**Техника идеальной передачи.**

*Осмысленный материал по статье* <https://habrahabr.ru/post/242639/>.

Пусть имеется некоторый класс myclass, и вектор из элементов объектов этого класса.

class myclass {

public:

myclass(int ii\_, float ff\_) { … }

private:

// …

};

std::vector<myclass> v;

Классический способ помещения объекта myclass в вектор v:

v.push\_back(myclass(2, 3.14f);

Такой способ не самый эффективный: создается анонимный объект, затем работает конструктор копирования для создания объекта в векторе и затем анонимный объект уничтожается.

В стандарте c++11 появилась возможность создать объект непосредственно в векторе. Для этого был введен новый метод emplace\_back:

v.emplace\_back(2, 3.14f);

Для этого выражения выполнится только один конструктор, который и создает объект внутри вектора без использования временных объектов. Такое поведение стало возможным благодаря двум нововведениям C++11:

* **Вариативные шаблоны** (variadic templates)
* **Идеальная передача** (perfect forwarding)

Возможно ли реализовать подобный механизм идеальной передачи без этих нововведений? По сути emplace\_back делегирует вызов конструктору объекта myclass, не создавая временную копию его параметра. Тип объекта заранее неизвестен (вектор это шаблон). Попытаемся воспроизвести это – то есть создать такую функцию, которая бы вызвала другую функцию, не создавая временных копий аргументов, при этом тип этих самых аргументов заранее не известен (функция является шаблонной). Рассмотрим простой случай для двух аргументов. Решение в «лоб»:

template<typename T1, typename T2>

void wrapper(T1 e1, T2 e2)

{

func(e1, e2);

}

Очевидно, не годится, так как в этом случае создаются временные копии e1 и e2 для wrapper. Вдобавок, если func как-то по замыслу должна их изменять, т.е. принимает эти аргументы по неконстантной ссылке, ничего не выйдет, потому что она будет работать с копиями аргументов wrapper. Хорошо, того передадим аргументы во wrapper по ссылке:

template<typename T1, typenameT2>

void wrapper(T1& e1, T2& e2)

{

func(e1, e2);

}

Тоже не годится. Хотя теперь мы передаем аргументы во wrapper по ссылке, и никаких временных объектов не создается, это не будет работать для аргументов rvalue. Например, мы не можем сделать такой вызов:

wrapper(2, 3.14f);

Компилятор выругается при попытке инициализировать неконстантную ссылку rvalue значением. Хорошо, сделаем её константной? – не годится, так как func может как-то осмысленно менять передаваемые ей аргументы. Тем не менее, выход есть. Можно создать 4 перегрузки функции на все случаи жизни. Теперь можно передавать и rvalue и lvalue как угодно – компилятор подберет нужную версию перегрузки.

template <typename T1, typename T2>

void wrapper(T1& e1, T2& e2) { func(e1, e2); }

template <typename T1, typename T2>

void wrapper(const T1& e1, T2& e2) { func(e1, e2); }

template <typename T1, typename T2>

void wrapper(T1& e1, const T2& e2) { func(e1, e2); }

template <typename T1, typename T2>

void wrapper(const T1& e1, const T2& e2) { func(e1, e2); }

Но какой ценой – куча перегрузок и это только для функции двух параметров. А если их будет 3? 5? Количество перегрузок растет по экспоненте. И хотя такое решение работает, на практике едва ли подходит.

Поэтому в C++11 был введен специальный механизм, позволяющий реализовать идеальную передачу. При этом в язык программирования были добавлены два новых правила:

* **Сжатие ссылок** (reference collapsing).
* **Правила особого вывода типа** (special type deduction rules).

Сжатие ссылок.

Известно, что в C++ не допускается создавать ссылку на ссылку. Но бывают такие случаи, когда компилятору поневоле приходится это «проглатывать». Рассмотрим такой случай:

template<typename T>

void baz(T t)

{

T& k = t;

}

Если при инстанцировании шаблона мы укажем какой-то тип, у нас нет никаких проблем. А если мы укажем ссылку, например int&? Что получим тогда при интстанцировании?

int ii = 4;

baz<int&>(ii); // int& & k = 4;

Так как int& & - это запрещенная конструкция, компилятор просто преобразует это в обычную ссылку. И до C++11 такое поведение не было стандартизованным, но многие компиляторы обрабатывали такой код именно так, так как такие ситуации часто встречаются в метапрограммировании. После того, как в C++ были добавлены rvalue-ссылки, стало важным определить поведение при совмещении различных типов ссылок (например, как разрешить ситуацию int&& &)? Так появилось правило сжатия ссылок. Это правило очень простое: одиночный амперсанд (&) всегда побеждает. Таким образом, (& & -> &, && & -> &, & && -> &, но && && -> &&).

