Нововведения в шаблонах

18. января 2013 разработка

Как вы наверно знаете, язык C++ считается языком мультипрадигменным. Тремя основными парадигмами являются: процедурное программирование, объектно ориентированное программирование и программирование обобщенное, или, если быть более точным – метапрограмирование. О нововведениях в последнем и пойдёт речь в настоящей статье. Так как наиболее сложным нововведением являются шаблоны с переменным количеством параметров(*variadic templates*), львиную долю статьи будет занимать именно их описание. По моему мнению появление *variadic templates* является наиболее значимым и ожидаемым новшеством C++11, с которым может поспорить разве что появление многопоточной модели описанной в предыдущих статьях.

Шаблоны с переменным количеством параметров

Всё то, что связано с метапрограммированием в C++, как правило, вызывает панику в неокрепшем мозгу, а об ошибках компилятора порождаемых шаблонным кодом ходят легенды не самого приятного содержания. Код *boost* вообще стал притчей во языцах и для многих нет ничего более непонятного в этом мире, чем код библиотек *boost*. Хотя я понимаю чем вызвано подобное отношение, я считаю, что в синтаксисе шаблонов нет ничего сложного, просто из довольно простых блоков всегда можно наворотить такой агрегат, что сам создатель потом с трудом в нём сможет разобраться. *Variadic templates* не стали исключением и являют собой довольно продуманный и элегантный дизайн к рассмотрению которого мы и приступаем

Базовый синтаксис

В сердце нового синтаксиса лежит оператор, который уже имеет своё назначение в C++ - *эллипсис(…)*. С новым стандартом помимо своего старого употребления, в рамках аргументов функции, он приобретает совершенно новое назначение в шаблонах. В шаблонах, в отличие от своего собрата по аргументам функции, эллипсис не является самостоятельным оператором, а является дополняющим оператором и означает разные вещи в разных контекстах применения. Рассмотрим базовый пример:

```
1 template <typename ... Args>
2 struct A;
```

Эта запись означает буквально следующее: декларация класса A, который может иметь переменное количество шаблонных параметров(от 0 до ∞). Подобная запись для шаблонов означает *пачку шаблонных параметров(template parameter pack)*. Здесь эллипсис употребляется как указание на то, что количество параметров в *пачке* переменно. Более того, подобная запись применима и к не-типам в параметрах шаблона, т.е. следующая запись полностью легальна, и означает переменное количество *int* в параметрах шаблона:

```
1 template <int ... Numbers>
```

```
2  struct A;
```

> Нет никакой необходимости ставить пробелы между*typename* и *…,* я их добавил просто для наглядности. В дальнейшем я буду “прилеплять” эллипсис к*typename*

Следующим применением эллипсиса является его конкатенация с оператором *sizeof…*:

```
1  template<typename... Args>
2  struct A
3  {
4      static const size_t number = sizeof...(Args);
5  };
6
7  int main()
8  {
9      std::cout << A<bool, int, int, int, int, int>::number;
10 }
```

Оператор *sizeof…* возвращает количество параметров в пачке.

> Не путать с sizeof(Args)…! Эт[illegible] запись означает совсем другое.

Последним и, на мой взгляд, наиболее сложным, употреблением эллипсиса в контексте шаблонов является *распаковка пачки(pack expansion).* Пример:

```
1  template<typename... Args>
2  struct A
3  {
4      typedef std::tuple<Args...> Tuple_t;
5  };
```

Здесь эллипсис применяется к пачке **Args** тем самым распаковывая её в параметры *std::tuple*. Т.е. если мы создадим конкретный экземпляр такого шаблона, то *std::tuple* будет иметь все те же параметры, что и инстанциированный класс:

```
1  //Tuple_t есть typedef на std::tuple<int, float, std::string>
2  A<int, float, std::string>::Tuple_t tuple;
```

В выражении *распаковки,* представленном выше, левая часть(перед эллипсисом; в примере **Args**) называется *образцом(pattern).* И эллипсис проводит *распаковку* согласно образцу, в зависимости от контекста распаковки. В примере выше *распаковка* происходила в контексте списка аргументов шаблона *std::tuple*. Контексты могут быть следующие:

