Беды с шаблонами в C++

Добрый день, меня, напомню, зовут Данила Григорьев и тема нашей сегодняшней встречи — магия шаблонов в C++. Но прежде чем перейти к этой весьма конкретной теме, я предлагаю поговорить о более абстрактных вещах.

Введение

Метапрограммирование — это парадигма, при которой мы пишим не непосредственно программу, а, если так можно выразиться, генератор программы — конечный код будет написан не нами и не нейросетью, а сгенерирован в процессе компиляции или даже во время выполнения.

На этом этапе большой дом метапрограммирования делится на два подъезда — генерация и самомодификация.

Самомодификация зачастую означает, что программа будет изменять свой код или взаимодействовать со внутренними структурами языка прямо во время выполнения. Сюда можно подвести и функцию `eval` в скриптах, и рефлексию, и другие техники.

Генерация же происходит на этапе компиляции. Согласно заданным нами инструкциям или даже автоматически компилятор самостоятельно вычисляет то, что можно вычислить, и генерирует варианты кода для разных типов параметров или даже всю программу целиком.

Самый простой пример генерации — это когда компилятор вместо вычисления `2+2` в конечный код будет подставляет `4`. Ведь зачем тратить время выполнения на выполнение тех операций, результат которых заведомо известен? Такие примитивные оптимизации являются частным случаем метапрограммирования: по сути мы не подставили конкретное значение (`4`), а невольно описали компилятору способ его получения (выражение `2+2`).

Гораздо более интересный случай — когда для одного абстрактного блока кода компилятор генерирует несколько конкретных вариантов — частных случаев, адаптированных под конкретные ситуации. Это **макросы** и **шаблоны**.

Макросы — это синтаксические конструкции, которые разворачиваются в последовательности инструкций.

Разговаривая о макросах, я люблю приводить весьма наглядный пример на Расте, где, к слову, макросы обозначаются восклицательным знаком.

```rust
let v: Vec<u32> = vec![1, 2, 3];
```



```rust
let v: Vec<u32> = {
  let mut tmp = Vec::new();
  tmp.push(1);
  tmp.push(2);
  tmp.push(3);
  tmp
};
```

Несколько иначе работают **шаблоны**, ради которых мы сегодня и собрались.

Шаблон задаёт способ построения частных случаев сущности (функции или структуры) из её более общего описания.

Звучит сложно, но не так страшен чёрт, как его малюют.

Базовый синтаксис

```
template<typename T>
struct Foo {
  T bar;
};
Foo<int> foo1 = Foo<int>();
Foo<std::string> foo2 =
Foo<std::string>();
```

```
struct Foo<int> {
  int bar;
};
struct Foo<std::string> {
  std::string bar;
};
Foo<int> foo1 = Foo<int>();
Foo<std::string> foo2 =
Foo<std::string>();
```

В приведённом коде мы объявили структуру `Foo`, а перед ней вписали префикс `template<typename T>`. Он означает, что мы не просто объявляем тип `Foo`, а описываем шаблон типа `Foo` с параметром `T`, причём под аргументом `T` мы обозначаем тип данных (об этом говорит ключевое слово `typename`).

Что происходит с точки зрения компилятора? Для него есть шаблон структуры `Foo`, на основе которого будут создаваться много разных структур `Foo` в зависимости от того, какие шаблонные аргументы мы предоставим. Фактически на данном этапе созадётся целых 0 структур — поскольку мы не знаем, какие значения `T` впоследствии встретятся.

Когда компилятор доходит до строк, где мы создаём экземпляры структур `Foo<int>` и `Foo<std::string>`, компилятор вспоминает про наш шаблон и **инстанцирует** (создаёт) две независимые структуры `Foo` для `T = int` и `T = std::string`. Они не являются родственными, они не полиморфны, не связаны ни наследованием, ничем — каждая инстанцированная структура полностью самодостаточна.

