УДК 004.435
DOI: 10.15827/0236-235X.126.190-196
Дата подачи статьи: 29.05.18
2019. Т. 32. № 2. С. 190–196

Реализация метаязыковой абстракции для поддержки ООП средствами языка Си

А.М. Дергачев [1], *к.т.н., доцент, dam600@gmail.com*
И.О. Жирков [1], *тьютор, igorjirkov@gmail.com*
И.П. Логинов [1], *аспирант, ivan.p.loginov@gmail.com*
Ю.Д. Кореньков [1], *аспирант, ged.yuko@gmail.com*

[1] *Санкт-Петербургский национальный исследовательский университет информационных технологий, механики и оптики (Университет ИТМО), г. Санкт-Петербург, 197101, Россия*

В данной работе демонстрируется использование макроопределений высшего порядка для реализации поддержки объектно-ориентированной парадигмы программирования в языке C89 без расширений. Выбор парадигмы программирования является важной задачей, предшествующей реализации программы. В рамках объектно-ориентированной парадигмы программирования описывается широкий класс задач. Многие распространенные высокоуровневые языки общего назначения, такие как C++, C#, Java, предоставляют поддержку этого программирования. Однако применение языков, поддерживающих объектно-ориентированное программирование, не всегда технически возможно из-за отсутствия средств разработки под целевую платформу, в частности, компилятора. Так, например, для предметно-ориентированных процессоров (ASIP) зачастую предоставляется только компилятор языка Си как наиболее распространенного низкоуровневого языка программирования. Кроме того, относительно небольшой размер языка, а также его близость к языку ассемблера позволяют быстро реализовать компилятор для новой архитектуры. Вместе с тем препроцессор языка Си позволяет за счет создания системы макроопределений высшего порядка реализовать сложную логику генерации кода, выходящую за рамки тривиального заполнения шаблона значениями параметров макроопределения.

В статье с помощью примеров исходного кода показана реализация инкапсуляции, наследования и полиморфизма. Инкапсуляция делает невозможным обращение к непубличным методам и полям класса извне уже в момент компиляции. Особое внимание авторы уделяют типобезопасности генерируемого кода: введение наследования не означает еще большее ослабление правил типизации языка Си.

Результаты исследования предполагают применимость такого подхода для реализации программ, эффективно использующих объектно-ориентированное программирование, при разработке на языке Си в случае невозможности использования современных объектно-ориентированных языков.

Ключевые слова: *язык Си, препроцессор, объектно-ориентированное программирование, метапрограммирование, макрос.*

При написании программы важно определить адекватную задаче парадигму программирования. Это может быть программирование от состояний, объектно-ориентированное, декларативное и т.д. Выбор парадигмы, с помощью которой решение задачи будет описано максимально естественно и лаконично, позволяет быстрее создавать более надежные программы за счет уменьшения сложности процесса написания кода и времени на его отладку. В мире широко распространены языки, как предполагающие следование определенным парадигмам (Smalltalk, ML), так и поддерживающие множество парадигм (C#, Java, OCaml и др.). В то же время лишь для небольшого количества языков существуют компиляторы для большинства различных аппаратных и программных целевых платформ, так как разработка эффективных кросс-компиляторов, особенно языков высокого уровня, является достаточно трудоемкой задачей.

Весьма интересным кажется использование существующих компиляторов языка Си, реализованных практически под все существующие платформы. При этом Си не обладает богатыми выразительными возможностями, предоставляя лишь самые простые абстракции для работы с кодом и данными. Из всего многообразия парадигм программирования Си предоставляет поддержку лишь процедурного языка. Язык C++, созданный на основе Си, включает в себя поддержку других парадигм, в том числе объектно-ориентированной. К сожалению, до настоящего времени существуют платформы,

для которых компиляторы C++ отсутствуют или обладают большим количеством ошибок, что неудивительно, так как C++ гораздо сложнее для трансляции, чем Си. По этим причинам многие программисты реализовывали поддержку объектно-ориентированной парадигмы программирования в Си с целью упростить решение определенного класса задач, в котором этот подход оправдан. Возможно, самым известным является вариант, описанный в [1].

