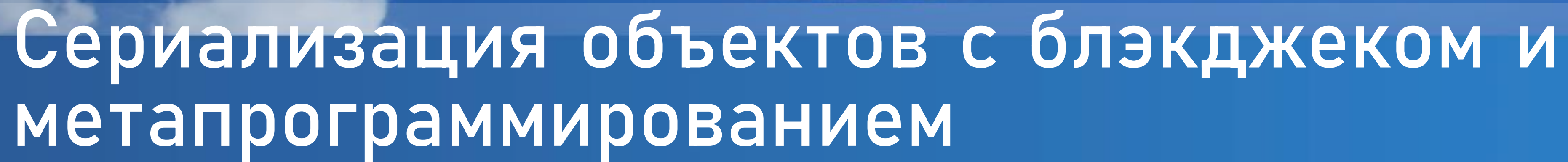


Сериализация объектов с блэкджеком и метапрограммированием

Александр Ганюхин <alexander.ganyukhin@mera.com>

Автомобильная индустрия

Бытовая электроника

Здравоохранение и медицина

Интернет вещей

Ритейл и торговля

Телекоммуникации и связь

Финансовая индустрия

Энергетика

Setup

Setup

Все имена и фамилии изменены, а все совпадения - случайны

- **С рассматриваемой задачей столкнулись в телекоме несколько лет назад**
- **Рассматриваемое решение появилось уже после окончания проекта телекома**
- **Для упрощения мы будем все рассматривать на примере казино**
- **Код, показанный на слайдах – собирательный образ,
но источник – реальный проект**
- **Все имена классов и переменных изменены, а все совпадения - случайны**

Setup

Представьте себе: вы – владелец казино.

```
class Casino {
    /* ... */
    void getBlackJackDealer(BlackJackDealer * & b);
    void getTexasHoldemDealer(TexasHoldemDealer & b);
    FiveCardDrawDealer * getFiveCardDrawDealer();
    RouletteDealer const & getRouletteDealer() const;
    /* ... */
};
```

Setup

Вам нужны детальные отчеты

```
class Casino {
    /* ... */
};
```

```
makeReport(casino);

// result.xml
<some_report id="1" …  />
<some_report id="2" …  />
<some_report id="3" …  />
// …
```

Setup

Давайте рассмотрим какой-нибудь конкретный геттер

```
class Casino {
    void getBlackJackDealer(BlackJackDealer * & b);
    /* ... */
};
struct BlackJackDealer {
    Card * getNextCard();
};
struct Card {
    int getValue();
};
```

Setup

Получаем очередную карту

```
   void printCardValue(Casino & casino) {
1      BlackJackDealer * blackJackDealer;
2      casino.getBlackJackDealer(blackJackDealer);

3      if (blackJackDealer) {
4          Card * card =  blackJackDealer->getNextCard();
5          if (card) {
6              cout << card->getValue();
7          }
8      }
   }
```

Setup

Проблемы?

- **Если для 1 геттера с 3-м уровнем вложенности мы написали 8 строчек, то:**
  - Для 100 геттеров необходимо 800 строк
  - Для 200 – 1600
  - Для 300 – 2400
  - И так далее...
- **А теперь подумаем:**
  - Солько ошибок можно сделать в этих 800 строчках?
  - Легко ли поддерживать такой объем кода?
  - А что, если изменится сигнатура?
  - Сможете ли вы увидеть «цельную» картинку за таким объемом кода?

Чего бы хотелось

```
constexpr serializer<Casino, 3> toc{
    makeSerializer<&Casino::getBlackJackDealer,
                                  &BlackJackDealer::getCard,
                                      &Card::getValue>("call1"),
    makeSerializer<&Casino::getBlackJackDealer,
                                  &BlackJackDealer::getCard,
                                      &Card::getValue>("call2"),
    makeSerializer<&Casino::getBlackJackDealer,
                                  &BlackJackDealer::getCard,
                                      &Card::getValue>("call3"),
 };
toc(casino, serializationInterface);

// result.xml

<card id="call1" value="10" />

<card id="call2" value="3"  />

<card id="call3" value=“BlackJack!" />
```

План действий

План действий

Или то, что нам пока не понятно

- **Как вызвать функцию по указателю, не передавая аргументы?**
- **Как вызвать «вложенные» геттеры?**
- **Как создать «оглавление»?**

