А вы знали об анонимных классах в D?

Адам Руппе*

2021-02-15

(**Блог Адама Руппе**,http://dpldocs.info/this-week-in-d/Blog.Posted_2021_02_15.html)

Я использую анонимные классы D, но даже я до этой недели не знал, как передавать аргументы в их конструкторы!

Совет недели

В D, также, как в Jav'e, есть анонимные классы. Чтобы создать такой экземпляр по месту, можно воспользоваться выражением **return new class {...}**:

```
class A{}

A foo() {
  return new class A {
    override string toString()
      { return "ANONYMOUS!"; }
    };
}
```

При этом даже не требуется указывать имя базового класса, поскольку это будет стандартный **Object**:

```
Object foo() {
  return new class A {
    // что-то полезное
    };
}
```

Можно даже реализовать и родителя, и какие-нибудь интерфейсы:

```
interface I { int getInt(); }

class Base { string getString() { return "string"; } }

I foo() {
  return new class Base, I {
    int getInt() { return 15; }
  };
}

void main() {
  import std.stdio;

```

*Перевод Глеб Куликов

```
    I i = foo();

    writeln(i.getInt());
    Base b = cast(Base) i;
    writeln(b.getString());
}
```

Миленько, иногда может сэкономить одну или пару строк отдельного определения класса. Ну ОК, разница не такая уж и большая, поскольку можно просто вложить класс в функцию и точно так -- же обращаться к локальным переменным, но всё равно может быть приятным.

На самом деле круто то, что вы *можете смешивать это с автоматическим возвратом* (**auto return**) *в D* и всё равно получать ссылку *на сам анонимный тип*, а не на его базового предка или интерфейс:

```
auto foo() {
  return new class {
    int i;
    // что-то полезное
    };
}
```

Почему это круто? Потому, что можно использовать на этом Дишные рефлексии!

```
auto foo() {
  return new class {
    int i;
  };
}

void main() {
  auto f = foo();

  //напечатаем "i" и автосгенерированный конструктор this:
  pragma(msg, __traits(derivedMembers, typeof(f)));
}
// компилятор напечатает "tuple("i", "this")"
```

Больше нигде D не разрешает объявлений в контексте выражения, поэтому анонимный класс может быть удобной группировкой для одноразовых штучек, передаваемых для рефлексии... но спецификация языка не допускает передачи классовых объектов в аргументы шаблона по значению. Поэтому вам придётся работать с ними строго, как с типами, или же исключительно на стороне CTFE[1].

```
template КрутаяШтучка(T) {
  // напечатаем, какие же члены типа автоматически выводятся
  pragma(msg, __traits(derivedMembers, T));
}

void main() {
  КрутаяШтучка!( typeof(new class {int y;}) );
}
//Компилятор напечатает tuple("y", "this")
```

[1] вычисление функций во время компиляции

Выражение **typeof (new class {})** здесь допустимо и позволяет вам сгруппировать некоторые объявления для использования в шаблоне и в то же время, не вводить отдельное имя... но действительно ли это полезно? Ну... возможно! Я собираюсь попридержать это дело про запас на тот случай, что в один прекрасный день, да пригодится. Но этот способ несколько многословен, так что не настолько убедителен, чтобы просто не написать объявление отдельно.

В любом случае, использование выражения в функции в качестве быстрой анонимной группировки не требует ничего особенного, и может быть забавной альтернативой кортежам. Но конечно, нужно понимать, что при этом происходит выделение памяти под класс. Чтобы этого избежать, можно объявить локальную переменную в границах[2]: **scope i = new class{}**;

```
void main() @nogc {
  scope a = new class {
    int a = 5;
  };

import core.stdc.stdio;
printf("%d\n", a.a);
}
// программа напечатает "5"
```

Но, конечно, такой трюк можно использовать только как локальную переменную, поскольку её нельзя вернуть (`return`) !

Забавно, что компилятор **dmd** с ключом «**-betterC**» принимает эту функцию, но линковка проваливается из-за отсутствия поддержки в рантайме. Возможно, это ошибка обнаружения ограничений **betterC**, но, поскольку я ненавижу эти ограничения, регистрировать её я не буду.