Добавление поддержки нового стиля в язык – это описание нового предметно-ориентированного языка (DSL) [2] средствами исходного языка. В идеале использование DSL не должно приводить к накладным расходам во время выполнения программы: к выделению памяти под структуры, без которых можно было бы обойтись, дополнительным вычислениям и т.д. Плюсом является и реализация DSL на самом языке без использования внешнего препроцессора, лишь благодаря собственным возможностям метапрограммирования, предоставляемым языком [3]. Существующие решения не удовлетворяют этим критериям: или используется специально написанный внешний препроцессор [4], или имеющий служебное назначение код прописывается вручную, или используются излишние абстракции времени выполнения (такие, как указатели на все методы, хранящиеся в экземплярах объектов).

Авторы данной статьи предлагают подход, основанный только на использовании макросов языка Си и не использующий такие расширения GNU C, как макросы с переменным количеством аргументов, что обеспечивает максимальную переносимость. Кроме того, предлагаемое решение направлено на использование системы типов для повышения надежности программы, а потому обходится без использования указателей неопределенного типа (void*).

Постановка задачи

Основополагающими концепциями *объектно-ориентированного программирования* (ООП) являются инкапсуляция, наследование и полиморфизм [5, 6].

Инкапсуляция предполагает скрытие деталей реализации объектов. Она выражается в наличии у классов приватных и защищенных полей и методов.

Наследование предполагает возможность определять классы на основе существующих и использовать их взаимозаменяемым образом.

Полиморфизм в контексте ООП предполагает возможность перегрузки виртуальных функций, при которой реализация метода в классе-наследнике отличается от реализации в классе-родителе.

В данной работе используются возможности макропроцессора Си для генерации кода, обеспечивающего построение иерархии классов и сопутствующих служебных конструкций, таких, как конструкторы.

К сожалению, язык макроподстановок крайне беден (не реализует даже весь класс примитивно-рекурсивных функций). Авторы предлагают пути обхода важных проблем, которые ограничивают выразительную силу макросов:

– макросы принимают только ограниченное количество параметров и не могут быть перегружены;

– макросы не могут вызываться рекурсивно;

– макросы не могут изменять глобальное состояние препроцессора, то есть определять или переопределять другие макросы.

Из соображений читаемости продемонстрируем не конфликтующие друг с другом способы реализации трех основополагающих концепций ООП, которые легко можно скомбинировать друг с другом.

Общие принципы реализации

Основной целью создаваемого DSL является предоставление программисту возможности описания каждого класса в компактном виде. При этом на основании этого описания необходимо сгенерировать большое количество служебных конструкций.

Поскольку объекты являются составными типами данных, моделируем их с помощью структур. Методы класса будут соответствовать функциям, неявно принимающим в качестве первого аргумента указатель на экземпляр класса, называемый this. Чтобы обезопаситься от коллизий имен, все названия методов класса предварим префиксом: именем класса, за которым следует символ подчеркивания. Например, для класса object метод char* tostring() будет реализован на основе функции object_tostring(object* this).

В Си макросы могут принимать лишь ограниченное количество аргументов. Кроме того, их нельзя перегружать [7]. Классы, однако, могут иметь сколь угодно большое описание, поэтому использовать для них макросы, параметризованные данными о полях и методах, нельзя.

Для такого описания предлагаем использовать макросы второго порядка, которые в качестве аргументов будут принимать имена других макросов и запускать их на фрагментах описания класса, таких, как имя родителя, имена полей и т.д. Таким образом, с точки зрения вычислителя препроцессора [8] описание класса – обобщенный алгоритм генерации кода, который может специализироваться под те или иные нужды:

```
#define class_mc( name, public_field, public_method) \
   name( MC )\
   public_field( int, x, 0, "%d" )\
   public_field( float, y, 0, "%f" )\
   public_method( somemethod )
```

Аналогичным образом будет строиться описание методов:

```
#define somemethod(name, of, arg, returns, body )\
   name( somemethod ) of( mc ) returns( int )\
   arg( int, a )\
   arg( int, b )\
   body( { return this->x + this->y + a + b; } )
```

Для облегчения дальнейшей работы опишем несколько макросов.

