Variadic templates. Tuples, unpacking and more

C++*

В этом посте я поговорю о шаблонах с переменным числом параметров. В качестве примера будет приведена простейшая реализация класса *tuple*. Также я расскажу о распаковке *tuple*'а и подстановки, хранимых там значений в качестве аргументов функции. И напоследок приведу пример использования вышеописанных техник для реализации отложенного выполнения функции, которое может быть использовано, например, в качестве аналога *finally* блоков в других языках.

Теория

Шаблоном с переменным числом параметров (*variadic template*) называется шаблон функции или класса, принимающий так называемый *parameter pack*. При объявлении шаблона это выглядит следующим образом

```
template<typename… Args> struct some_type;
```

Такая запись значит то, что шаблон может принять 0 или более типов в качестве своих аргументов. В теле же шаблона синтаксис использования немного другой.

```
template<typename… Args> // Объявление
void foo(Args… args); // Использование
```

Вызов *foo(1,2.3, "abcd")* инстанциируется в *foo<int, double, const char*>(1, 2.3, "abcd")*. У *parameter pack*'ов есть много интересных свойств (например они могу использоваться в листах захвата лямбд или в *brace-init-lists*), но сейчас я хотел бы остановится на двух свойствах, которыми я буду активно пользоваться далее.

1. Вариадик параметр можно использовать в качестве аргумета вызова функции, применять к нему операции каста и т.п. При этом раскрывается он в зависимости от положения эллипсиса, а именно, раскрывается выражение непосредственно прилегающее к эллипсису. Звучит непонятно, но на примере думаю все станет ясно.

```
template<typename T>
T bar(T t) {/*...*/}

template<typename... Args>
void foo(Args... args)
{
    //...
}

template<typename... Args>
void foo2(Args... args)
{
    foo(bar(args)...);
}
```

В этом примере в функции *foo2*, так как эллипсис стоит после вызова *bar()*, то сначала для каждого

значения из args вызовется функция *bar()*, и в *foo()* в качестве аргументов попадут значения возвращенные *bar()*.
Еще несколько примеров.

```
(const args&..) // -> (const T1& arg1, const T2& arg2, ...)
((f(args) + g(args))...) // -> (f(arg1) + g(arg1), f(arg2) + g(arg2), ...)
(f(args...) + g(args...)) // -> (f(arg1, arg2,...) + g(arg1, arg2, ...))
(std::make_tuple(std::forward<Args>(args)...)) // -> (std::make_tuple(std::forward<T1>(arg1), std::for
ward<T2>(arg2), ...))
```

2. Количество параметров в паке можно получить используя оператор *sizeof...*

```
template<typename... Args>
void foo(Args... args)
{
    std::cout << sizeof...(args) << std::endl;
}

foo(1, 2.3) // 2
```

Tuple

Класс *Tuple* интересен, как мне кажется, даже не столько тем что для его написания и создания вспомогательных функций сейчас используются *variadic templates* (можно обойтись и без них), сколько тем, что *tuple* — рекурсивная структура данных, пришелец из другого, функционального мира (привет Haskell), что в свою очередь в очередной раз показывает насколько многогранен может быть C++.
Я приведу, набросанную на коленке простейшую реализацию такого класса, которая, тем не менее, показывает основную технику работы с *variadic* шаблонами — "откусывание головы" пака параметров и рекурсивная обработка "хвоста", которая, кстати, также широко распространена в функциональных языках.
Итак.
Базовый шаблон класса, никогда не инстанциируется, поэтому без тела.

```
    template<typename... Args>
    struct tuple;
```

Основная специализация шаблона. Здесь мы отделяем "голову" типов параметров и "голову" переданных нам аргументов в конструкторе. Этот аргумент мы сохраняем в текущем классе, остальными рекурсивно займутся базовые. Получить доступ к данным базового класса мы можем скастовав "себя" к базовому типу.

```
    template<typename Head, typename... Tail>
    struct tuple<Head, Tail...> : tuple<Tail...>
    {
        tuple(Head h, Tail... tail)
            : tuple<Tail...>(tail...), head_(h)
        {}
        typedef tuple<Tail...> base_type;
```

```
    typedef Head           value_type;

    base_type& base = static_cast<base_type&>(*this);
    Head       head_;
};
```