Когда мы обращаемся с шаблонными структурами или функциями, компилятор инстанцирует их автоматически — это называется **ленивым инстанцированием**. При желании мы можем выполнить **явное инстанцирование** с помощью такой конструкции:

```
template<typename T>
struct Foo {
  T bar;
};
template struct Foo<void*>();
Foo<int> foo1 = Foo<int>();
Foo<std::string> foo2 =
Foo<std::string>();
```

```
struct Foo<int> {
  int bar;
};
struct Foo<std::string> {
  std::string bar;
};
struct Foo<void*> {
  void* bar;
};
Foo<int> foo1 = Foo<int>();
Foo<std::string> foo2 =
Foo<std::string>();
```

Причём шаблонные параметры не всегда относятся к типу:

```
template<typename T, size_t N>
struct Foo {
  T bar[N];
};
auto foo = Foo<int, 69>();
```

```
struct Foo<int, 69> {
  int bar[69];
};
Foo<int, 69> foo = Foo<int, 69>();
```

Им можно задавать значения по умолчанию:

```
template<typename T = int, size_t N =
69>
```

```
struct Foo<int, 69> {
  int bar[69];
```

```
struct Foo {
  T bar[N];
};
Foo foo1 = Foo(); // то же, что и
Foo<int, 69>
Foo<bool, 1> foo2 = Foo<bool, 1>();
```

```
};
struct Foo<bool, 1> {
  bool bar[1];
};
Foo<int, 69> foo1 = Foo<int, 69>();
Foo<bool, 1> foo2 = Foo<bool, 1>();
```

А для частных случаев можно задавать особенные реализации благодаря специализациям шаблонов:

```
template<typename T, size_t N>
struct Foo {
  T bar[N];
};
// частичная специализация
template<size_t N>
struct Foo<char, N> {
  std::string bar = std::string(N);
};
// полная специализация
template<>
struct Foo<int32_t, 2> {
  int64_t bar;
};
Foo<bool, 3> foo1 = Foo<bool, 3>();
Foo<int32_t, 2> foo2 = Foo<int32_t,
2>();
Foo<char, 42> foo3 = Foo<char, 42>();
```

⇒

```
struct Foo<bool, 3> {
  bool bar[3];
};
struct Foo<int32_t, 2> {
  int64_t bar;
};
struct Foo<char, N> {
  std::string bar = std::string(N);
};
Foo<bool, 3> foo1 = Foo<bool, 3>();
Foo<int32_t, 2> foo2 = Foo<int32_t,
2>();
Foo<char, 42> foo3 = Foo<char, 42>();
```

Обратите внимание, порядок объявления шаблонов имеет значение:

```
// полная специализация
template<>
struct Foo<int32_t, 2> {
  int64_t bar;
};
// шаблон
template<typename T, size_t N>
struct Foo {
  T bar[N];
};
// частичная специализация
template<size_t N>
struct Foo<char, N> {
  std::string bar = std::string(N);
};
Foo<bool, 3> foo1 = Foo<bool, 3>();
Foo<int32_t, 2> foo2 = Foo<int32_t, 2>();
Foo<char, 42> foo3 = Foo<char, 42>();
```

2:8: error: ‘Foo’ is not a typename template

```
2 | struct Foo<int32_t, 2> {
  |        ^~~
  |                        ^
```

7:8: error: ‘Foo’ is not a template

```
7 | struct Foo {
  |        ^~~
```

2:8: note: previous declaration here

```
2 | struct Foo<int32_t, 2> {
  |        ^~~
```

12:8: error: ‘Foo’ is not a typename template

```
12 | struct Foo<char, N> {
   |        ^~~
```

12:18: error: ‘N’ was not declared in this scope

```
12 | struct Foo<char, N> {
   |                  ^
```

12:19: error: ‘Foo’ is not a template

```
12 | struct Foo<char, N> {
   |                   ^
```

```
2:8: note: previous declaration here
    2 | struct Foo<int32_t, 2> {
      |        ^~~
```

Первым всегда должен объявляться общий шаблон и лишь потом его специализации. Выбор подходящей специализации компилятор производит с учётом ещё одного, более важного нежели порядок объявления, критерия: чем меньше список аргументов специализации, тем она более узкая:

```
template<typename T> typename Vector;        // общий шаблон
template<typename T> typename Vector<T*>;    // специализация для любых указателей
template<>           typename Vector<void*>; // специализация для void*
```

В ситуации, если возникает неоднозначность и два независимых шаблона полностью подходят, ожидаемо происходит ошибка компиляции.

