Пишем tuple в 100 строк кода

Салихов Аяз

https://github.com/mathbunnyru/enhanced-tuple

Обо мне

- Более 6 лет в High-Frequency Trading компании
- Conan Community Reviewer
- Разработчик и мейнтейнер Python+Docker проекта в Jupyter
- Jupyter Distinguished Contributor
- Python, C++
- Увлекаюсь игрой на гитаре

Содержание

1. Постановка задачи
2. Готовимся к имплементации - Parameter pack / Fold expressions
3. Начинаем писать tuple
4. Пишем методы - *tuple_size*, *get*, *tuple_element*
5. Corner cases - дорабатываем имплементацию
6. Отличия от *std::tuple*
7. Выводы

1. Постановка задачи и первые идеи

1.1 Постановка задачи

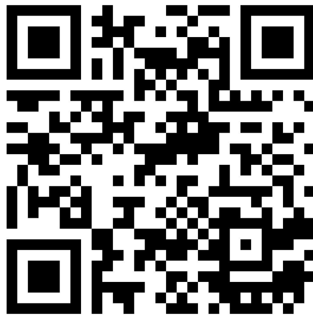
- Хотим отправлять пакет данных с заданными структурами, в каком-то порядке, с повторениями
- Структуры предоставлены вендором
- Хотим пробовать разные типы отправляемых пакетов, изменяя код в наименьшем количестве мест

```
struct AddOrder {
 char side;
 int64_t price;
 uint32_t amount;
 ...
};
```

```
struct Message {
 AddOrder v0;
 AddOrder v1;
 CancelOrder v2;
};

class Sender {
public:
 void send() {
   fill(message.v0);
   fill(message.v1);
   fill(message.v2);
   send(message);
 }

private:
 void fill(AddOrder&);
 void fill(CancelOrder&);
 template <typename T> void send(const T&);
};
```

1.2 Используем std::tuple!

```
#include <tuple>

using Message = std::tuple<AddOrder, AddOrder, CancelOrder>;

Message message;
std::apply(
 [](auto&...x) {
   (fill(x), ...);
 },
 message
);
send(message);
```



https://gcc.godbolt.org/z/rfGvMfzW9

1.2 Отправим по сети и выясним

```
struct A {
 int num;
};


template <size_t pos>
size_t offset_of(const std::tuple<A, A>& t) {
 return reinterpret_cast<const uint8_t*>(&std::get<pos>(t)) -
        reinterpret_cast<const uint8_t*>(&t);
}


int main() {
 std::tuple<A, A> t;
 std::cout << offset_of<0>(t) << '\n' << offset_of<1>(t) << '\n';
}
```

```
Вывод:
4
0
```

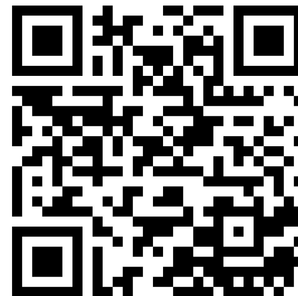
https://gcc.godbolt.org/z/5eaoEoKdj

1.3 Что можно сделать?

- Перевернуть tuple
- libc++ / microsoft STL
- 3rd-party tuple
- Не использовать std::tuple
- boost::pfr
- Написать свой tuple!

https://gcc.godbolt.org/z/5xn9zM6c4

```
#include "boost/pfr.hpp"
#include <tuple>


struct Message {
 AddOrder v0;
 AddOrder v1;
 CancelOrder v2;
};


int main() {
 Message message;
 boost::pfr::for_each_field(
   message,
   [](auto& field) { fill(field); }
 );
}
```



1.4 Что мы хотим

- C++ Reflection
- Итерация по всем полям объекта
- Количество полей у структуры
- Понятный memory layout
- Возможность писать обобщённый код