Однако, если уж речь зашла о **betterC**, эта маленькая программа работает и там!

```
extern(C++) class O {
  extern(D)
  abstract int omg();
}

extern(C)
int main() @nogc {
  scope a = new class O {
    int a = 5;
    override int omg() { return a; }
  };

  import core.stdc.stdio;
  printf("%d\n", a.a);
  return 0;
} // программа напечатает 5
```

Ура, **dmd -betterC** работает. Недавно я использовал анонимные классы для реализации **COM**, и это было неплохо, поэтому тот факт, что вы можете работать с **betterC**, действительно крут и, вероятно, полезен.

Заметьте, границы класса переопределяют локальную область видимости! С точки зрения правил языка смысл в этом есть: переменная – член родительского класса

[2] т.е., в заданной области видимости

«ближе» к этому конструктору. Но об этом легко забыть, а компилятор не предупредит вас о бесполезном самоприсваивании.

```
class A {
  int a;
}

A foo() {
  int a;

  return new class A {
    this() {
      /* Здесь 'a' ссылается на 'A.a', а НЕ на локальную
         переменную 'a' выше по тексту!
      */

    // Здесь this самоприсваемое
  this.a = a; // ничего не делает
    }
  };
}
```

Так что это всё забавно, но опыт подсказывает, что предупреждение необходимо. Как можно передать внутрь эту локальную переменную? В норме вы могли быть передать её конструктору, но могут ли конструкторы анонимных классов иметь аргументы?

Я думал, что нет. Но, просмотрев спецификацию Java и обнаружив их там, я еще раз взглянул на спецификацию D... и нашел их и здесь. Вот о чём я не знал до этой недели! Однако, полагаю, что синтаксис дикий. Оцените велеречивость:

```
void main() {
  int a = 83;
  auto obj = new class (a)
  {// вот это (a) и есть список аргументов конструктора!
    this(int a){
      import std.stdio;
      writeln("Эта _a_ была передана во внутрь: ", a);
    }
  };
} // напечатает: Эта _a_ была передана во внутрь: 83
```

Да–да, список аргументов для конструктора следует **после** ключевого слова **class**, *перед* базовым классом и интерфейсами.

Технически, спецификация допускает размещение списка аргументов аллокатора и после ключевого слова **new**, но похоже, что это нужно как раз для размещения **new** и к делу не относится.

Как бы то ни было, место после ключевого слова **class**, это последнее, что можно было бы предположить. Обычно вы пишете «**new Object (аргументы)**», поэтому я попробовал запись «**new class Base (аргументы)**». В Jav'e это работает, но в D не прокатило... *(**new class ... } (аргументы)**? Не-а.*

Только после просмотра грамматики в спецификации, я дотумкал до описанного варианта. Я думаю, смысл в этом, вроде как, есть. Аргументы идут после имени класса, имя класса следует за ключевым словом **class** и, поскольку у этого класса нет имени, кажется разумным, что аргументы находятся там, где они есть. А поскольку у D есть шаблоны, раз-

мещение аргументов в том же месте, что и в Jav'e, может показаться непонятным. (Хотя... опять же, для создания экземпляра шаблона требуется оператор **!**...).

Как бы там ни было, факт в том, что вы можете это сделать! И это тот способ, которым вы можете передавать локальные переменные, не беспокоясь о конфликтах имён между членами базового класса и этими локальными переменными.

Бесплатный совет!

Знаете ли вы, что можно указывать пользовательские атрибуты (**UDA**) в объявлении модуля?

```
1 @foo module чойто;
2
3 import модуль_с_foo;
```

Прикол в том, что хотя объявления импорта не допустимы до объявления модуля, тем не менее, символы из более поздних импортов, оказываются доступны.
Сейчас в объявлениях модулей допустимы только **UDA** и ключевое слово **deprecated**.

Я этого не знал, пока кто-то в чате не указал мне на это место в спецификации.