1. В списке аргументов функции. Образцом выступает тип аргумента.(parameter-declaration)
2. В параметрах шаблона. Образцом выступает тип параметра.(type-parameter)
3. В списке инициализации. Образцом выступает инициализирующее выражение.(initializer-clause)
4. В перечислении базовых классов, при наследовании. Образцом выступает тип базового класса.(base-specifier)
5. В списке инициализации конструктора. Образцом выступает инициализатор. (mem-inititalizer)
6. Список аргументов шаблона. Образцом выступает аргумент шаблона. (template-argument)
7. В спецификации исключений. Образцом выступает тип.(type-id)
8. В списке атрибутов. Образцом выступает атрибут.(attribute)
9. В спецификации выравнивания. Образцом выступает спецификатор выравнивания(alignment-specifier)
10. В списке захвата лямбда-функции. Образцом выступает аргумент с типом захвата. (capture)
11. В выражении *sizeof*....(identifier)

Рассмотрим пример, в котором применим все вышеперечисленные контексты:

```
template<typename... Types>
struct C: Types...//#4 образец - Types
{
    typedef std::tuple<const Types...> Tuple_t;//#6 образец - cor
    C(): Types()...//#5 образец - Types()
    {}
    void executor(const Types&... args)//#1 образец - const Types
    {
        auto lambda = [&args...]()//#10 образец - &args
        {
            std::cout << sizeof...(Types) << "\n";//#11
        };
        alignas(Types)... int a[];//#9 образец - alignas(Types)
    }
    void tooMuchThrowing() throw(Types...)//#7 образец - Types
    {}
    [[Types...]] void attributedFun()//#8 образец - Types
    {}
    template<Types... inner>//#2 образец Types
    class innerC{};
};

template<int... Values>
void perfectSquare()
{
    auto list = {(Values*Values)...};//#3  образец - Values*Value
    for(auto& item : list)
        std::cout << item << " ";
}

int main()
{
    C<A, B> c;
    perfectSquare<1, 2, 3, 4>();
}
```

```
35
```

Вот как будет выглядеть класс после инстанциации произведенной в *main*:

```
struct C: A, B
{
    typedef std::tuple<const A, const B> Tuple_t;
    C(): A(), B()
    {}
    void executor(const A& args1, const B& args2)
    {
        auto lambda = [&args1, &args2]()
        {
            std::cout << 2 << "\n";
        };
        alignas(A) alignas(B) int a[];
    }
    void tooMuchThrowing() throw(A, B)
    {}
    //Вероятно это будет  возможно в будущем, но сейчас атрибутов
    // и они не относятся к A, B типам
    [[A, B]] void attributedFun()
    {}
    class innerC<A, B>{};
};

void perfectSquare<1, 2, 3, 4>()
{
    auto list = {1, 4, 9, 16};
    for(auto& item : list)
        std::cout << item << " ";
}
```

Разумеется это не правильный C++ код и он не будет компилироваться, я привел это для того, чтобы показать идею.

Таким образом вы могли заметить, что пачка всегда распаковывается в некий список. Причем эта распаковка выполняется довольно умно*;* к примеру, следующий код будет работать замечательно даже с пустой пачкой:

```
template<typename... Types>
void fun(int a, Types... args, int b);
```

С пустой пачкой вышеозначенная функция будет инстанциирована в ничто иное как в:

```
void fun(int a, int b);
```

И никаких проблем с запятыми! Variadic templates действительно хорошо спроектированы.

Еще одна особенность проявляется в использование нескольких разных пачек с одной сущностью:

```
struct Base
{};

template<typename T, typename U>
struct Derived: Base
{};


template<typename... Ts>
struct Victim
{
    template<typename... Us>
    static void fun()
    {
        std::initializer_list<Base*> list = {(new Derived<Ts, Us>
    }
};

int main()
{
    Victim<int, void, char>::fun<double, float, char>();
    //Ошибка, количество Ts меньше Us
    Victim<int, void>::fun<double, float, char>();
    //Ошибка, количество Us меньше Ts
    Victim<int, void, char>::fun<double, char>();
}
```

Комментарии говорят сами за себя – при одновременной распаковке двух пачек их размер должен совпадать! После вышеприведённой распаковки в списке будут указатели на следующие элементы: *Derived<int, double>, Derived<void, float>, Derived<char, char>.* Т.е. распаковка при участии двух и более пачек происходит попарно.