2. Готовимся к имплементации

2.1. Parameter pack

A template parameter pack is a template parameter that accepts zero or more template arguments (non-types, types, or templates). A function parameter pack is a function parameter that accepts zero or more function arguments.
https://en.cppreference.com/w/cpp/language/parameter_pack

```
template<class... Types>
struct Tuple {};
```

```
template<typename U, typename... Ts>
struct valid {};
```

```
template <class... Mixins> struct X : public Mixins... {
 X(const Mixins&...mixins) : Mixins(mixins)... {}
};
```

```
template<class T, T... Ints>
class integer_sequence;
```

```
template<class... Types>
void f(Types... args);
```

```
template <class... Ts>
void g(Ts... args) {
 f(&args...);
}
```

```
template <class... Args> void f(Args... args) {
 auto lm = [&, args...] { return g(args...); };
 lm();
}
```

```
template <class... Types>
struct count {
 inline static constexpr std::size_t value = sizeof...(Types);
};
```

2.2. Fold expressions

Синтаксис

```
( pack op ... ) - unary right
( ... op pack ) -  unary left
( pack op ... op init ) - binary right
( init op ... op pack ) - binary left
```

```
template <typename... Args>
constexpr bool all(Args... args) {
 return (... && args);
}
```

Операция:

```
+ - * / % ^ & | = < > << >> +=
-= *= /= %= ^= &= |= <<= >>= ==
!= <= >= && || , .* ->*
```

2.3 index_sequence

```
template<class T, T... Ints>
class integer_sequence;
```

```
template<class T, T N>
using make_integer_sequence = std::integer_sequence<T, /* a sequence 0, 1, 2, ..., N-1 */ >;

template<std::size_t... Ints>
using index_sequence = std::integer_sequence<std::size_t, Ints...>;

template<std::size_t N>
using make_index_sequence = std::make_integer_sequence<std::size_t, N>;

template<class... T>
using index_sequence_for = std::make_index_sequence<sizeof...(T)>;
```

3. Начинаем писать Tuple

3.1 Идеи имплементации

- Рекурсивная имплементация (с откусыванием головы)
- Множественное наследование
- Ручное управление памятью - char[]
- Комбинация вышеперечисленных

3.2 Множественное наследование - что не так?

```
template <typename... Types>
class Tuple : public Types...{};
```

3.2 Не все типы одинаковы

```
template <typename... Types>
class Tuple : public Types...{};


Tuple<int> t;
```

```
error: base specifier must name a class
class Tuple : public Types...{};
```

3.3 Начнём хранить элементы в контейнере

```
template <typename Type>
struct Item {
 Type value;
};


template <typename... Types>
class Tuple : public Item<Types>...{
 // ctor
};
```

А здесь?

3.3 Типы могут повторяться!

```
template <typename Type>
struct Item {
 Type value;
};

template <typename... Types>
class Tuple : public Item<Types>...{
 // ctor
};

Tuple<int, int> t;
```

```
error: base class 'Item<int>' specified more than once as a direct base class
class Tuple : public Item<Types>...{
```

3.4 Добавим индекс в *контейнер*

```
template <std::size_t /* Index */, typename Type>
struct IndexedItem {
 Type value;
};


template <typename... Types>
class Tuple : public ?? {
};
```

3.5 index_sequence + TupleBase

```
template <typename... Types>
class Tuple
 : public impl::TupleBase<
     std::index_sequence_for<Types...>,
     Types...
   > {
public:
 using TB = impl::TupleBase<std::index_sequence_for<Types...>, Types...>;
 using TB::TB;
};
```

```
template <std::size_t... Is, typename... Types>
struct TupleBase<std::index_sequence<Is...>, Types...>
   : public IndexedItem<Is, Types>... {
 template <typename... Us>
 TupleBase(Us&&...us) : IndexedItem<Is, Types>{std::forward<Us>(us)}... {}
};
```

4. Пишем методы tuple

4.1 Что нужно, чтоб заменить std?

