Нумерация аргументов variadic template, или что скрывает скромный pair

из песочницы

C++*

Освоение стандарта C++11 — процесс, который не может происходить скачкообразно. Изучение новой языковой конструкции требует не только заучивания синтаксиса, но и осмысления её предназначения и типичных способов применения. Важным подспорьем в обучении является похорошевшая STL, которая зачастую может открыть глаза на существование весьма интересных и нужных возможностей. А уж зная, что какая-то вещь возможна и реализована в STL, докопаться до способа реализации нетрудно.

Об одном из любопытных примеров, связанном с обновлённым и улучшенным классом *pair*, и пойдёт речь в статье.
Новый стандарт добавил такой, казалось бы, простой вещи, как pair, удобства и универсальности. Если раньше к типам, входящим в состав пары, предъявлялись достаточно суровые требования, то сейчас слепить в пару можно практически что угодно. В частности, снято ограничение на конструирование таких типов. Теперь необязательно применять операции копирования или даже перемещения, возможно создание пары непосредственным конструированием членов (такая операция называется emplace, «размещение», и в C++11 поддерживается контейнерами STL), с применением нетривиальных конструкторов.

… А вот тут, как говорится, подробнее. Каким образом мы можем вызвать конструктор *pair* и передать ему два набора аргументов, да так, чтобы он понял, какие аргументы отдать какому конструктору? Среди привычных конструкторов, связанных с копированием или перемещением членов или целикового *pair*, видим такое:

```
template< class... Args1, class... Args2 >
pair( std::piecewise_construct_t,
       std::tuple<Args1...> first_args,
       std::tuple<Args2...> second_args );
```

piecewise_construct_t — просто пустой тип, который поможет нам сигнализировать, что мы хотим именно создать *pair* по кусочкам, передав аргументы конструкторам *first* и *second*. В этом нам поможет константа такого типа под названием *piecewise_construct*. Ну а дальше мы указываем два набора аргументов, упаковав их в кортежи (tuple, в их создании поможет функцияmake_tuple). Для тех читателей, которые забыли или не в курсе, что это такое, напомню: кортеж — собрание произвольного количества значений произвольного типа. В C++ с его строгим контролем типов кортежи реализованы при помощи шаблонов с переменным количеством аргументов (variadic template).

Что же, вроде бы проблема удачно решена: в качестве «упаковок» аргументов для конструкторов *first* и *second* выступают кортежи. На этом этапе у программиста, знакомящегося с

новинками стандарта, может возникнуть вопрос: «Кстати, а как распаковываются данные из кортежей?» Документация даёт нам единственный способ: функция get, которой в качестве шаблонного параметра указывают индекс элемента в кортеже.

Каким же образом наши аргументы попадут в конструктор? Извлекать данные из кортежа по одному несложно. Несложно извлекать их рекурсивно. Но как уместить все значения в вызове функции (в данном случае — конструктора)?
Здесь пригодится распаковка variadic-шаблонов. Однако распаковывать надо не список типов *tuple*, а список индексов. Который сначала надо ещё изготовить.

Начнём с основного: сделаем добавление нового индекса к уже существующему списку. Очевидно, что если нумерация начинается с 0, то новый индекс будет равен размеру входного списка. Нам понадобится структура, которую мы параметризуем списком индексов:

```
template<size_t ... Indices> struct PackIndices {
    // Здесь мы добавим к Indices новое число, равное sizeof ... (Indices)
};
```

Результат, представляющий из себя список целочисленных констант времени компиляции и одновременно — аргументы шаблона, нельзя хранить сам по себе, но можно хранить тип, параметризованный этими аргументами. И у нас как раз уже есть подходящий кандидат на роль такого типа:

```
template<size_t ... Indices> struct PackIndices {
    typedef PackIndices<Indices... , sizeof ... (Indices)> next;
};
```

Теперь сделаем рекурсивный генератор, создающий список индексов длиной**N**. Делается это простым добавлением последнего индекса к списку длиной **N-1**:

```
template<size_t N> struct CreatePackIndices {
    typedef typename CreatePackIndices<N-1>::type::next type;
};
```

… и остановим рекурсию:

```
template<> struct CreatePackIndices<0> {
    typedef PackIndices<> type;
};
```

Заполучив способ создания списка индексов, займёмся распаковкой кортежа в параметры конструктора. Для простоты рассмотрим сначала конструирование только одного объекта, *first*.