Вызываем функцию по фотографии

Начнем с малого

Давайте возьмем у дилера две карты и покажем их на экране

```
struct BlackJackDealer {
    void getTwoCards(Card * a, Card & b);
};

void printCards(BlackJackDealer & bj) {
    Card a { };
    Card b { };

    bj.getTwoCards(&a, b);

    cout << "Cards: " << a << b;
}
```

Указатель на функцию

Давайте добавим параметр – указатель на функцию

```
void printCards(BlackJackDealer & bj) {
    Card a { };                    /* 1 */
    Card b { };                    /* 1 */
    bj.getTwoCards(&a, b);         /* 2 */
    cout << a << b;                /* 3 */
}
```

Указатель на функцию

Давайте добавим параметр – указатель на функцию

```
void printCards(BlackJackDealer & bj) {
    Card a { };                    /* 1 */
    Card b { };                    /* 1 */
    bj.getTwoCards(&a, b);         /* 2 */
    cout << a << b;                /* 3 */
}
```

Указатель на функцию

Давайте добавим параметр – указатель на функцию

```
using Fx = void(BlackJackDealer::*)(Card *, Card &);
void printCards(BlackJackDealer & bj, Fx const & fx) {
    Card a { };                    /* 1 */
    Card b { };                    /* 1 */
    (bj.*fx)(&a, b);               /* 2 */
    cout << a << b;                /* 3 */
}

struct BlackJackDealer {
    void getTwoCards2(Card & a, Card & b);
};
// printCards(obj, &BlackJackDealer::getTwoCards2); // ERROR
```

Шаблонный указатель на функцию

```
template<typename Ret, typename Class, typename ... Args>
void printCards(Class & obj, Ret(Class::*ptr)(Args...));

struct BlackJackDealer {
    void getTwoCards(Card* a, Card& b) const;
    void getTwoCards2(Card& a, Card& b) const volatile &;
};
//printCards(obj, &BlackJack::getTwoCards);  // ERROR
//printCards(obj, &BlackJack::getTwoCards2); // ERROR

cout << typeid(&BlackJackDealer::getTwoCards2).name();
// void(BlackJackDealer::*)(Card &, Card &) const volatile &
```

Поступим более универсально

```
template<typename TObj, typename Fx>
void print(TObj & obj, Fx const & fx);
```

- Проблемы?
  - Как FX превратить в `Return(Class::*)(Args...)`?
  - Как создать пак аргументов на стэке?
  - Как передать эти аргументы в функцию?
  - Как вывести на экран, кстати не забыв провалидировать, например, указатели?

Превращаем Fx

Частичная специализация шаблонов

```
template<typename T>
struct function_info;
template<typename R, typename Cl, typename ... Args>
struct function_info<R(Cl::*)(Args...)> {
    using ret       = R;
    using cl        = Cl;
    using args      = tuple<Args...>;
};
/* ... */
template<typename R, typename Cl, typename ... Args>
struct function_info<R(Cl::*)(Args...) const volatile &>
{ /* ... */ };
```

Превращаем Fx

Вспомним, как мы вызывали функцию

```
struct BlackJackDealer {
    void getTwoCards(Card * a, Card & b);
};
void printCards(BlackJackDealer & bj) {
    Card a { };
    Card b { };
    bj.getTwoCards(&a, b);
}

struct function_info<R(Cl::*)(Args...)> {
    using args         = tuple<Args...>;
                       /* tuple<Card *, Card &> */
};
```

Превращаем Fx

Давайте пофиксим?

```
template<typename R, typename Cl, typename ... Args>
struct function_info<R(Cl::*)(Args...) const> {
    using stack_args = tuple<
                         remove_reference<
                           remove_pointer<Args>
                         >...>;
};
```

Превращаем Fx

Вот, что получилось

```
template<typename T>
struct function_info;
template<typename R, typename Cl, typename ... Args>
struct function_info<R(Cl::*)(Args...)> {
    using ret        = R;
    using cl         = Cl;
    using args       = tuple<Args...>;
    using stack_args = tuple<
                         remove_reference<
                           remove_pointer<Args>
                         >...>;
};
/* ... */
```

Применим на практике

Было

```
using Fx = void(BlackJackDealer::*)(Card *, Card &);
void printCards(BlackJackDealer & bj, Fx const & fx) {
    Card a { };                     /* 1 */
    Card b { };                     /* 1 */
    (bj.*fx)(&a, b);                /* 2 */
    cout << a << b;                 /* 3 */
}
```