- Следующие макросы игнорируют свои параметры (передавая их в описание класса, пропустим ненужную в данном контексте информацию о нем):

```
#define _1(q)
#define _2(q,w)
#define _3(q,w,e)
#define _4(q,w,e,r)
```

- Макрос id повторяет свой аргумент:

```
#define id(x) x
```

- Макрос classname раскроется в имя класса, чье описание он принимает в качестве параметра:

```
#define classname(descr) descr(id, _4, _4)
classname(mc) // раскроется в “mc”
```

Специализируя макрос, описывающий класс, можно достаточно легко получить практически любой код, тем или иным образом использующий информацию о классе.

- Описание класса как структуры:

```
#define _field(t, n, dv, fs) t n;
#define class(descr) \
   typedef struct classname(descr) {\
      descr( _1, _field, _1)\
}  classname(descr);
```

Для класса mc результатом подстановки class(class_mc) будет:

```
typedef struct mc { int x; float y; } mc ;
```

- Конструктор по умолчанию. Эта функция возвращает инициализированный экземпляр класса, в котором всем полям присвоены значения по умолчанию:

```
#define pair(x,y) pair_(x,y)
#define pair_(x,y) x##_##y
#define _field_init( t, n, v, _ ) this.n = v;
#define method_name( descr )
#define default_ctor_proto(descr) \
   classname(descr) pair(classname(descr),
init)(void)
#define default_ctor(descr)  default_ctor_proto(descr) {\
         classname(descr) this; \
         descr(_1, _field_init, _1)\
      return this; \
   }
```

Здесь также приведены несколько служебных макросов. Стоит отметить, что макросы pair и pair_ необходимы, так как части токена, составленного с помощью оператора ##, в таком случае смогут быть посчитаны другими макросами.

Реализация инкапсуляции

Язык Си предоставляет механизм инкапсуляции с помощью модулей. Другие механизмы сокрытия сущностей языка не предусмотрены. Нельзя скрыть часть структуры так, чтобы она была не видима извне модуля, лишь всю реализацию целиком, используя неполные типы и запретив таким образом инстанцирование. Однако можно использовать более гибкий подход, основанный на отношении конвертируемости типов друг в друга. Так как модули транслируются раздельно, типы данных, определенные в разных модулях с помощью одного и того же описания, являются разными типами. Типы одинакового размера считаются конвертируемыми друг в друга, а если у них одинаковое описание, то семантика доступа к их полям определена.

Для скрытия приватных членов, однако, необходимы различные описания: как структура может быть видна внешнему миру и как ее воспринимают функции внутри модуля. Удовлетворим требование конвертируемости так, как показано в таблице.

Определение структур

Structure definition

Заголовочный файл (описание, доступное другим модулям)	**Описание внутри модуля**

typedef struct mc {
int x; int y;
union {
char _stub[sizeof(struct {int _z;})];
};
} mc;
	typedef struct mc {
	int x; int y;
	union {
	struct { int _z; };
	char _stub[sizeof(struct {int _z;})];
	};
	} mc;

Поскольку оператор sizeof(struct {int _z;}) вернет количество байтов, равное размеру структуры struct{int _z;}, размеры объединений в описаниях слева и справа совпадают. Упаковка в union гарантирует, что политика выравнивания, выбранная компилятором, будет одинакова для объединений во всех описаниях, а значит, sizeof(struct mc) будет возвращать для них одинаковые значения.

Приватные методы скрыть легче: достаточно поместить их внутрь модуля и пометить ключевым словом static.

Чтобы описания класса и его методов различались внутри модуля и заголовочного файла, используем технику X Macro, описанную в [9]. Все описание класса будет содержаться в заголовочном файле (в примере – myclass.h). Файл модуля будет выглядеть следующим образом:

```
#define IMPL
#include “myclass.h”
```

Внутри файла myclass.h потребуется определить две разные версии макроса class с помощью конструкции #ifdef: если IMPL определен, то эта версия включит в себя реализации функций, а не только их прототипы, а также описание структуры, включающее приватные поля.

