**Шаблоны C++.**

Пусть имеется некоторый шаблон структуры:

template<int Position, typename T1 = NullType, typename T2 = NullType, typename T3 = NullType, typename T4 = NullType> struct MakePinList

{

// …

};

Где NullType – это пустой класс, так называемый маркер конца списка:

class NullType{};

Шаблон MakePinList имеет 5 параметров, среди них Position – это параметр значения, а T1…T4 – параметры типов. Для параметров значения в шаблонах должен быть указан тип этих значений, а для параметров типов должно быть использовано ключевое слово typename.

Процесс подстановки конкретных типов в шаблон при его использовании в коде называется инстанцированием шаблона, то есть определение параметров шаблона конкретными аргументами так, чтобы получился класс, готовый к компиляции. При инстанцировании шаблона аргументы подставляются в угловые скобочки сразу после имени шаблона. Инстанс шаблона – это готовый к компиляции класс, со всеми определенными аргументами.

Пример инстанцирования вышеприведённого шаблона:

MakePinList<0, PA0, PB1, PC1, PC2> pinlist;

Мы подставили все аргументы для данного шаблона. Но так как все параметры типов шаблона имеют их значения по умолчанию NullType, можно определить аргументы выборочно:

MakePinList<0, PA0, PB1> pinlist;

Здесь мы определили только параметры Pos, T1, T2, в то время как параметры T3 и T4 остались в значениях по-умолчанию NullType. Следует иметь ввиду, что порядок указания аргументов в шаблоне должен строго соответствовать порядку задания его параметров. Например, следующая запись не скомпилируется:

MakePinList<PA0, PB1, 0> pinlist;

Компилятор сообщит нам о неправильных аргументах шаблона, т.к. вместо значения int в первом аргументе подставлен тип PA0.

Вернёмся к нашему примеру списка пинов. Так как все параметры значения шаблона известны на этапе компиляции, мы можем использовать их для определения значений членов перечислений или размера массивов:

template<int Pos = 0, typename T1 = NullType, typename T2 = NullType, typename T3 = NullType, typename T4 = NullType>

struct MakePinList

{

enum{PositionInList = Pos};

};

Теперь наша структура имеет константу перечисления PositionInList, значение определяется параметром шаблона Pos.

**Специализация шаблонов.**

В C++ имеется возможность определять частные полные или частичные специализации шаблона для структур и классов и полные специализации шаблона для методов и функций. Специализация шаблона это отдельный шаблон, поведение которого для одного или нескольких конкретных аргументов специализируется явно. Если шаблон специализации имеет свои собственные шаблонные параметры, используемые для инстанцирования специализирующего аргумента, такая специализация называется частичной.

Например, мы можем специализировать шаблон MakePinList для значения параметра Position = 0, и в этой специализации мы можем убрать перечисление:

template<typename T1, typename T2, typename T3, typename T4>

struct MakePinList<0, T1, T2, T3, T4>

{

};

Таким образом, для аргумента Pos = 0, следующий код не скомпилируется, так как компилятор всегда выберет частную специализацию вместо общего шаблона, если таковая имеется:

MakePinList<0, PA0, PB1> pinlist;

std::cout << "pos = " << pinlist.PositionInList << std::endl;

Следует обратить внимание на то, что специализируемый параметр не указывается в списке параметров template, а также на то, что при определении специализации шаблона не допускается указывать значения по-умолчанию для параметров, для которых уже были заданы значения по-умолчанию в общем шаблоне.

Далее рассмотрим пример частичной специализации шаблона. Здесь трюк состоит в том, что аргумент задан не полностью, а в виде шаблона. Синтаксически это может выглядеть достаточно сложно, но специализацию шаблона всегда можно опознать по наличию специализирующего (хоть и шаблонизированного) аргумента, стоящего в <> после имени шаблона.