Применим на практике

Было

```
using Fx = void(BlackJackDealer::*)(Card *, Card &);
void printCards(BlackJackDealer & bj, Fx const & fx) {
    Card a { };                 /* 1 */
    Card b { };                 /* 1 */
    (bj.*fx)(&a, b);            /* 2 */
    cout << a << b;             /* 3 */
}
```

Применим на практике

Стало

```
template<typename TObj, typename Fx>
void print(TObj & obj, Fx const & fx) {
    using FI     = function_info<decay_t<Fx>>;
    using Args_t = typename FI::stack_args;
    Args_t tuple{};
}

struct BlackJackDealer {
    void getTwoCards2(Card * a, Card & b) const;
    void getTwoCards2(Card & a, Card & b) &;
};
// print(obj, &BlackJackDealer::getTwoCards);  // OK!
// print(obj, &BlackJackDealer::getTwoCards2); // OK!
```

Вызываем функцию

Но пока без метапрограммирования

```
struct BlackJackDealer {
    void getTwoCards(Card * a, Card & b);
};
template<typename TObj, typename Fx>
void print(TObj & obj, Fx const & fx) {
    tuple<Card, Card> args { };
    (obj.*fx)(&get<0>(args), get<1>(args));
}
```

Пока без метапрограммирования

Все внимание на вызов функции

```
struct Dealer {
    void getThreeCards(Card &, Card &, Card &);
};

void print(Dealer & d) {
    tuple<Card, Card, Card> t { };

    using namespace std;
    d.getThreeCards(get<0>(t), get<1>(t), get<2>(t));
}
```

Добавим метапрограммирование

Все внимание на вызов функции

```
struct Dealer {
    void getThreeCards(Card &, Card &, Card &);
};

void print(Dealer & d) {
    tuple<Card, Card, Card> args { };

    using namespace std;
    d.getThreeCards(get<0>(args), get<1>(args), get<2>(args));

    d.getThreeCards(get<I>(args)...); // I = { 0, 1, 2 }
}
```

Создаем набор индексов

```
template<size_t ... Ints>
struct index_sequence {};

template<typename T, size_t Cnt>
struct make_is_impl;

template<size_t ... Ints, size_t Cnt>
struct make_is_impl<index_sequence<Ints...>, Cnt>
    : make_is_impl<index_sequence<Cnt, Ints...>, Cnt - 1>
{};

template<size_t ... Ints>
struct make_is_impl<index_sequence<Ints...>, 0> {
    using type = index_sequence<0, Ints...>;
};
```

Создаем набор индексов

И сделаем красивый интерфейс

```
template<typename T, size_t Cnt>
struct make_is_impl;
template<size_t ... Ints, size_t Cnt>
struct make_is_impl<index_sequence<Ints...>, Cnt>
    : make_is_impl<index_sequence<Cnt, Ints...>, Cnt - 1>{};
template<size_t ... Ints>
struct make_is_impl<index_sequence<Ints...>, 0> {
    using type = index_sequence<0, Ints...>;
};
template<size_t I>
struct make_index_sequence_
    : public make_is_impl<index_sequence<>, I - 1>
{};
```

Создаем набор индексов

Разберём на практике

```
template<size_t...I, size_t C>
struct make_impl<seq<I...>, C>
    : make_impl<seq<C, I...>, C - 1>
{};
template<size_t ... I>
struct make_impl<seq<I...>, 0> {
    using type = seq<0, I...>;
};
template<size_t I>
struct make_seq_
    : public make_impl<seq<>, I - 1>
{};
```

```
make_seq_<4>
: make_impl<seq<>, 3>
: make_impl<seq<3>, 2>
: make_impl<seq<2,3>, 1>
: make_impl<seq<1,2,3>, 0> {
    type = seq<0, 1, 2, 3>;
}
```

Создаем набор индексов

Разберем на практике make_seq_<500>

- **Открываем Visual Studio**
- **Пишем “**`using T = typename make_seq_<500>::type;`**”**
- **Компилируем**
- **Ииии…**
- **error C1202: recursive type or function dependency context too complex**

Создаем набор индексов

```
using T = make_index_sequence<1000000>;
```

- **Прекрасно компилируется везде, кроме VS.**
- **VS не поддерживает 1’000’000, но поддерживает явно больше 500. Как минимум 150’00**

Проблемы рекурсии

Подробнее с проблемой с способами решения возможно ознакомиться в докладе Олега Фатхиева “Эволюция метапрограммирования”

Эволюция метапрограммирования: как правильно работать со списками типов

День 1 / 12:30 / Зал 2 / RU /

В избранное

Глубокий рассказ о метапрограммировании в ретроспективе. Обсудим общие подходы в метапрограммировании, а затем перейдём к спискам типов. По шагам напишем небольшую библиотеку для работы со списками типов, похожую на Boost.Hana. Для каждой проблемы рассмотрим несколько возможных решений: от использования устаревших, но не менее интересных подходов из C++98/03 до применения продвинутых техник из C++17 и даже немного из C++20.