Реализация наследования

Наследование позволяет доопределять классы. При этом методы, работающие на экземплярах класса-родителя, могут быть запущены на классах-потомках. Достичь этой цели просто: достаточно добавить первым полем структуры экземпляр структуры-родителя. Помимо этого, конструктор должен начинать свое выполнение с инициализации родителя:

```
#define declare_base( descr ) classname(descr)
pair(as, classname(descr));

#define base_ctor_call(bdescr) this.pair(as,class-
name(bdescr)) = \
pair(classname(bdescr), init)();
#define default_ctor(descr) default_ctor_proto(descr) {\
        classname(descr) this; \
        descr(_1, base_ctor_call, _4)\
        descr(_1, _1,_field_init)\
        return this; \
  }
```

Реализация полиморфизма

Для реализации виртуальных функций потребуется реализовать таблицы виртуальных функций. Каждая такая таблица будет храниться в глобальной переменной, а конструктор объекта класса должен будет заполнить ссылку на нее в специальном поле класса. Конструирование такой таблицы представляет определенные трудности.

- Ссылки на эти таблицы должны быть во всех классах с одинаковым смещением относительно их начала (например, их первыми элементами), но их типы должны различаться.

Для решения этой проблемы предлагается использовать объединения, которые позволяют наложить две разные карты памяти на один и тот же фрагмент, а именно – на первое поле структуры, хранящее адрес таблицы. Сгенерированный код будет выглядеть так:

```
union a { a_vtable* vtable;
struct {object as_object; int y;  };};
```

- Таблицы содержат функции, в сигнатурах которых упоминается имя класса (для параметра this), а в классе упоминается тип соответствующей таблицы, то есть объявления класса и таблицы взаимозависимы.

Для решения этой проблемы необходимо отделить неполные описания типов таблицы и класса от их определений:

```
#define declarations(descr) \
   struct vtable_name(descr);\
   union classname(descr);\
   typedef union classname(descr) classname(descr);\
   typedef struct vtable_name(descr)
vtable_name(descr);
```

- Имя экземпляра виртуальной таблицы зависит от макросов, описывающих и класс, и метод. Это означает, что возникает цикл: по описанию класса генерируется таблица методов, для каждого макроса-описания метода она запускает макрос-описание класса, информацию о котором метод хранит с меткой of. Препроцессор, однако, не поддерживает рекурсию в явном виде и откажется раскрывать макрос-описание класса внутри него же.

Решение этой проблемы возможно с помощью или отложенного выполнения макросов, или определенных соглашений, устанавливающих в рамках предлагаемого решения правила именования классов и макроопределений. Авторы придерживаются второго варианта, который, хотя и менее автоматизирован, более переносим из-за того, что первое решение сильно зависит от реализации препроцессора (от порядка раскрытия макросов).

Прокомментируем демонстрационный код, показывающий, как осуществить автоматическое построение таблиц виртуальных функций. Основные особенности этого решения:

– виртуальные методы или определяются для класса, или переопределяются с указанием

того, в каком классе они были объявлены изначально; если перегрузок было несколько, все они должны быть указаны;

– таблицы виртуальных методов должны быть инициализированы вызовом специальных функций в начале работы программы – по одной функции для каждого класса;

– во всех классах присутствует неявное поле, хранящее адрес таблицы виртуальных функций и инициализируемое конструктором по умолчанию; в дальнейшем это можно легко использовать для реализации автоматизированной рефлексии времени выполнения (RTTI) [10], добавив в таблицу метаданные класса.