Распаковка пачки происходит изнутри-наружу, т.е. сначала раскрываются внутренние пачки, а затем внешние, следующий пример показывает это на практике. Он содержит необходимые комментарии и я не буду пояснять его:

```
template<typename... Ts>
struct Victim
{
    template<typename... Us>
    struct SubVictim
    {
        //tuple содержит один tuple который содержит Ts+Us типов
        //Количество Us и Ts может быть разным.
        typedef std::tuple<std::tuple<Ts..., Us...>> SingleElemer
        //tuple содержит Ts(или Us) tuple'ов каждый из которых сс
        //Количество Us и Ts должно совпадать
```

```
        typedef std::tuple<std::tuple<Ts, Us>...> MultipleElement
        //tuple содержит Us tuple'ов каждый из которых содержит T
        //Количество Us и Ts может быть разным
        typedef std::tuple<std::tuple<Ts..., Us>...> MMultipleEle
    };
};

int main()
{
    typedef Victim<int, int, char>::SubVictim<double, float, char
SingleElementTuple_t SE_t;
    typedef Victim<int, int, char>::SubVictim<double, float, char
MultipleElementsTuple_t ME_t;
    typedef Victim<int, int, char>::SubVictim<double, float, char
MMultipleElementsTuple_t MME_t;
    std::cout << "=======Tuple consists of single tuple with"
        " sizeof...(Ts)+sizeof...(Us) types===\n";
    std::cout <<std::tuple_size<SE_t>::value << "\n";
    std::cout << std::tuple_size<
       std::tuple_element<0, SE_t>::type>::value << "\n";;
    std::cout << "=======Tuple consists of sizeof...(Ts)"
         " tuples with 2 types each======\n";
    std::cout <<std::tuple_size<ME_t>::value << "\n";
    std::cout << std::tuple_size<
      std::tuple_element<0, ME_t>::type>::value << "\n";;
    std::cout << "=======Tuple consists of sizeof...(Us) tuples w
        " sizeof...(Ts)+1 types each==\n";
    std::cout <<std::tuple_size<MME_t>::value << "\n";
    std::cout << std::tuple_size<
      std::tuple_element<0, MME_t>::type>::value << "\n";;
}
```

В качестве примера, которые вполне можно использовать в реальной жизни, я хочу привести аналог функции *std::make_shared* для*std::unique_ptr:*

```
template <class T, class... Args>
std::unique_ptr<T> make_unique(Args&&... args)
{
    return std::unique_ptr<T>(new T(std::forward<Args>(args)...));
}
```

Итак мы рассмотрели синтаксис работы с variadic templates, и сейчас мне бы хотелось перечислить те недостатки, которые мне видятся в нём:

- Отсутствие возможности сохранить пачку для последующей работы с ней. Т.е. typedef T… Something; является некорректным C++ кодом.

- Нельзя получить часть пачки, к примеру мы не хотим использовать все типы, а только часть из них.
- Нельзя получить элемент из пачки, по его номеру или еще как-нибудь.

Эти недостатки заставляют нас изворачиваться, чтобы получить желаемое, ведь для создание чего-то более сложного чем проброс аргументов другой сущности нам необходимо развернуть пачку. Об одном из методов такой развёртки и пойдёт речь в следующем параграфе.

Рекурсия

Первым методом развёртывания пачки, который мы рассмотрим, является рекурсия. Как вы, вероятно, знаете в рекурсии всегда есть общий и базовый случаи. Базовый случай необходим для завершения рекурсии.

В качестве базы для примеров развёртывания мы возьмем простую задачу: получить строку бит из заранее известной последовательности значений интегральных типов.

Решение через классы

Начнём мы наше решение с задания общего случая:

```
template <bool... Bits>
struct recursiveClassBitsUnrolling;
```

> Вообще говоря в этом примере нет необходимости в данном общем случае, но я его привёл, т.к. он очень часто будет вам встречаться. Во многих местах он всё таки нужен.

Мы декларировали класс, который никогда не будет применяться и который нужен лишь для того, чтобы в дальнейшем обозначить более конкретные случаи. Более конкретизированный общий случай будет выглядеть следующим образом:

```
template <bool FirstBit, bool... Tail>
struct recursiveClassBitsUnrolling<FirstBit, Tail...>
{
    static std::string exec()
    {
        return recursiveClassBitsUnrolling<FirstBit>::exec() +
recursiveClassBitsUnrolling<Tail...>::exec();
    }
};

```

Происходит здесь следующее: при каждой инстанциации шаблонного класса **recursiveClassBitsUnrolling** с 2-мя и более шаблонными аргументами от общей пачки аргументов отщипывается один(**recursiveClassBitsUnrolling<FirstBit>::exec()**), а оставшиеся инстанциируют новый класс с количеством аргументов уменьшенным на единицу(**recursiveClassBitsUnrolling<Tail...>::exec()**). Т.е., к примеру, для **recursiveClassBitsUnrolling<true, false, true>::exec()** получится следующий псевдокод:

```
template <bool FirstBit=(true), bool... Tail=(false, true)>
struct recursiveClassBitsUnrolling<FirstBit, Tail...>
{
    static std::string exec()
    {
        return recursiveClassBitsUnrolling<true>::exec() +
recursiveClassBitsUnrolling<(false, true)>::exec();
    }
};

