Концепты для отчаявшихся

Программирование*, C++*

Всё началось с того, что мне понадобилось написать функцию, принимающую на себя владение произвольным объектом. Казалось бы, что может быть проще:

```
template <typename T>
void f (T t)
{
    // Завладели экземпляром `t` типа `T`.
    ...

    // Хочешь — переноси.
    g(std::move(t));

    // Не хочешь — не переноси.
    ...
}
```

Но есть один нюанс: требуется, чтобы принимаемый объект был строго `rvalue`. Следовательно, нужно:

1. Сообщать об ошибке компиляции при попытке передать `lvalue`.
2. Избежать лишнего вызова конструктора при создании объекта на стеке.

А вот это уже сложнее сделать.

Поясню.

Требования к входным аргументам

Допустим, мы хотим обратного, то есть чтобы функция принимала только`lvalue` и не компилировалась, если ей на вход подаётся `rvalue`. Для этого в языке присутствует специальный синтаксис:

```
template <typename T>
void f (T & t);
```

Такая запись означает, что функция `f` принимает `lvalue`-ссылку на объект типа`T`. При этом заранее не оговариваются `cv`-квалификаторы. Это может быть и ссылка на константу, и ссылка на неконстанту, и любые другие варианты.

Но ссылкой на `rvalue` она быть не может: если передать в функцию `f` ссылку на `rvalue`, то программа не скомпилируется:

```
template <typename T>
void f (T &) {}

int main ()
{
    auto x = 1;
    f(x); // Всё хорошо, T = int.

    const auto y = 2;
    f(y); // Всё хорошо, T = const int.

    f(6.1); // Ошибка компиляции.
}
```

Может, есть синтаксис и для обратного случая, когда нужно принимать только`rvalue` и сообщать об ошибке при передаче `lvalue`?

К сожалению, нет.

Единственная возможность принять `rvalue`-ссылку на произвольный объект — это *сквозная ссылка* (`forwarding reference`):

```
template <typename T>
void f (T && t);
```

Но сквозная ссылка может быть ссылкой как на `rvalue`, так и на `lvalue`. Следовательно, нужного эффекта мы пока не добились.

Добиться нужного эффекта можно при помощи механизма `SFINAE`, но он достаточно громоздкий и неудобный как для написания, так и для чтения:

```
#include <type_traits>

template <typename T,
    typename = std::enable_if_t<std::is_rvalue_reference<T &&>::value>>
void f (T &&) {}

int main ()
{
    auto x = 1;
    f(x); // Ошибка компиляции.
```

```
    f(std::move(x)); // Всё хорошо.

    f(6.1); // Всё хорошо.
}
```

А чего бы на самом деле хотелось?

Хотелось бы вот такой записи:

```
template <typename T>
void f (rvalue<T> t);
```

Думаю, смысл данной записи выражен достаточно чётко: принять произвольное `rvalue`.

Первая мысль, которая приходит в голову, — это создать псевдоним типа:

```
template <typename T>
using rvalue = T &&;
```

Но такая штука, к несчастью, не сработает, потому что подстановка псевдонима происходит *до* вывода типа шаблона, поэтому в данной ситуации запись`rvalue<T>` в аргументах функции полностью эквивалентна записи `T &&`.

Скрытый текст

Ещё одна идея — по сути, аналогичная, — которая может прийти в голову знатоку шаблонного метапрограммирования — это написать следующий код:

```
template <typename T>
struct rvalue_t
{
    using type = T &&;
};

template <typename T>
using rvalue = typename rvalue_t<T>::type;
```

К структуре `rvalue_t` можно было бы припилить `SFINAE`, которое отваливалось бы, если бы `T` было ссылкой на `lvalue`.

Но, к сожалению, эта идея также обречена на провал, потому что такая структура "ломает" механизм вывода типов. В результате функцию `f` вообще будет невозможно вызвать без явного

указания аргумента шаблона.

Я очень расстроился и на время забросил эту идею.

Возвращение

В начале этого года, когда появилась новость о том, что комитет не включил концепты в стандарт C++17, я решил вернуться к заброшенной идее.

Немного поразмыслив, я сформулировал "требования":

1. Должен работать механизм вывода типа.
2. Должна быть возможность натравливать `SFINAE`-проверки на выводимый тип.

Из первого требования немедленно следует, что нужно всё-таки использовать псевдонимы типов.

Тогда возникает закономерный вопрос: можно ли натравливать `SFINAE` на псевдонимы типов?

Оказывается, можно. И выглядеть это будет, например, следующим образом:

```
template <typename T,
    typename = std::enable_if_t<std::is_rvalue_reference<T &&>::value>>
using rvalue = T &&;
```

Наконец-то получаем и требуемый интерфейс, и требуемое поведение:

```
template <typename T>
void f (rvalue<T>) {}

int main ()
{
    auto x = 1;
    f(x); // Ошибка компиляции.

    f(std::move(x)); // Всё хорошо.

    f(6.1); // Всё хорошо.
}
```

Победа.

Концепты

Внимательный читатель негодует: "Так где же тут концепты-то?".

Но если он не только внимательный, но ещё и сообразительный, то быстро поймёт, что эту идею можно использовать и для "концептов". Например, следующим образом:

```
template <typename I,
    typename = std::enable_if_t<std::is_integral<I>::value>>
using Integer = I;

template <typename I>
void g (Integer<I> t);
```

Мы создали функцию, которая принимает только целочисленные аргументы. При этом получившийся синтаксис достаточно приятен и пишущему, и читающему.

```
int main ()
{
    g(1); // Всё хорошо.
    g(1.2); // Ошибка компиляции.
}
```

Что ещё можно сделать?

Можно попытаться ещё больше приблизиться к истинному синтаксису концептов, который должен выглядеть следующим образом:

```
template <Integer I>
void g (I n);
```

Для этого воспользуемся, кхм, макроснёй:

```
#define Integer(I) typename I, typename = Integer<I>
```

Получим возможность писать следующий код:

```
template <Integer(I)>
void g (I n);
```

На этом возможности данной техники, пожалуй, заканчиваются.

Недостатки

Если вспомнить название статьи, то можно подумать, что у этой техники есть какие-то недостатки.

Таки да. Есть.

Во-первых, она не позволяет организовать перегрузку по концептам.
Компилятор не увидит разницы между сигнатурами функций

```
template <typename I>
void g (Integer<I>) {}

template <typename I>
void g (Floating<I>) {}
```

и будет выдавать ошибку о переопределении функции `g`.

Во-вторых, невозможно одновременно проверить несколько свойств одного типа. Вернее, возможно, но придётся городить достаточно сложные конструкции, которые сведут на нет всю удобочитаемость.

Выводы

Приведённая техника — назовём её техникой *фильтрующего псевдонима типов*— имеет достаточно ограниченную область применения.

Но в тех случаях, когда она применима, она открывает программисту достаточно неплохие возможности для чёткого выражения намерения в коде.

Считаю, что она имеет право на жизнь. Лично я пользуюсь. И не жалею.

Ссылки по теме

1. Библиотека "Boost Concept Check"
2. Концепты из прототипа библиотеки диапазонов "range-v3"
3. Библиотека "TICK"
4. Статья "Concepts Without Concepts"