Приведем пример кода, демонстрирующий перегрузку функций. В нем создана иерархия из классов Object, A и B. Object определяет метод tostring, который перегружен в B и наследуется A:

```
#define object_tostring(name, of, arg, returns, body) \
   name( tostring ) of( object ) returns( const char*
)\
   body( { return "Object or A"; } )

#define b_tostring(name, of, arg, returns, body) \
   name( tostring ) of( b ) returns( const char* )\
   body( { return "B"; } )

#define class_object(name, extends, public_field,
vmethod, override)\
   name( object )\
   public_field( int, x, 0, "%d" )\
   vmethod( object_tostring )

#define class_a(name, extends, public_field,
vmethod, override)\
   name( a )\
   extends( class_object )\
   public_field( int, y, 0, "%d" )\

#define class_b(name, extends, public_field,
vmethod, override)\
   name( b )\
   extends( class_a )\
   public_field( int, z, 0, "%d" )\
   override( object, tostring, b_tostring )

class( class_object )
class( class_a )
class( class_b )

void puts(const char*);
int main() {
   object_vtable_init();
   a_vtable_init();
   b_vtable_init();
   b myb = b_init();
   a mya = a_init();
/* Выведет B */
   puts(vt_object_tostring( (object*)&myb)((ob-
ject*)&myb));
/* Выведет Object or A */
   puts(vt_object_tostring( (object*)&mya)((ob-
ject*)&mya));
   return 0;
}
```

Приведем некоторые служебные макросы, аналогичные методам чтения (геттерам):

```
#define classname(descr) descr(id, _1, _4, _1, _3)
#define classbase(descr) descr(_1, id, _4, _1, _3)
#define for_base(descr, f) descr(_1, f, _4, _1, _3)
#define for_fields(descr, f) descr(_1, _1, f, _1, _3)
#define for_vmethods(descr, f) descr(_1, _1, _4, f, _3)
#define for_override(descr, f) descr(_1, _1, _4, _1, f)
```

Конструктор, помимо инициализации полей, должен установить ссылку на правильный экземпляр таблицы виртуальных функций:

```
#define _field_init( t, n, v, _ ) this.n = v;
#define ctor_name(descr) pair(classname(descr), init)
#define default_ctor_proto(descr) classname(descr)
ctor_name(descr) (void)
#define base_ctor_call(bdescr) this.pair(as,class-
name(bdescr)) =\
 pair(classname(bdescr), init)();

#define default_ctor(descr) default_ctor_proto(de-
scr) {\
        classname(descr) this; \
        descr(_1, base_ctor_call, _4,_1, _3)\
        descr(_1, _1,_field_init,_1, _3)\
        this.vtable = & vtable_instance(descr);\
        return this; \
   }
```

Геттеры для информации о методах:

```
#define m_name( m ) m(id, _1, _2, _1, _1 )
#define m_of( m ) m(_1, id, _2, _1, _1 )
#define m_for_args( m, f ) m(_1, _1, f, _1, _1 )
#define m_returns( m ) m(_1, _1, _2, id, _1 )
#define m_body( m ) m(_1, _1, _2, _1, id )
#define m_arg_def( t, n ) , t n
```

Макрос, строящий прототип метода (сигнатуру функции) по его описанию:

```
#define m_proto( m ) m_returns(m) pair(_vt,
pair(m_of(m), m_name(m)))\
   ( m_of(m)* this \
      m_for_args(m, m_arg_def) )

#define method_def( m ) m_proto(m) { m_body(m) }
```

Макросы, описывающие структуру таблицы виртуальных функций:

```
#define vmethod_entry( m ) m_returns(m)
(*m_name(m))( m_of(m) * \
      m_for_args(m, m_arg_def) );

#define vtable_name(descr) pair(classname(descr), vtable)
#define vtable_base_def(descr) struct
vtable_name(descr) vtable_base;
```

```
#define vtable_instance(descr) pair(vtable_name(descr),inst)

#define vtable(descr) \
   struct vtable_name(descr) {\
   descr(_1, vtable_base_def, _4, _1, _3)\
descr(_1, _1, _4, vmethod_entry , _3)\
} vtable_instance(descr);

#define call_vtable_base(descr)
pair(vtable_name(descr),init)();
#define init_base_vtable_part(descr)
vt->vtable_base = vtable_instance(descr);

#define vtable_init_member(m) pair(m_of(m) ,
vtable_inst).m_name(m ) = pair(_vt,pair( m_of(m),
m_name(m)));
```