Скачать презентацию

Все доклады

Олег Фатхиев

Яндекс

Работает в компании «Яндекс», занимается разработкой сетевого балансера запросов, а также некоторых базовых C++ библиотек Яндекса. В прошлом работал в Российском квантовом центре. Увлекается функциональным и метапрограммированием. Призер полуфинала соревнования ACM ICPC по спортивному программированию.

Добавим все это в наш пример.

Напомню, на чем мы остановились

Мы пытались раскрывать tuple при вызове функции

```
struct BlackJackDealer {
    void getTwoCards(Card * a, Card & b);
};
template<typename TObj, typename Fx>
void print(TObj & obj, Fx const & fx) {
    tuple<Card, Card> args { };
    (obj.*fx)(&get<0>(args), get<1>(args));
}
```

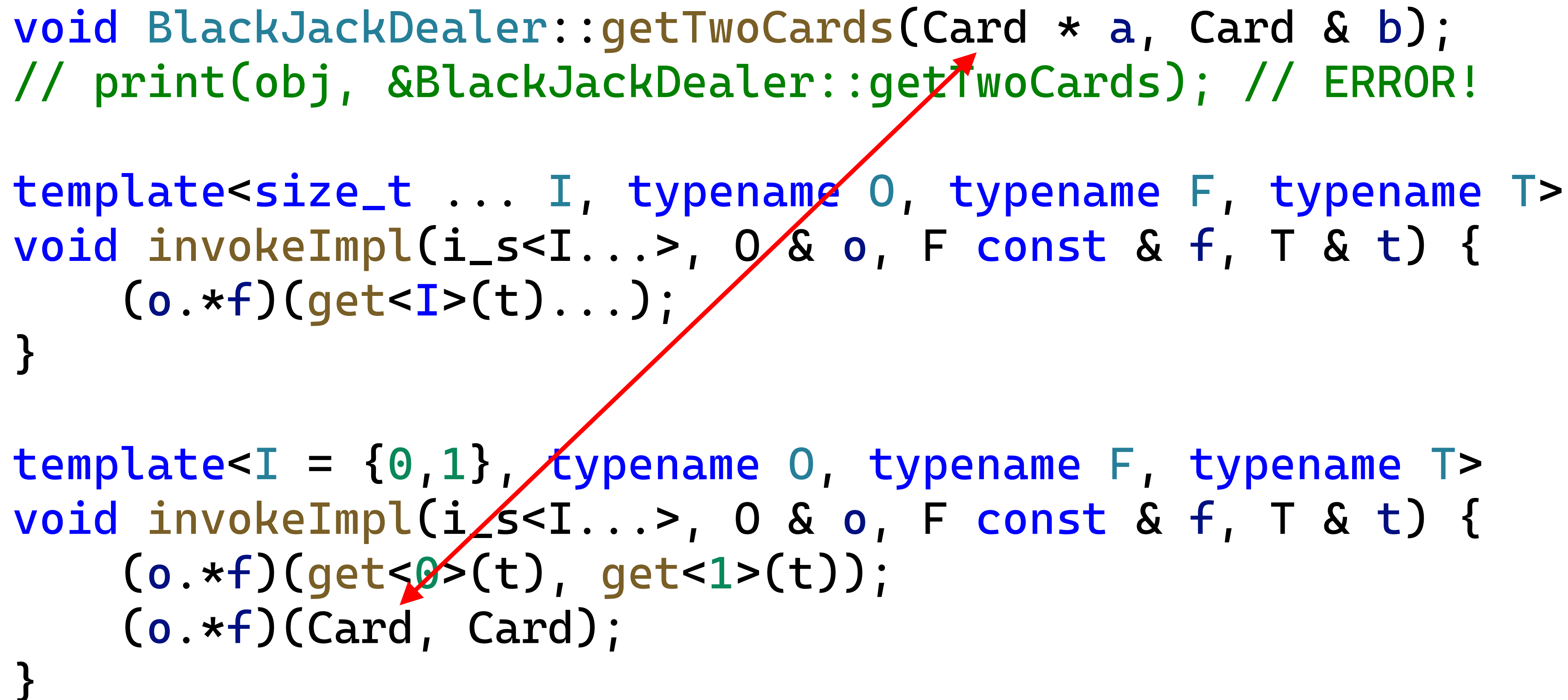
Добавим все это в наш пример.