Беды с шаблонами

- Перегрузка шабона, эвристика работы компилятора
- Бесконечность не предел: пакеты параметров и сжатые выражения
- Шаблонные шаблоны
- ОП — НО: Почему ошибка подстановки — не ошибка?
- Условная компиляция
- Вычисления на этапе компиляции
- Метафункции
- Любопытно повторяющийся шаблонный шаблон
- Шаблонные переменные
- Зависимые имена

Перегрузка шаблона, эвристика работы компилятора

Загадка от Жака Фреско: что выведет этот код?

```
template<typename T, typename U>
void f(T, U)     { std::cout << 1; }

template<typename T>
void f(T, T)     { std::cout << 2; }

template<>
void f(int, int) { std::cout << 3; }

void f(int, int) { std::cout << 4; }

int main() {
  f(0, 0);
}
```

1. Сначала делается перегрузка между шаблонами
2. Из шаблона выбирается подходящая версия
3. В выбранный шаблон подставляются аргументы
4. Происходит перегрузка между получившимися функцями

Порядок объявлений специализацией может повлиять на то, кто чьей специализацией является.

Как следствие, в данном коде приоритеты определятся следующим образом: $4 \to 3 \to 2 \to 1$.

Пакеты параметров и сжатые выражения

Шаблонные шаблоны

```
template<typename T>
struct MyContainer;

template<typename T, template<typename> typename Container = MyContainer<T>>
struct Pod {
  Container<T> container;
};

Pod<int> pod = Pod<int>(); // T = int, Container = MyContainer<int>
```

Как можно заметить, в качестве второго параметра шаблона мы передали другой шаблон, принимающий аргументом единственный тип, и назвали его `Container`.

Ошибка подстановки — не ошибка (ОП,НО — SFINAE)

Условная компиляция

В C++11 появилось ключевое слово `constexpr`, которое позволяет гарантировать детерменированность конструкции на этапе компиляции.

Это секретный инструмент, который понадобится нам позже.

Вычисления на этапе компиляции

Я думаю, вы уже догадались, что весь изученный нами аппарат можно использовать в самых странных целях. На пациенте можно спокойно проводить вычисления на этапе компиляции и писать полноценные программы, выполняемые не выходящим бинарником, но самим компилятором.

Давайте научим компилятор считать числа Фибоначчи.

```
template<int N>
struct Fibonacci {
  static const int value = Fibonacci<N - 1>::value + Fibonacci<N - 2>::value;
};

std::cout << Fibonacci<20>::value;
```

Попытка скомпилировать приведёт к следующей ошибке:

```
In instantiation of ‘const int Fibonacci<-879>::value’:
recursively required from ‘const int Fibonacci<19>::value’
   static const int value = Fibonacci<N - 1>::value + Fibonacci<N - 2>::value;
                                                  ^~~~~
required from ‘const int Fibonacci<20>::value’
required from here
   std::cout << Fibonacci<20>::value;
                              ^~~~~
fatal error: template instantiation depth exceeds maximum of 900 (use ‘-ftemplate-
depth=’ to increase the maximum)
   static const int value = Fibonacci<N - 1>::value + Fibonacci<N - 2>::value;
                                                  ^~~~~
compilation terminated.
```

Дело в том, что, разворачивая написанный нами код и доходя до `Fibonacci<-879>`, компилятор определяет превышение максимальной глубины рекурсии для шаблонов. Как это пофиксить? Мы можем компилировать с флагом `-ftemplate-depth=`. Но, введя `g++ -ftemplate-depth=100000000 -o main main.cpp` , мы получим новую ошибку:

```
g++-14: internal compiler error: Segmentation fault signal terminated program cc1plus
Please submit a full bug report, with preprocessed source (by using -freport-bug).
See <file:///usr/share/doc/gcc-14/README.Bugs> for instructions.
```

Компилятор упал с Segmentation fault из-за переполнения стека. Всё верно, мы программируем на языке, который позволяет уничтожить компилятор изнутри. Впрочем, мы сами приказали ему выпилиться, сняв разумное ограничение глубины рекурсии. Пациент следовал указаниям санитара, а уж то, что санитар оказался изощрённым садистом и маньяком — не вина пациента.