```

Таким образом рекурсия всегда конечна и для того, чтобы всё встало на свои места осталось лишь определить базовый случай для одного аргумента:

```
template <bool Bit>
struct recursiveClassBitsUnrolling<Bit>
{
    static std::string exec()
    {
        return  boost::lexical_cast<std::string>(Bit);
    }
};

```

В коде используется кусочек из boost, но вас не должно это смущать(это всего лишь преобразование из *bool* в *std::string*). Таким образом получается, что мы откусываем по одному *bool* слева-направо на каждом витке рекурсии и преобразуем его в строку содержащую 0 или 1. Пример использования:

```
int main()
{
    std::cout << recursiveClassBitsUnrolling<1, 0, 1,
 false, false, 1, 1, 0, 1, 1, 1, 1>::exec() << "\n";
}

```

Вывод:

```
101001101111
```

Описанный выше метод является примером реализации модели рекурсивной развёртки шаблонных параметров во что-то, что можно использовать. Вы встретите(и сами напишите) немало рекурсивной развёртки где будет больше частных случаев, или базовый случай не будет вообще иметь аргументов. Но модель, так или иначе, будет схожей.

Решение через функции

В предыдущем примере мы использовали классы, хотя имели чисто функциональную задачу. И не мудрено, ведь функции не поддерживают частичную специализацию, без которой нам, казалось бы, не обойтись. Но,

на самом деле, это вполне реализуемая задача которая решается с помощью перегрузки. Начнём определение рекурсии с базового случая:

```
template <bool Bit>
std::string recursiveFunBitsUnrolling()
{
    return  boost::lexical_cast<std::string>(Bit);
}

```

Полагаю, что тут всё понятно и пояснения излишни, так что переходим к общему случаю:

```
template <bool FirstBit, bool SecondBit, bool... Tail>
std::string recursiveFunBitsUnrolling()
{
    return recursiveFunBitsUnrolling<FirstBit>() +
     recursiveFunBitsUnrolling<SecondBit, Tail...>();
}

```

А вот тут уже есть интересный момент: обратите внимания. что помимо пачки шаблонных параметров мы имеем **два** обязательных шаблонных параметра. Именно наличие двух параметров гарантированно отличает общий случай от базового. Таким образом вторая функция является недвусмысленной перегрузкой базового случая! Можно заметить, что решение через функции выглядит куда компактнее аналогичного решения посредством классов.

Без рекурсии

Помимо рекурсивного метода развёртывания пачки существуют и другие методы. По крайней мере два из них я опишу ниже(на момент написания статьи мне не известны другие методы). Начнём с метода, который вы уже могли наблюдать в первом параграфе. Этот метод основан на использовании *std::initializer_list*:

```
template <bool... Bits>
std::string nonRecursiveBitsUnrolling()
{
    auto list = {Bits...};
    std::string bits;
    for(auto& bit : list)
        bits += boost::lexical_cast<std::string>(bit);
    return bits;
}

```

В основе метода лежит возможность распаковки пачки в контексте*initializer_list.* При этом над каждым аргументом, которые поступает в список можно выполнить произвольную операцию(как мы делали это в первом параграфе с **Values**). Удобно, компактно и никакой рекурсии.

Следующий метод использует возможность распаковки пачки в аргументах функции. Для начала определим вспомогательную функцию:

```
template<typename... Args>
void stub(Args&&...){}
```

Именно эта функция лежит в основе метода, т.к. именно она позволяет выполнить следующее:

```
template <bool... Bits>
std::string nonRecursiveBitsUnrolling2()
{
    std::string bits;
    stub((bits += recursiveFunBitsUnrolling<Bits>())...);
    return bits;
}

```

Каждое выражение **bits += recursiveFunBitsUnrolling<Bits>()** является аргументом функции **stub** и за счёт этого появляется возможность распаковки пачки! Еще одни замечательный метод, который, тем не менее, хорошо показывает недостаток текущего синтаксиса. Ведь было бы намного лучше, если бы можно было просто написать:

```
template <bool... Bits>
std::string nonRecursiveBitsUnrolling2()
{
    std::string bits;
    (bits += recursiveFunBitsUnrolling<Bits>())...;
    return bits;
}

```

и оно бы вылилось в:

```
(bits += recursiveFunBitsUnrolling<Bit1>()),
(bits += recursiveFunBitsUnrolling<Bit2>()) ;
return bits;

```

где порядок выражений был бы чётко определён. Но, увы, имеем то, что имеем – приходится использовать функции а-ля **stub** и помнить о том, что аргументы функции вычисляются в произвольном порядке.