- **Tuple**
- **std::tuple_size<Tuple>**
- **std::tuple_element<Tuple>**
- **get<Index>, get<Type>**
- **Deduction guides**
- std::swap
- operator==, operator<=>
- std::uses_allocator<Tuple>

- std::make_tuple
- std::tie
- std::forward_as_tuple

- std::tuple_cat

- std::apply
- std::make_from_tuple

4.2 tuple_size + tuple_element

```
template <class... Types>
struct std::tuple_size<Tuple<Types...>> {
 static constexpr auto value = sizeof...(Types);
};
```

```
template <std::size_t I, typename T, typename... Us>
struct std::tuple_element<I, tuple<T, Us...>> {
 static_assert(I < sizeof...(Us) + 1, "Index out of bounds.");
 using type = tuple_element<I - 1, tuple<Us...>>::type;
};

template <typename T, typename... Us>
struct std::tuple_element<0, tuple<T, Us...>> {
 using type = T;
};
```

4.3 get<Index> (using tuple_element)

```
template <size_t Index, typename... Types>
 requires(Index < sizeof...(Types))
constexpr auto& get(Tuple<Types...>& t) {
 using ResultT = tuple_element_t<Index, Tuple<Types...>>;
 return static_cast<IndexedItem<Index, ResultT>&>(t).value;
}
```

4.4 get<Type>

```
template <typename Type> struct GetterByType {
 template <size_t Index>
 static constexpr Type& get_reference(IndexedItem<Index, Type>& itm) {
   return itm.value;
 }
};

template <typename Type, typename... Types>
constexpr auto& get(Tuple<Types...>& t) {
 return GetterByType<Type>::get_reference(t);
}
```

4.5 Deduction guides (CTAD)

```
Tuple deducted_t(5, 3.5);
std::cout << get<int>(deducted_t) << ' ' << get<double>(deducted_t) << '\n';
```

```
template <typename... Types>
Tuple(Types...) -> Tuple<Types...>;
```

Спасибо за внимание!

Вопросы?

5. Corner cases

5.1 EBO в *libstdc++* (Empty base optimization)

```
template<size_t _Idx, typename _Head, bool = is_empty<_Head>::value>
 struct _Head_base;

template<size_t _Idx, typename _Head>
struct _Head_base<_Idx, _Head, true>
 : public _Head
 {};

template<size_t _Idx, typename _Head>
 struct _Head_base<_Idx, _Head, false>
 {
   _Head _M_head_impl;
 };
```

5.1 EBO в *libstdc++*

```
template<size_t _Idx, typename _Head, bool = __empty_not_final<_Head>::value>
 struct _Head_base;

template<size_t _Idx, typename _Head>
struct _Head_base<_Idx, _Head, true>
 : public _Head
 {};

template<size_t _Idx, typename _Head>
 struct _Head_base<_Idx, _Head, false>
 {
   _Head _M_head_impl;
 };
```

5.2 std::tuple в *libstdc++*

```
template<size_t _Idx, typename _Head, typename... _Tail>
  struct _Tuple_impl<_Idx, _Head, _Tail...>
  : public _Tuple_impl<_Idx + 1, _Tail...>,
    private _Head_base<_Idx, _Head>
    {};
```

5.3 EBO в C++20

```
template <std::size_t/* Index */, typename Type>
struct IndexedItem {
 [[no_unique_address]] Type value;
};
```

5.3 Default constructor

```
constexpr tuple();

Default constructor. Value-initializes all elements, if any. The default constructor is trivial if

sizeof...(Types) == 0
```

```
TupleBase::TupleBase() = default;
```

5.3 Default constructor C++20

```
template <std::size_t /* Index */, typename Type>
struct IndexedItem {
 [[no_unique_address]] Type value{};
};
```

https://quuxplusone.github.io/blog/2022/06/03/aggregate-parens-init-considered-kinda-bad/

5.4 std::tuple<T&>

```
#include <iostream>
#include <tuple>

int main() {
  int x = 10;
  int y = 20;
  std::tuple<int&> x_t(x);
  std::tuple<int&> y_t(y);
  x_t = y_t;
  std::cout << std::get<int>(x_t) << ' ' << x << '\n';
  x = 30;
  std::cout << std::get<int>(x_t) << ' ' << x << '\n';
}
```

Скомпилируется ли программа и что выведет на экран?