Давайте закончим издеваться над бедолагой и наконец пофиксим баг в нашем шаблоне Фибоначчи, из-за которого происходит вылет в отрицательные N. Сделаем мы это специализацией:

```
template<int N>
struct Fibonacci {
  static const int value = Fibonacci<N - 1>::value + Fibonacci<N - 2>::value;
};
template<>
struct Fibonacci<1> {
  static const int value = 1;
};
template<>
struct Fibonacci<0> {
  static const int value = 0;
};

std::cout << Fibonacci<20>::value;
```

И это уже будет работать без всяких флагов, пока какой-нибудь оболтус не захочет найти число Фибоначчи для отрицательного N. Специально для таких балбесов мы воспользуемся конструкцией из C++20 `requires`:

```
template<int N> requires (N >= 0)
struct Fibonacci {
  static const int value = Fibonacci<N - 1>::value + Fibonacci<N - 2>::value;
};
template<>
struct Fibonacci<1> {
  static const int value = 1;
};
template<>
struct Fibonacci<0> {
  static const int value = 0;
};
std::cout << Fibonacci<20>::value;
```

Теперь попытка компиляции какой-нибудь `Fibonacci<-15>` приведёт к более адекватной ошибке, а не к попыткам компилятора натянуть несчастного Фибоначчи на глобус:

```
In function ‘int main()’:
error: template constraint failure for ‘template  requires  N >= 0 struct Fibonacci’
```

```
    std::cout << Fibonacci<-20>::value;
                              ^

note: constraints not satisfied
In substitution of ‘template<int N>  requires  N >= 0 struct Fibonacci [with int N =
-15]’:
   required from here
   required by the constraints of ‘template  requires  N >= 0 struct Fibonacci’

note: the expression ‘N >= 0 [with N = -15]’ evaluated to ‘false’

    template<int N> requires (N >= 0)
                        ~~~^~~~
```

Давайте полюбуемся, во что превращает компилятор вывод `Fibonacci<5>`:

```
template<int N> requires (N >= 0)
struct Fibonacci {
  static const int value = Fibonacci<N -
1>::value + Fibonacci<N - 2>::value;
};
template<>
struct Fibonacci<1> {
  static const int value = 1;
};
template<>
struct Fibonacci<0> {
  static const int value = 0;
};
std::cout << Fibonacci<5>::value;
```

⇒

```
struct Fibonacci<0> {
  static const int value = 0;
};

struct Fibonacci<1> {
  static const int value = 1;
};
struct Fibonacci<2> {
  static const int value =
Fibonacci<1>::value +
Fibonacci<0>::value; // 1
};
struct Fibonacci<3> {
  static const int value =
Fibonacci<2>::value +
Fibonacci<1>::value; // 2
};
struct Fibonacci<4> {
  static const int value =
Fibonacci<3>::value +
Fibonacci<2>::value; // 3
};
struct Fibonacci<5> {
  static const int value =
Fibonacci<4>::value +
Fibonacci<3>::value; // 5
};
std::cout << Fibonacci<5>::value; // 5
```

Скорее всего, компилятор оптимизирует операции сложения и на выходе мы получим не операции сложения, а уже подставленные числа, но давайте сделаем это явно с помощью `constexpr` и добьём этим наш поток абсурда:

```
template<int N> requires (N >= 0)
struct Fibonacci {
  static constexpr const int value =
Fibonacci<N - 1>::value + Fibonacci<N -
2>::value;
};
template<>
struct Fibonacci<1> {
  static const int value = 1;
};
```

⇒

```
struct Fibonacci<0> {
  static const int value = 0;
};

struct Fibonacci<1> {
  static const int value = 1;
};
struct Fibonacci<2> {
  static constexpr const int value = 1;
};
```

```
template<>
struct Fibonacci<0> {
  static const int value = 0;
};
std::cout << Fibonacci<5>::value;
```

```
struct Fibonacci<3> {
  static constexpr const int value = 2;
};
struct Fibonacci<4> {
  static constexpr const int value = 3;
};
struct Fibonacci<5> {
  static constexpr const int value = 5;
};
std::cout << Fibonacci<5>::value; // 5
```

Метафункции

“Ну, рекурсивную функцию написать каждому дураку под силу”, — скажете вы. Если вы всё ещё не удивлены выполнением рекурсии, да ещё по сути с динамикой, во время компиляции, если вы всё ещё не верите, что шаблоны являются полноценным языком программирования, то давайте заставим пациента на этапе компиляции проверить условие и в зависимости от его истинности вернуть нужный нам тип — одним словом, напишем оператор `if` и назовём его `conditional`.

```
template<bool B, class T, class F>
struct conditional {
  using type = T;
};

template<class T, class F>
struct conditional<false, T, F> {
  using type = F;
};

// шаблонный using для удобства
template<bool B, class T, class F>
using conditional_t = typename conditional<B, T, F>::type;
```

Итак, мы всё ближе к тому, чтобы изобрести метаязык метапрограммирования на шаблонах. Наш тернарный оператор принимает на вход буль и на выход даёт тип T или F в зависимости от истинности буля.