На этом я бы хотел завершить повествование о шаблонах с переменным количеством параметров и перейти к

Полезные мелочи

Здесь я опишу нововведения в шаблонах, которые нельзя даже сравнивать с variadic templates ни по популярности, ни по предоставляемым возможностям. Тем не менее они крайне важны и заслуживают упоминания. Эдакая работа над ошибками в части шаблонов.

typedef 2.0

Полагаю, что многие из вас хоть раз испытывали нужду в написании подобного кода:

```
template<typename T>
typedef std::vector<T> MyVector_t;
```

Но, к сожалению, это невозможно в C++ старого пошива и люди, как водится, открыли обходной путь:

```
template<typename T>
struct Stub
{
    typedef std::vector<T> MyVector_t;
};

Stub<int>::MyVector vector;

```

Хотя этот метод и работает, но его явно нельзя считать решением, т.к. для простейшей и, казалось бы, очевидной операции нам приходится заводить сущность, в которой мы не нуждаемся. Комитет по стандартизации решил не играть с *typedef* и представил старое ключевое слово *using*, в новом свете. Теперь мы можем использовать *using* в контексте задания псевдонимов для имён. Так, вышеописанный, желаемый код, будет выглядеть в C++11:

```
#include <vector>
template<typename T>
using MyVector_t = std::vector<T>;

int main ()
{
    MyVector_t<double> vec;
    return 0;
}

```

Мне кажется, что решения комитета о неиспользование *typedef* вполне понятно. Код с *using* выглядит более понятным и чистым. Но *using* решает не только задачу задания псевдонима шаблонному классу, но и полностью заменяет *typedef*:

```
using MyIntVector = std::vector<int>;
using PFun = void(*)(int);
typedef void(*PFun2)(int);
```

Не правда ли синтаксис задания псевдонима **PFun** выглядит лучше, чем для **PFun2**?

Таким образом, я считаю, что использование *typedef* более не целесообразно, и стоит использовать *using* вместо него.

extern templates

Рассмотрим следующую ситуацию: У нас есть заголовочный файл, назовём его Header.h:

```
template<typename T>
struct A
{
    void foo();

};
template<typename T>
void A<T>::foo()
{
    ...
}

```

Так же есть файл Source.cpp, который включает Header.h и инстанциирует шаблонный класс A<int>:

```
void boo()
{
    A<int> a;
    a.foo();
}

```

А еще у нас есть main.cpp, в котором, также , инстанциируется шаблонный класс A<int>:

```
int main()
{
    A<int> b;
    b.foo();
}

```

Если мы скомпилируем этот проект, то на выходе получим два объектных файла: **Source.obj** и **main.obj**. При этом, кроме всего прочего они оба будут содержать в себе копию **void A<T>::foo()**, т.к. инстанциация шаблона происходит в *каждом* модуле трансляции, где он используется. Только на этапе линковки будут устранены дубликаты и останется только одна копия. Таким образом мы имеем в наличие лишнюю работу как компилятора, так и линковщика.

Данную проблему призвано решить ключевое слово *extern*(не путать с*export*!). Как и в других случаях применения *extern* смысл его применения с шаблонами кристально ясен – оно указывает компилятору, что данный шаблон будет инстанциирован где-то в другом месте, и не нужно инстанциировать его в данном модуле. Поэтому, если мы изменим Source.cpp следующим образом:

```
extern template A<int>;

```

```
void boo()
{
    A<int> a;
    a.foo();
}

```

то **void A<T>::foo()** будет содержаться только в объектном файле**main.obj**.

Восстановление прав шаблонных функций

В C++03 функции были подвержены дискриминации и не могли содержать аргументы по умолчанию в параметрах шаблона. С появлением C++11 функции также могут содержать аргументы по умолчанию:

```
template<typename T, typename U = float>
U foo(T value)
{
   return value;
}

```

Восстановление прав локальных типов

В C++03 аргументами шаблона могли выступать лишь типы объявленные вне функции. C++11 убирает это недоразумение:

```
void foo()
{
    struct MyFunctor
    {
       ...
    };
    std::vector<int> vec;
    ...//заполняем вектор
    //Следующая строчка возможна только в C++11
    std::for_each(std::begin(vec), std::end(vec), MyFunctor());
}


```