Но нам понадобилась дополнительная функция

```
        // I = { 0,1,2,3... }
template<size_t ... I, typename O, typename F, typename T>
void invokeImpl(i_s<I...>, O & o, F const & f, T & t) {
    (o.*f)(get<I>(t)...);
}
template<typename TObj, typename Fx>
void print(TObj & obj, Fx const & fx) {
    using Args_t = typename function_info<Fx>::stack_args;
    constexpr size_t SIZE { tuple_size_v<Args_t> };
    Args_t args { };
    invokeImpl(make_i_s<SIZE>{}, obj, fx, args);
}
void BlackJackDealer::getTwoCards(Card * a, Card & b);
// print(obj, &BlackJackDealer::getTwoCards);
```

Добавим все это в наш пример.

Почему возникла ошибка?



Исправляем

Нужно обновить invokeImpl!

```
template<typename Arg, typename T>
decltype(auto) cndAddrOf (T& t) {
    if constexpr (is_pointer_v<Arg>) {
        return addressof(t);
    } else {
        return t;
    }
}
template<size_t ... I, typename O,  typename F, typename T>
void invokeImpl(i_s<I...>, O & o, F const & f, T & t) {
    using Args_t = typename function_info<Fx>::args;
    (o.*f)(cndAddrOf<tpl_el_t<I, Args_t>>(get<I>(t) )...);
}
```

Что получили в итоге

```
template<size_t ... I, typename O,  typename F, typename T>
void invokeImpl(i_s<I...>, O & o, F const & f, T & t) {
    using Args_t = typename function_info<Fx>::args;
    (o.*f)(cndAddrOf<tpl_el_t<I, Args_t>>(get<I>(t) )...);
}
template<typename TObj, typename Fx>
void print(TObj & obj, Fx const & fx) {
    using Args_t = typename function_info<Fx>::stack_args;
    constexpr size_t SIZE { tuple_size_v<Args_t> };
    Args_t args { };
    invokeImpl(make_i_s<SIZE>{}, obj, fx, args);
}
```

Вывод в поток

```
template<size_t ... I, typename T>
void streamImpl(index_sequence<Idx...>, ostream& os, T const& t)
 {
    (os << ... << (get<I>(t)));
}

template<typename T>
void printImpl(T const & t) {
    constexpr size_t SIZE { tuple_size_v<T> };
    streamImpl(make_index_sequence<SIZE>{}, std::cout, t);
}
```

Финальная версия

```
template<typename TObj, typename Fx>
void print(TObj & obj, Fx const & fx) {
    using Args_t = typename function_info<Fx>::stack_args;
    constexpr size_t SIZE { std::tuple_size_v<Args_t> };
    Args_t args { };
    invokeImpl(make_index_sequence<SIZE>{}, obj, fx, args);
    printImpl(make_index_sequence<SIZE>{},args);
}
```

Вызываем «вложенные» геттеры

Усложняем

```
struct Card {
    int getValue() { return 10; }
};
struct BlackJackDealer {
    Card * getNextCard();
};
struct Casino {
    void getBlackJackDealer(BlackJackDealer * & b);
};
void printCardValue() {
    Casino casino{ };
    cout << invoke(
      casino,
        &Casino::getBlackJackDealer,
          &BlackJackDealer::getNextCard,
            &Card::getValue);
    // Expected output: "10"
}
```

Попробуем разобраться

Как могла бы выглядеть такая функция?

```
template<typename O, typename ... F>
auto invoke (Obj & obj, TFx const & ... fxs) {
    tuple<A1> a1 {};
    (obj.*fxs1)(get<I1>(a1)...);

    tuple<A2> a2 {};
    (get<ST1>(a1).*fxs2)(get<I2>(a2)...);

    tuple<A3> a3 {};
    (get<ST2>(a2).*fxs3)(get<I3>(a3)...);
    /* ... */
}
```

Используем folding expression

Все выглядит довольно просто

```
template<typename T>
struct Invoker {
    T& t;
    Invoker(T & _t) : t { _t } { }
    template<typename Fx>
    auto operator<<(Fx const& fx) && {
        tuple<...> args { };
        (t.*fx)(get<I>(args));
        return Invoker { get<X>(args) };
    }
};
template<typename Obj, typename ... Args>
auto invoke(Obj & obj, Args const& ... a) {
    return (Invoker{ obj } << ... << a).get();
}
```