“Но ведь мы не можем выводить какую-то сложную логику, поскольку работа ведётся с типами данных, а не с булевыми значениями”, — возразите вы. Ну так давайте изобретём метабулевы значения!

```
#include <iostream>
template<bool B>
struct bool_type {
  static constexpr bool value = B;
};
typedef bool_type<true> true_type;
typedef bool_type<false> false_type;

template<bool B, class T, class F>
struct conditional {
  using type = T;
};

template<class T, class F>
struct conditional<false, T, F> {
```

```
  using type = F;
};

template<bool B, class T, class F>
using conditional_t = typename conditional<B, T = true_type, F = false_type>::type;

// сделаем функцию сравнения чисел на этапе компиляции
template<int A, int B, typename Z = conditional_t<A <= B>>
void f() {
  // условная компиляция
  if constexpr (Z::value) {
    std::cout << "Меньше или равно";
  } else {
    // даже не попадёт в выходной бинарник
    std::cout << "Больше";
  }
}

int main() {
  f<2, 3>();
}
```

Мы ввели типы `true_false` и `false_type`, оба из которых содержат единственное статическое поле `value` с соответствующим булевым значением. В зависимости от ситуации наш тип будет иметь внутри себя `type` соответствующего типа. Таким образом, по `type::value` мы с лёгкостью определим, истинно наше выражение или же ложно.

Использовать `true_type` и `false_type` в , конечно, наглядно, но не слишком осмысленно. Давайте напишем метафнукцию сравнения типов `is_same` и посмотрим на пример её использования.

```
template<bool B>
struct bool_type {
  static constexpr bool value = B;
};
typedef bool_type<true> true_type;
typedef bool_type<false> false_type;

template<typename T, typename U>
struct is_same : false_type;

template<typename T>
struct is_same<T, T> : true_type;

// шаблонные переменные появились в C++14
template<typename T, typename U>
bool is_same_v = is_same<T, U>::value;

template<typename T, typename U>
bool compare_types(const T& l, const U& r) {
  if constexpr (is_same_v<T, U>) {
    std::cout << "Это один и тот же тип";
  } else {
    std::cout << "Это разные типы";
  }
}
```

Что по-вашему константа, как вы думаете? Как говорил кто-то из великих, в этом мире всё относительно. Вот и наш пациент утверждает, что константность — в целом тоже понятие весьма относительное: сегодня константа, а завтра переменная. Давайте накидаем метафункцию, которая делает константу неконстантой.

```
#include <iostream>
template<typename T>
struct remove_const {
  using type = T;
};
template<typename T>
struct remove_const<const T> {
  using type = T;
};
template<typename T>
using remove_const_t = remove_const<T>::type;

template<typename T>
void f(T& a) {
  const_cast<remove_const_t<T>&>(a) = 69;
}

int main() {
  int i = 42;
  const int& rci = i;
  f(rci);
  std::cout << i; // 42 или 69?
}
```

Наследование и шаблоны

Шаблонный using

Любопытно повторяющийся шаблонный шаблон (CRTP)

Шаблонные переменные

Зависимые имена

```
template<typename T>
struct S {
  using A = int;
};
template<>
struct S<double> {
  static const int A = 5;
};
int x = 0;
template<typename T>
void f(T x) {
  S<T>::A* x; // объявление или выражение?
}
int main() {
  f<int>();
}
```

Параметры типа Параметры, не относящиеся к типу Аргументы по умолчанию Специализации Шаблонные классы Шаблонный using ? Порядок использования шаблонов

Перегрузка шаблона, эвристика работы компилятора Шаблонный аргумент по умолчанию Полная и частичная специализация Порядок перегрузки Non-template parameters Числовые параметры у классов constexpr Шаблонные-шаблонные параметры Вычисления на этапе компиляции Шаблонные переменные static_assert

Интересные примеры кода:

```
template<typename T>
void f(T x) {
  std::cout << 1;
}

void f(int x) {
  std::cout << 2;
}
int main() {
  f<int>(0);
  f(0);
  f(0.5);
  return 0;
}
```