С использованием кортежа *args* и списка индексов *Indices* распаковка должна выглядеть в своей основе так:

```
first(std::get<Indices>(args)...)
```

Чтобы получить доступ к *Indices*, надо, чтобы контекст, в котором мы производим распаковку (то есть, конструктор *pair*), был этим списком параметризован. Это означает, что нам понадобится создать второй конструктор-шаблон со всеми нужными параметрами. Здесь кстати придётся ещё одна новинка C++11, делегирование конструкторов, которое позволяет вызывать альтернативный конструктор в списке инициализации. А поскольку мы всё равно производим вызовы функций, то воспользуемся автоматическим выводом типа аргументов: передадим вспомогательному конструктору анонимный объект *PackIndices*. В результате мы получаем такой одноногий*pair*:

```
template<typename T> class pair {

    // Конструктор, распаковывающий кортеж
    template<typename ... ArgTypes, size_t ... Indices>
    pair(std::tuple<ArgTypes...>& first_args, PackIndices<Indices...>):
        first(std::get<Indices>(first_args)...)
    {}

public:
    // Конструктор, доступный пользователю
    template<typename ... ArgTypes>
    pair(std::piecewise_construct_t, const std::tuple<ArgTypes...>& first_args):
        pair(first_args, typename CreatePackIndices<sizeof ... (ArgTypes)>::type())
    {}

private:
    T first;
};
```

Здесь самое время вспомнить про perfect forwarding — механизм, необходимый для корректной передачи аргументов во вложенные вызовы без изменения их типов. В обновлённом STL предусмотрена функция forward, которую придётся применить к каждому аргументу и к тому же параметризовать типом аргумента. К счастью, создатели нового стандарта предусмотрели такую хитрую штуку, как *одновременная* распаковка нескольких наборов аргументов.
Поскольку *ArgTypes* и *Indices* заведомо имеют одинаковую длину, можно смело добавить вызов *forward* в паттерн распаковки:

```
template<typename ... ArgTypes, size_t ... Indices>
pair(std::tuple<ArgTypes...>& first_args, PackIndices<Indices...>):
    first(std::forward<ArgTypes>(std::get<Indices>(first_args))...)
{}
```

После того, как пройден весь путь от значений в *make_tuple* до параметров конструктора, поставим *pair* на обе ноги:

Код

```
template<typename T1, typename T2> class pair {

    // Конструктор, распаковывающий кортеж
    template<typename ... ArgTypes1, size_t ... Indices1, typename ... ArgTypes2, size_t ... Indices2>
    pair(std::tuple<ArgTypes1...>& first_args, std::tuple<ArgTypes2...>& second_args,
         PackIndices<Indices1...>, PackIndices<Indices2...>):
        first(std::forward<ArgTypes1>(std::get<Indices1>(first_args))...),
        second(std::forward<ArgTypes2>(std::get<Indices2>(second_args))...)
    {}

public:
    // Конструктор, доступный пользователю
    template<typename ... ArgTypes1, typename ... ArgTypes2>
    pair(std::piecewise_construct_t, std::tuple<ArgTypes1...> first_args, std::tuple<ArgTypes2...> second_args):
        pair(first_args, second_args,
            typename CreatePackIndices<sizeof ... (ArgTypes1)>::type(),
            typename CreatePackIndices<sizeof ... (ArgTypes2)>::type())
    {}

private:
    T1 first;
    T2 second;
};
```

Разумеется, этот приём полезен не только для создания кустарно-велосипедного *pair*. Так, программисту, имеющему дело со стековыми виртуальными машинами, наверняка придут на ум врапперы для функций. Несомненно, найдутся применения и в других областях.

C++, C++11, template, variadic, variadic templates