5.4 std::tuple<T&>

```cpp
#include <iostream>
#include <tuple>

int main() {
  int x = 10;
  int y = 20;
  std::tuple<int&> x_t(x);
  std::tuple<int&> y_t(y);
  x_t = y_t;
  std::cout << std::get<int&>(x_t) << ' ' << x << '\n';
  x = 30;
  std::cout << std::get<int&>(x_t) << ' ' << x << '\n';
}
```

```
Program returned: 0
 20 20
 30 30
```

https://gcc.godbolt.org/z/xsWWfsEfP

5.4 Tuple<T&> (IndexedItem<T&>)

```
template <std::size_t Index, typename Type>
struct IndexedItem<Index, Type&> {
 Type& value;

 constexpr IndexedItem(Type& val) : value(val) {}

 constexpr IndexedItem(const IndexedItem&) = default;

 constexpr IndexedItem& operator=(const IndexedItem& rhs) {
   value = rhs.value;
   return *this;
 }

 // + for move
};
```

6. Выводы

6.1 Отличия от std::tuple

- `Allocator`
- `operator=(const tuple& other)`
- Операторы сравнения
- `Swap`
- `Conditionally explicit`
- `noexcept`
- `tuple(pair)`
- `std::get`

6.2 Выводы

- Написать свой Tuple можно
- Написать std::tuple сложно
- Некоторые вещи могут быть лучше стандарта - например, количество создаваемых типов меньше
- Количество corner cases бывает велико
- Имплементации становятся проще (и быстрее) с выходом новых стандартов

6.3 Что посмотреть

C++ Siberia 2019: Олег Фатхиев, Эволюция метапрограммирования: списки типов

How C++20 Can Simplify std::tuple - Alisdair Meredith [ACCU 2019]

https://www.youtube.com/watch?v=IF51Gsu3Cec

https://www.youtube.com/watch?v=SvxBvSK4i4k

Спасибо за внимание!

Вопросы?

https://github.com/mathbunnyru
https://t.me/mathbunnyru
https://t.me/bun_life

7. Бонус

7.1 Пишем make_index_sequence

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

template <class Sequence1, class Sequence2>
struct merge_and_renumber;

template <size_t... I1, size_t... I2>
struct merge_and_renumber<index_sequence<I1...>, index_sequence<I2...>>
   : index_sequence<I1..., (sizeof...(I1) + I2)...>
{};

template <size_t N>
struct make_index_sequence: merge_and_renumber<
   typename make_index_sequence<N / 2>::type,
   typename make_index_sequence<N - N / 2>::type
>{};

template<> struct make_index_sequence<0> : index_sequence<> { };
template<> struct make_index_sequence<1> : index_sequence<0> { };
```

7.2 Проверяем make_index_sequence

```
template<size_t... Ints>
void f(index_sequence<Ints...> s) {
   ((std::cout << Ints << ' '), ...);
};


int main() {
   index_sequence<1, 2, 3> t;
   f(make_index_sequence<5>{});
}
```

https://gcc.godbolt.org/z/dWrb5ccEG

7.3 Пишем for_each

```
namespace impl {
template <typename TupleLike, typename Func, size_t... Idx>
void for_each(TupleLike&& t, Func&& f, std::index_sequence<Idx...>) {
 (f(get<Idx>(t)), ...);
}
} // namespace impl

template <typename TupleLike, typename Func>
void for_each(TupleLike&& t, Func&& f) {
 constexpr size_t TupleSize =
     std::tuple_size_v<std::remove_reference_t<TupleLike>>;
 impl::for_each(std::forward<TupleLike>(t), std::forward<Func>(f),
                std::make_index_sequence<TupleSize>{});
}
```