Общий шаблон, никогда не инстанцируется:

template<typename T> struct s;

Некоторый шаблонный тип, который мы будем использовать в качестве специализирующего аргумента:

template<int a, int b> struct c

{

enum {SUM = a + b};

};

Шаблон специализации шаблона s:

template <int a, int b> struct s<c<a, b>>

{

enum {VALUE = c<a, b>::SUM};

};

И, наконец, инстанцирование специализации:

std::cout << "VALUE = " << s<c<5,3>>::VALUE << std::endl;

NB: существует двоякое понимание термина «частичная специализация». С одной стороны, это может быть спциализация по неполному списку параметров, а с другой стороны – специализация шаблонным аргументом. Какое из этих двух определений более верное – в этом еще предстоит разобраться.

Как видим, появление новых параметров шаблона для структуры s поначалу вводит в непонимание, но наличие треугольных скобочек после неё свидетельствует о задании специализации, а значит параметры шаблона относятся не к структуре, а к ее специализирующему шаблону.

**Генерация инстансов шаблона.**

Инстанс шаблона может быть не только явно прописан вручную в коде, но также может быть сгенерирован компилятором. Как правило, такая генерация имеет место при рекурсивном обращении к шаблонам с модифицированным параметром. Как и для всякой рекурсии, необходимо задание условия достижения дна рекурсии, и в качестве такого условия используется частная специализация шаблона с граничным значением параметра шаблона.

Введём в нашу структуру некий тип, который принадлежит инстансу шаблона собственной структуры, инстанцированный модифицированным параметром Pos, и аргументами инстанса базового шаблона, которые передаются в этот новый инстанс со смещением на один параметр:

template<int Pos = 0, typename T1 = NullType, typename T2 = NullType, typename T3 = NullType, typename T4 = NullType>

struct MakePinList

{

enum{PositionInList = Pos};

typedef typename MakePinList<Pos+1, T2, T3, T4>::Result Result;

};

Инстанс базового шаблона по прежнему имеет вид:

MakePinList<0, PA0, PB1, PC1, PC2>

Когда компилятор встретит typedef, он сгенерирует новый инстанс:

MakePinList<1, PB1, PC1, PC2, NullType>

Последний параметр шаблона останется в значении по-умолчанию (NullType).

Далее компилятор попытается найти тип Result внутри сгенерированного инстанса, но это приведет к еще одной генерации инстанса:

MakePinList<2, PC1, PC2, NullType, NullType>

Таким образом, мы наблюдаем рекурсивную генерацию инстансов, при этом в каждом новом инстансе значение аргумента Pos увеличивается на единицу. Чтобы остановить рекурсию, мы должны задать её граничный случай, т.е. граничный инстанс, который должен быть явно специализирован. Известно, что компилятор будет пытаться в первую очередь использовать явные специализации шаблонов, а не общий шаблон. Пример такого граничного случая - специализация шаблона всеми аргументами типов NullType:

template<int Pos>

struct MakePinList<Pos/\*, NullType, NullType, NullType, NullType\*/>

{

typedef NullType Result;

enum{PositionInList = Pos};

};

У нас остался только один параметр-значение в данной специализации, все остальные параметры-типы заданы как NullType. Так как NullType является значением по-умолчанию для всех параметров-типов, то их можно не указывать. Чтобы остановить рекурсию, в данной специализации тип Result определён непосредственно.

Таким образом, мы сгенерировали слудующие инстансы:

MakePinList<0, PA0, PB1, PC1, PC2>

MakePinList<1, PB1, PC1, PC2, NullType >

MakePinList<2, PC1, PC2, NullType, NullType

MakePinList<3, PC2, NullType, NullType, NullType>

MakePinList<4, NullType, NullType, NullType, NullType>

Последний инстанс подчиняется специализации шаблона и является вырожденным случаем.

Теперь мы можем объединить список всех инстансов, генерируемых в примере выше, в один единственный тип:

typedef MakePinList<0, PA0, PB1, PC1, PC2>::Result PinList\_c;

Который может быть передан в другой шаблон с одним параметром.