Правила особого вывода типа.

Для rvalue-ссылок в различных ситуациях. Рассмотрим пример шаблонной функции:

template<class T>

void func(T&& t)

{

…  
}

В этой шаблонной функции двойной амперсанд (&&) вовсе не означает rvalue-ссылку! В данной ситуации работают правила особого вывода типа:

* Если аргумент является lvalue типа U, то T становится U&.
* Если аргумент является rvalue типа U, то T становится U.

Примеры:

func(4); // 4 это rvalue, значит, T становится int

double d = 3.14;

func(d); // d это lvalue, значит, T становится double&

float f() {…}

func(f()); // f() это rvalue, значит, T становится float

int bar(int i)

{

func(i); // i это lvalue, значит, T становится int&

}

Это правило необходимо для решения проблемы идеальной передачи.

**Реализация идеальной передачи с использованием std::forward.**

Вернемся к нашей шаблонной функции wrapper. Вот как она должна быть реализована с использованием C++11:

template<typename T1, typename T2>

void wrapper(T1&& e1, T2&& e2)

{

func(forward<T1>(e1), forward<T2>(e2));

}

Сразу бросаются в глаза параметры функции с типами T1&& и T2&& – с учетом рассмотренных правил особого вывода типа, фактически эти типы представляют либо ссылку (в случае передачи lvalue), либо значение (в случае передачи rvalue); Благодаря этим правилом корректно работает преобразование аргумента (см. далее), передаваемого в func. Вот как реализован forward:

template<class T>

T&& forward(typename std::remove\_reference<T>::type& t) noexcept

{

return static\_cast<T&&>(t);

}

Посмотрим, что будет происходить при инстанцировании шаблона.

Если во wrapper будет передано lvalue типа int, тогда T1 представляет ссылку int&. Соответственно, инстанс forward<int&>(e1) будет иметь реализацию:

int& && forward(typename std::remove\_reference<int&>::type& t) noexcept

{

return static\_cast<int& &&>(t);

}

Для разрешения этой кучи амперсандов начинает работать правило сжатия ссылок:

int& forward(typename std::remove\_reference<int&>::type& t) noexcept

{

return static\_cast<int&>(t);

}

Таким образом, если передается аргумент lvalue, то производится передача по ссылке, но с этим у нас и не было проблем. Проблемы появились при попытке передать rvalue – мы были вынуждены создавать большое количество перегрузок на все случаи жизни. Посмотрим теперь, что произойдет при передаче rvalue во wrapper, реализованный по стандарту C++11.

Если во wrapper будет передано rvalue типа int, тогда T1 представляет значение int. Соответственно, инстанс

forward<int>(e1) будет иметь реализацию:

int &&forward(typename std::remove\_reference<T>::type& t) noexcept

{

return static\_cast<int&&>(t);

}

Таким образом, если передается аргумент rvalue, то производится передача по ссылке на rvalue, что и требовалось для решения задачи идеальной передачи. Имеет смысл добавить, что func должна объявлять свои параметры как ссылки на rvalue (с двойным амперсандом) – такие параметры допускают передачу в них как ссылок lvalue, так и rvalue.

Теперь wrapper является одним шаблоном, который обработает любые сочетания типов аргументов.

Шаблонная функция **forward** по сути является оберткой над оператором приведения типов **static\_cast<T&&>(t)**, которая инстанцируется правильным образом для lvalue и rvalue аргументов, благодаря **правилам особого вывода типа** и **правилу сжатия ссылок** и возвращает lvalue-ссылку для lvalue аргумента и rvalue-ссылку для rvalue-аргумента. Функция forward реализована в C++11, в заголовочном файле «utility», в пространстве имен std.

Замечание про