Для описания кода инициализации таблиц, перезаписывающего их части в соответствии с перегрузками, используются следующие макросы:

```
#define vtable_override( baseentry, name, overridden) ((baseentry ## _vtable*)vt)->name = \
   pair(_vt, overridden);

#define vtable_init(descr) void
pair(vtable_name(descr),init)(void) {\
   descr(_1, call_vtable_base, _4, _1, _3)\
   vtable_name(descr) * const vt = &vtable_instance(descr);\
   for_base(descr, init_base_vtable_part)\
   for_vmethods( descr, vtable_init_member)\
   for_override( descr, vtable_override )\
}

#define override_method_def(baseentry, name,
overridden) method_def(overridden)

#define m_caller(m) m_returns(m) \
   (*pair(vt, pair( m_of(m),
m_name(m)))(m_of(m)* this m_for_args(m,
m_arg_def)))(m_of(m)*) { return this->vtable->m_name(m); }

#define declare_base classname(descr) pair(as,
classname(descr));
```

Само описание класса состоит из описаний неполных типов, таблицы виртуальных функций, структуры его полей, методов, конструктора и инициализатора таблицы виртуальных функций:

```
#define class(descr) \
   declarations(descr)\
   vtable(descr)\
      union classname(descr) {\
      vtable_name(descr) * vtable;\
      struct {\
      descr(_1, declare_base, _4, _1, _3)\
        descr( _1, _1,  _field, _1, _3)\
      };\
};\
default_ctor( descr )\
for_vmethods(descr, method_def)\
for_override(descr, override_method_def)\
for_vmethods(descr, m_caller)\
vtable_init(descr)
```

Заключение

Предлагаемая концепция использования макросов высшего порядка показала себя мощным инструментом, достаточным для реализации очень многих абстракций исключительно за счет дополнительных вычислений времени компиляции. Представляется возможной реализация и других инструментов, доступных программистам на языках высокого уровня, таких, как конструкторы не по умолчанию, деструкторы, автоматическое создание и удаление объектов, множественное наследование. Интересна также возможность применения расширения GNU C, позволяющего использовать макросы с произвольным количеством аргументов и расширяющего выразительные возможности макросов, по крайней мере, до примитивно-рекурсивных функций.

Литература

1. Deniau L. The C object system: using C as a high-level object-oriented language. arXiv.org. 2010. URL: https://arxiv.org/abs/1003.2547 (дата обращения: 20.03.2018).
2. Mernik M., Heering J., Sloane A.M. When and how to develop domain-specific languages. ACM Computing Surveys (CSUR), 2005, vol. 37, no. 4, pp. 316–344. DOI: http://doi.acm.org/10.1145/1118890.1118892.
3. Фаулер М., Парсонс Р. Предметно-ориентированные языки программирования. М.: Вильямс, 2011. 576 с.
4. Jacobs M., Lewis E.C. SMART C: a semantic macro replacement translator for C. Proc. 6th IEEE Intern. Workshop Source Code Analysis and Manipulation (SCAM'06), 2006, pp. 95–106.
5. Smith B. Object-oriented programming: In Advanced ActionScript 3.0: Design Patterns, USA, NY, Apress, 2011, pp. 1–25.
6. Pecinovský R., Pavlíčková J., Pavliček L. Let's modify the objects-first approach into design-patterns-first. ACM SIGCHI Bul., 2006, vol. 38, no. 3, pp. 188–192.
7. Ernst M.D., Badros G.J., Notkin D. An empirical analysis of C preprocessor use. Software Engineering, IEEE Transactions, 2002, vol. 28, no. 12, pp. 1146–1170. DOI: 10.1109/TSE.2002.1158288.
8. Meyers R. The new C: X macros. 2001. URL:

http://www.drdobbs.com/the-new-c-x-macros/184401387 (дата обращения: 20.03.2018).

9. Подбельский В.В. Динамическая идентификация типов (RTTI): В кн.: Язык Си++. М.: Финансы и статистика, 2003. 560 с.

10. Fokin A., Troshina K., Chernov A. Reconstruction of class hierarchies for decompilation of C++ programs. Proc. 14th Europ. Conf. Soft. Maintenance and Reengineering. IEEE, 2010, pp. 240–243. DOI: 10.1109/csmr.2010.43.