**Списки типов (до C++11).**

В программировании на C++ до стандарта С++11 было очень мало инструментов метапрограммирования, по крайней мере стандартизованных. Большой вклад в развитие метапрограммной магии сделал американский ученый Андрей Александреску. Он изложил результаты своих трудов в библиотеке Loki, которую описал в своей книге «Современное проектирование на C++: Обобщенное программирование и прикладные шаблоны проектирования».

Для организации списка типов в библиотеке Loki используется шаблон:

template <class T, class U>

struct Typelist

{

typedef T Head;

typedef U Tail;

};

Суть идеи заключается в том, что во второй параметр Typelist можно передать другой Typelist, во второй парметр которого можно передать следующий Typelist, и так пока не будут переданы все типы. Пример:

typedef Typelist<char, Typelist<signed char, unsigned char>> Chars;

Видоизменим наш шаблон MakePinList так, чтобы тип Result был синонимом шаблона Typelist, в этом случае мы сможем получить список типов на основе Typelist, который можно будет передавать далее в другие шаблоны и вот каким образом:

Откушенную «голову» T1 мы используем для первого параметра Typelist, а во второй подставим Typelist следующего инстанса. Далее в следующем инстансе в качестве первого параметра Typelist будет использован T2, а второго – Typelist следующего третьего инстанса и т.д.

template<int Pos, typename T1 = NullType, typename T2 = NullType, typename T3 = NullType, typename T4 = NullType>

struct MakePinList

{

private:

enum{PositionInList = Pos};

typedef typename MakePinList<Pos + 1, T2, T3, T4>::Result TailResult;

public:

typedef Typelist<PW<T1, PositionInList>, TailResult> Result;

};

Мы снова заставляем компилятор рекурсивно генерировать инстансы, тип TailResult каждого из которых соответствует типу Result последующего инстанса, который, в свою очередь, является типом Typelist. В случае списка из четырёх типов получим:

Typelist<T1, Typelist <T2, Typelist <T3, Typelist<T4, Nulltype>>>>.

Далее рассмотрим способ обработки списка типов на основе Typelist.

Зададим общий шаблон клиентского класса GetPorts:

template <class TList> struct GetPorts;

В качестве аргумента шаблона ему предполагается передача только списка типов, для которого будет записана явная частная специализация. Следовательно, общий шаблон никогда не инстанцируется, поэтому он не имеет тела.

Запишем специализацию для аргумента типа Typelist:

template <class Head, class Tail>

struct GetPorts< Typelist<Head, Tail> >

{

private:

// запоминаем порт, к которому относится пин из PW

typedef typename Head::pin::Port Port; // Head = PW

// требуется для рекурсивной генерации списка

typedef typename GetPorts<Tail>::Result TailResult;

public:

typedef Typelist<Port, L1> TailResult;

};

template <> struct GetPorts<NullType>

{

typedef NullType Result;

};

Здесь в каждом последующем генерируемом инстансе сохраняется порт, извлекаемый из типа, принадлежащего аргументу Typelist. Сохраняем список портов в новом типе:

typedef GetPorts<PinList>::Result PortList;

Далее будем обрабатывать уже список портов, удаляя из него повторяющиеся типы.

**Списки типов (начиная с C++11)**.

Общий шаблон имеет вид:

template<typename … Args> struct TypeList

{

typedef internal::Void Head;

typedef internal::Void Tail;

};

Если шаблон TypeList инстанцирован пустым пакетом параметров, то такой список является пустым:

typedef TypeList<> EmptyTypeList;

Частичная специализация для списка, как минимум, из одного элемента:

template<typename H, typename … T> struct TypeList<H, T…>

{

// голова отдельно

typedef H Head;

// все остальное в хвост

typedef TypeList<T…> Tail;

};

NB: я напоминаю, что если записанный инстанс матчится и с общим шаблоном, и со специализацией, компилятор выберет именно специализацию.