Последний штрих (что опять же привычно для функциональных языков) — это специализировать “дно” рекурсии.

```
template<>
struct tuple<>
{};
```

В общем-то, необходимый минимум уже написан. Можно пользоваться нашим классом следующим образом:

```
tuple<int, double, int> t(12, 2.34, 89);
std::cout << t.head_ << " " << t.base.head_ << " " << t.base.base.head_ << std::endl;
```

Однако, отсчитывать вручную, сколько раз надо написать .base, чтобы добраться до нужного нам элемента не очень удобно, поэтому в стандартной библиотеке написан шаблон функции *get()*, позволяющий получить содержимое N-ного элемента объекта класса *tuple*. Мы вынуждены обернуть функцию в структуру, чтобы обойти запрет на частичную специализацию функций. В этом базовом шаблоне также происходит “откусывание головы” от тупла и перенаправление к следующему типу *getter* со значением индекса на единицу меньше как в случае типа элемента, так и, собственно, функции получения этого элемента.

```
template<int I, typename Head, typename... Args>
struct getter
{
    typedef typename getter<I-1, Args...>::return_type return_type;
    static return_type get(tuple<Head, Args...> t)
    {
        return getter<I-1, Args...>::get(t);
    }
};
```

И лишь когда мы стукнемся о дно рекурсии, можно сделать первые реальные действия. Тип возвращаемого значения мы возьмем на этот раз из тупла и вернем взятое оттуда же значение.

```
template<typename Head, typename... Args>
struct getter<0, Head, Args...>
{
    typedef typename tuple<Head, Args...>::value_type return_type;
    static return_type get(tuple<Head, Args...> t)
    {
        return t.head_;
```

```
        }
    };
```

Ну и как это обычно принято, пишется небольшая вспомогательная функция, избавляющая нас от необходимости вручную писать параметры шаблона структуры.

```
    template<int I, typename Head, typename... Args>
    typename getter<I, Head, Args...>::return_type
    get(tuple<Head, Args...> t)
    {
        return getter<I, Head, Args...>::get(t);
    }
```

Эту функцию мы и используем.

```
    test::tuple<int, double, int> t(12, 2.34, 89);
    std::cout << t.head_ << " " << t.base.head_ << " " << t.base.base.head_ << std::endl;
    std::cout << get<0>(t) << " " << get<1>(t) << " " << get<2>(t) << std::endl;
```

Unpacking

Распаковка *tuple* в C++! Что может быть круче=)? Эта возможность показалась настолько важной создателям Питона, что они даже внесли в язык специальный синтаксис для поддержки этой операции. Теперь мы можем пользоваться этим и в C++. Реализовать это можно по-разному (по крайней мере внешне, сам принцип везде один и тот же), я же покажу здесь самое простое на мой взгляд решение. К тому же оно напоминает то, что мы видели выше при реализации *getter*'а для извлечения элементов тупла. Здесь нам поможет свойство номер 1, описанное в теории выше. Наша функция распаковки должна выглядеть как-то так

```
template<typename F, typename Tuple, int… N>
auto call(F f, Tuple&& t)
{
    return f(std::get<N>(std::forward<Tuple>(t))...);
}
```

Как вы помните,

```
f(std::get<N>(std::forward<Tuple>(t))...);
```

распакуется в

```
 f(std::get<N1>(std::forward<Tuple>(t)), std::get<N2>(std::forward<Tuple>(t)), ...)
```

Но тут есть одна проблема, а именно, в такой функции нужно будет вручную указывать все интовые аргументы шаблона, причем указывать их правильно (в нужном порядке и нужное количество). Было бы очень хорошо, если бы удалось автоматизировать этот процесс. Для этого поступим похожим на подход к извлечению элементов из тупла образом.

```
template<typename F, typename Tuple, bool Enough, int TotalArgs, int... N>
struct call_impl
{
    auto static call(F f, Tuple&& t)
    {
        return call_impl<F, Tuple, TotalArgs == 1 + sizeof...(N),
                                   TotalArgs, N..., sizeof...(N)
                        >::call(f, std::forward<Tuple>(t));
    }
};
```