Давайте починим

```
template<typename T>
struct InvokerImpl {
    Tuple args;
    constexpr InvokerImpl(T const & method, Class & obj)
        : args{ } {
        invoke_impl(make_i_s<TupleSize>{}, method, obj);
    }
    template<typename Fx>
    auto operator<<(Fx fx) && {
        using Next = typename func_info<Fx>::class;
        return InvokerImpl<Fx>{ fx, get<Next>(args) };
    }
    template<size_t ... I, typename F, typename O>
    void invoke_impl(i_s<I...>, F const & f, O & o) {
        (obj.*fn)(get<Idx>(args)...);
    }
};
```

Давайте починим

Осталось только обновить invoker

```
template<typename T>
struct Invoker {
    T& t;
    Invoker(T & _t) : t { _t } { }
    template<typename Fx>
    auto operator<<(Fx const& fx) && {
        return InvokerImpl<Fx> { fx, t };
    }
};

template<typename Obj, typename ... Args>
auto invoke(Obj & obj, Args const& ... a) {
    return (Invoker{ obj } << ... << a).get();
}
```

Создаем «оглавление»

The road so far

Чего бы хотелось

```
constexpr Serializer<Casino> serialization_toc = {
    makeSerializer(&Casino::getter1),
    makeSerializer(&Casino::getter2),
    /* ... */
    makeSerializer(&Casino::getterN)
};
struct SerializationInterface {
    template<typename ... T>
    void operator()(char const* tag, std::tuple<T...> t);
};

serialization_toc(casino, serializationInterface);
```

Создадим вот такой вот constexpr invoker

```
template<auto ... Fx> struct fwd {};

template<typename T>
struct serializtion_invoker {
    template<auto ... Fx>
    constexpr serialization_invoker(fwd<Fx...>, char const* tag)
        : ptr { &theInvoker<Fx...> }
        , m_tag{ tag }
    { }
    void operator()(T& t, SerializationInterface & si) const  {
        (*ptr)(t, m_tag, si);
    }
    template<auto ... Fx>
    static void theInvoker(T& t, char const * tag, SI & si) {
        si(tag, invoke(t, Fx...));
    }
};
```

Создадим make-функцию

```
template<typename ...T>
struct first_class {};

template<typename T, typename ... R>
struct first_class<T, R...> {
    using type = typename function_info<T>::Class_t;
};

template<typename ...T>
using first_class_t = typename first_class<T...>::type;


template<auto ... fx>
constexpr auto makeSerializer(char const * tag) {
     return serialization_invoker<
                first_class<decltype(fx)...>::type
          > { fwd<fx...> {}, tag };
}
```

Последний шаг

Создадим сериализатор, который будет хранить «оглавление»

```
template<typename T, size_t N>
class serializer
{
    std::array<invoker<T>, N> m_arr;
public:
    template<typename ... Args>
    constexpr serializer(Args ... args) : m_arr{ args... } {}

    void operator()(T& obj, SerializationInterface & si) const {
        for (auto it : m_arr) { it (obj, si); }
    }
};
```

Получилось!

```
constexpr serializer<Casino, 3> toc{
    makeSerializer<&Casino::getBlackJack,
                                  &BlackJack::getCard,
                                      &Card::getValue>("call1"),
    makeSerializer<&Casino::getBlackJack,
                                  &BlackJack::getCard,
                                      &Card::getValue>("call2"),
    makeSerializer<&Casino::getBlackJack,
                                  &BlackJack::getCard,
                                      &Card::getValue>("call3"),
 };
toc(casino, serializationInterface);

// result.xml
<card id="call1" value="10" />
<card id="call2" value="3"  />
<card id="call3" value="BlackJack!" />
```

Эпилог

Эпилог

Это только начало!

- **Текущая реализация не оптимальна**
  - Если два разных сериализатора зависят от одного и того же геттера – он будет вызван дважды
  - Так же не поддерживаются геттеры, которые возвращают один и тот же тип дважды
  - Новые возможности C++ 20 (Концепты, расширение constexpr…)
- **Можно расширять:**
  - Вместо геттеров в «цепочку» можно включить еще указатели на члены класса, лямбда-функции, функции и т.д.

Сериализация объектов с блэкджеком и метапрограммированием

Александр Ганюхин <alexander.ganyukhin@mera.com>

Давайте создавать вместе!

QR-код на библиотеку (Apache2) →