Определение факта пустого списка (IsEmpty).

Общий шаблон:

template <typename TL> struct IsEmpty : std::true\_type {};

Специализация для аргумента-пустого списка:

template <> struct IsEmpty<TypeList<internal::Void, internal::Void>> : std::true\_type;

**Частичная** специализация для аргумента-списка:

**template <typename … Args>** struct IsEmpty<TypeList<Args…>> :

std::integral\_constant<

bool,

std::is\_same<typename TypeList<Args…>::Head, internal::Void>::value &&

IsEmpty<typename TypeList<Args…>::Tail>::value

>

{

};

Список пуст, если его голова — это вспомогательный тип, обозначающий void, и если его хвост также является пустым списком. Как видно, здесь используется рекурсия, которую, как раз и останавливает полная специализация шаблона для пустого списка.

**Удаление дублирующихся типов из списка.**

Имеется шаблонный класс zipper<Args…>, который принимает пакет параметров Args. Zipper образует иерархию наследования, состояющую из инстансов zipper, аргументами которых являются типы инстансов process\_zipper\_arguments<…>::type.

template <typename … Args> struct zipper;

Таким образом, пакет параметров Args шаблона zipper раскрывается и передаётся в другой шаблон process\_zipper\_arguments, который и занимается обработкой типов из пакета.

Две вспомогательные структуры, индицирующие true и false:

struct true\_type {};

struct false\_type{};

Вспомогательная структура, назначение которой мне пока не понятно.

template <typename … Types> struct typelist{};

Определение факта наличия элемента в списке.

Шаблон структуры, индицирующий наличие элемента в списке typelist. Его тип type является true\_type, если тип присутствует в списке типов, или false\_type, если тип не присутствует в списке типов.

Полагаем, что TElement не содержится в списке, если не доказано иное. Тогда общий шаблон имеет вид:

template <typename TElement, typename TList> struct is\_in {

typedef false\_type type;

};

Специализация по параметру TList: будет выбрана, если в качестве аргумента будет передан typelist<TElement, TTail…> (то есть если тип TElement, указанный при иснтанцировании, совпадёт с одним из элементов typelist, при рекурсивной обработке).

template<typename TElement, typename … TTail>

struct is\_in<TElement, typelist <TElement, TTail…> {

typedef true\_type type;

};

Специализация по параметру TList: рекурсивная обработка хвоста.

template < typename TElement, typename THead, typename... TTail >

struct is\_in < TElement, typelist < THead, TTail... > >

{

typedef typename is\_in < TElement, typelist < TTail... > >::type type;

};

Добавление уникального элемента в список.

Шаблон модифицирует скармливаемый ему список typelist, добавляя к нему новый элемент TNew, или оставляет без изменения, в зависимости от того, какую специализацию выберет компилятор. Компилятор, в свою очередь, выберт специализацию, в зависимости от получившегося инстанса is\_in: если он даст тип true\_type, значит такой элемент в списке уже есть и type для add\_unique будет соответствовать изначальному typelist, а если он даст тип false\_type, значит такого элемента нет, и type для add\_unique будет уже с новым элементом TNew.

Общий шаблон имеет вид:

template<

typename TNew, typename TList,

typename Tis\_duplicate = typename is\_in<TNew, TList>::type

> struct add\_unique;

Специализация по параметру TList, которую компилятор выберет в случае наличия элемента в списке:

template < typename TNew, typename... TList >

struct add\_unique < TNew, typelist < TList... >, true\_type >

{

typedef typelist < TList... > type;

};

Специализация по параметру TList, которую компилятор выберет в случае отсутствия элемента в списке:

template < typename TNew, typename... TList >

struct add\_unique < TNew, typelist < TList... >, false\_type >

{

typedef typelist < TNew, TList... > type;

};

NB: многие программисты, говоря о метапрограммных конструкциях, назвыют шаблоны – функциями, а возвращаемые значения – оператор typedef здесь\_что\_то обрабатывается type.