Здесь, как мне кажется, стоит объяснить подробнее. Начнем с параметров шаблона. С *F* и *Tuple* я думаю все понятно. Первый отвечает за наш *callable*объект, второй, собственно, за тупл, из которого мы будет брать объекты и подсовывать *callable*'у в качестве аргументов вызова. Далее идет булевый параметр *Enough*. Он сигнализирует набралось ли уже достаточно int параметров в*...N* и по нему мы будем далее специализировать наш шаблон. Наконец, *TotalArgs*— значение равное размеру тупла. В функции *call* мы, как и раньше, перенаправляем рекурсивно вызов к следующей инстанциации шаблона.
При этом в самом первом вызове тип будет

```
call_impl<F, Tuple, TotalArgs == 1,  TotalArgs,  0> // (N… - пусто, sizeof...(N) = 0)
```

, во втором

```
call_impl<F, Tuple, TotalArgs == 2,  TotalArgs,  0, 1>  // (N… =0, sizeof...(N) = 1)
```

и т.п. то есть ровно что нам и нужно.

Наконец нам нужна специализация, в котором будут производится реальные действия, будет наконец вызываться наша функция с нужными аргументами. Эта специализация выглядит следующим образом

```
template<typename F, typename Tuple, int TotalArgs, int... N>
struct call_impl<F, Tuple, true, TotalArgs, N...>
{
    auto static call(F f, Tuple&& t)
    {
        return f(std::get<N>(std::forward<Tuple>(t))...);
    }
};
```

Также не помешает вспомогательная функция.

```
template<typename F, typename Tuple>
auto call(F f, Tuple&& t)
{
    typedef typename std::decay<Tuple>::type type;
    return call_impl<F, Tuple, 0 == std::tuple_size<type>::value,
                                  std::tuple_size<type>::value
                    >::call(f, std::forward<Tuple>(t));
}
```

Здесь, я думаю, все прозрачно.

Использовать это можно следующим образом.

```
int foo(int i, double d)
{
    std::cout << "foo: " << i << " " << d << std::endl;
    return i;
}

    std::tuple<int, double> t1(1, 2.3);
    std::cout << call(foo, t1) << std::endl;
```

Defer

Описанные выше приемы, позволяют организовывать ленивые, отложенные вычисления. В качестве частного примера таких вычислений я рассмотрю здесь ситуацию, когда нужно выполнить какой-то функционал, независимо от того каким образом мы выходим из функции, независимо от условных конструкций внутри, а также от того было ли вызвано исключение. Такая поведение похоже на finally блоки в питонах и явах или, например, в языке Go есть оператор defer, который обеспечивает описанное выше поведение.
Хочу сразу оговориться, что как и многое другое в C++, эту задачу можно решить разными способами, например, используя *std::bind* или лямбду, собирающую аргументы и возвращающую другую лямбду и т.п. Но также вполне подойдет и хранение *callable* объекта и тупла с нужными аргументами. Собственно, зная то, что мы уже знаем, реализация тривиальна.

```
template<typename F, typename... Args>
struct defer
{
    defer(F f, Args&&... args) :
        f_(f), args_(std::make_tuple(std::forward<Args>(args)...))
    {}
    F f_;
    std::tuple<Args...> args_;

    ~defer()
    {
        try
        {
            call(f_, args_);
        }
        catch(...)
        {}
    }
};
```

Как обычно, вспомогательная функция

```
template<typename F, typename... Args>
defer<F, Args...> make_deferred(F f, Args&&... args)
{
    return defer<F, Args...>(f, std::forward<Args>(args)...);
```

```
}
```

И использование

```
auto d = make_deferred(foo, 1 ,2);
```

tuple, deferred, templates