Software & Systems
DOI: 10.15827/0236-235X.126.190-196

Received 29.05.18
2019, vol. 32, no. 2, pp. 190–196

Implementing metalinguistic abstraction to support OOP using C

A.M. Dergachev [1], *Ph.D. (Engineering), Associate Professor, dam600@gmail.com*
I.O. Zhirkov [1], *tutor, igorjirkov@gmail.com*
I.P. Loginov [1], *Postgraduate Student, ivan.p.loginov@gmail.com*
Yu.D. Korenkov [1], *Postgraduate Student, ged.yuko@gmail.com*

[1] *The National Research University of Information Technologies, Mechanics and Optics, St. Petersburg, 197101, Russian Federation*

Abstract. The paper shows the use of higher order macro definitions to support the object-oriented programming paradigm in C89 without extensions. Choosing the right programming style is important prior to writing a code. A large class of problems is described using object-oriented programming style. Many mainstream programming languages such as C++, C# or Java provide support for this programming style. However, it is not always possible to utilize these languages as the required development software such as compilers for some platforms might not be available. A typical example of this situation is Application-Specific Instruction-set Processor (ASIP), which is provided with a C compiler. The smaller set of C language features and its low-level nature allow quick and cheap compiler implementation. At the same time, the C preprocessor can be used for a sophisticated logic generation that goes far beyond simple parameterized substitutions.

This paper presents an internal support of the object-oriented programming style implemented in C89 without language extensions via an extensive usage of higher-order macro definitions. The example code shows the implementation of encapsulation, inheritance and polymorphism principles. Encapsulation syntactically prohibits accessing private fields and methods in compile time. We pay special attention to type-safety of generated code: the inheritance implementation does not weaken the already weak static typing used in C.

The results of this work can be used to construct object-oriented programs using only C89 compiler in case the usage of object-oriented languages is impossible.

Keywords: C, preprocessor, object-oriented programming, metaprogramming, macro definition.

References

1. Deniau L. *The C Object System: Using C as a High-Level Object-Oriented Language*. 2010. Available at: https://arxiv.org/abs/1003.2547 (accessed March 20, 2018).

2. Mernik M., Heering J., Sloane A.M. When and how to develop domain-specific languages. *ACM Computing Surveys (CSUR)*. 2005, vol. 37, no. 4, pp. 316–344. DOI: http://doi.acm.org/10.1145/1118890.1118892.

3. Fowler M., Parsons R. *Domain-Specific Languages.* Addison Wesley Publ., 2010, 640 p. (Russ. ed.: Moscow, Volyams Publ., 2011, 576 p.).

4. Jacobs M., Lewis E.C. SMART C: A semantic macro replacement translator for C. *SCAM'06. 6th IEEE Intern. Workshop on Source Code Analysis and Manipulation*. 2006, pp. 95–106.

5. Smith B. Object-oriented programming. *AdvancED ActionScript 3.0: Design Patterns.* Apress, 2011, pp. 1–25.

6. Pecinovský R., Pavlíčková J., Pavlíček L. Let's modify the objects-first approach into design-patterns-first. *ACM Sigcse Bul*. 2006, vol. 38, no. 3, pp. 188–192.

7. Ernst M.D., Badros G.J., Notkin D. An empirical analysis of C preprocessor use. *IEEE Trans. on Software Engineering*. 2002, vol. 28, no. 12, pp. 1146–1170. DOI: 10.1109/TSE.2002.1158288.

8. Meyers R. *The New C: X Macros. Dr.Dobb's 2001*. Available at: http://www.drdobbs.com/the-new-c-x-macros/184401387 (accessed March 20, 2018).

9. Podbelsky V.V. 12.6 Dynamic Type Identification (RTTI). *C++*. 4th ed. Moscow, Finansy i Statistika Publ., 2003, 560 p.

10. Fokin A., Troshina K., Chernov A. Reconstruction of class hierarchies for decompilation of C++ programs. *IEEE 14th European Conf. on Software Maintenance and Reengineering*. 2010, pp. 240–243. DOI: 10.1109/csmr.2010.43.