MODELING AND ANALYSIS OF INFORMATION SYSTEMS, VOL. 30, NO. 1, 2023
journal homepage: www.mais-journal.ru

THEORY OF COMPUTING

C Language Extension to Support Procedural-Parametric Polymorphism

A. I. Legalov[1], P. V. Kosov[1]

DOI: 10.18255/1818-1015-2023-1-40-62

[1]Higher school of Economics, National research University, 20, Myasnitskaya str., Moscow 101000, Russia.

MSC2020: 68N15, 68Q55
Research article
Full text in Russian

Received November 10, 2022
After revision February 3, 2023
Accepted February 8, 2023

Software development is often about expanding functionality. To improve reliability in this case, it is necessary to minimize the change in previously written code. For instrumental support of the evolutionary development of programs, a procedural-parametric programming paradigm was proposed, which made it possible to increase the capabilities of the procedural approach. This allows to extend both data and functions painlessly. The paper considers the inclusion of procedural-parametric programming in the C language. Additional syntactic constructions are proposed to support the proposed approach. These constructions include: parametric generalizations, specializations of generalizations, generalizing functions, specialization handlers. Their semantics, possibilities and features of technical implementation are described. To check the possibilities of using this approach, models of procedural-parametric constructions in the C programming language were built. The example in the article demonstrates the flexible extension of the program and support of multiple polymorphism.

Keywords: programming languages; compilation; procedural-parametric programming; polymorphism; multiple polymorphism; evolutionary software development

INFORMATION ABOUT THE AUTHORS

Alexander I. Legalov correspondence author	orcid.org/0000-0002-5487-0699. E-mail: alegalov@hse.ru Doctor of Technical Sciences, Professor.
Pavel V. Kosov	orcid.org/0000-0002-9035-312X. E-mail: pvkosov@hse.ru Graduate Student.

For citation: A. I. Legalov and P. V. Kosov, “C Language Extension to Support Procedural-Parametric Polymorphism”, *Modeling and analysis of information systems*, vol. 30, no. 1, pp. 40-62, 2023.

МОДЕЛИРОВАНИЕ И АНАЛИЗ ИНФОРМАЦИОННЫХ СИСТЕМ, ТОМ 30, № 1, 2023
сайт журнала: www.mais-journal.ru

THEORY OF COMPUTING

Расширение языка C для поддержки процедурно-параметрического полиморфизма

А. И. Легалов[1], П. В. Косов[1]

DOI: 10.18255/1818-1015-2023-1-40-62

[1]Национальный исследовательский университет «Высшая школа экономики», ул. Мясницкая, д. 20, г. Москва, 101000 Россия.

УДК 004.4'42, 004.43
Научная статья
Полный текст на русском языке

Получена 10 ноября 2022 г.
После доработки 3 февраля 2023 г.
Принята к публикации 8 февраля 2023 г.

Разработка программного обеспечения зачастую связана с расширением функциональности. Для повышения надежности в этом случае необходимо минимизировать изменение ранее написанного кода. Для инструментальной поддержки эволюционной разработки программ была предложена процедурно-параметрическая парадигма программирования, что позволило повысить возможности процедурного подхода. Это обеспечивает безболезненное расширение как данных, так функций, используя при этом статическую типизацию. В работе рассматривается включение процедурно-параметрического программирования в язык С. Предлагаются дополнительные синтаксические конструкции, ориентированные на поддержку предлагаемого подхода. К ним относятся: параметрические обобщения, специализации обобщений, обобщающие функции, обработчики специализаций. Описываются их семантика, возможности и особенности технической реализации. Для проверки возможностей использования данного подхода построены модели процедурно-параметрических конструкций на языке программирования С. Приведенный пример демонстрирует гибкое расширение программы и поддержку множественного полиморфизма.

Ключевые слова: языки программирования; компиляция; процедурно-параметрическое программирование; полиморфизм; множественный полиморфизм; эволюционная разработка программного обеспечения

ИНФОРМАЦИЯ ОБ АВТОРАХ

Александр Иванович Легалов автор для корреспонденции	orcid.org/0000-0002-5487-0699. E-mail: alegalov@hse.ru доктор технических наук, профессор.
Павел Владимирович Косов	orcid.org/0000-0002-9035-312X. E-mail: pvkosov@hse.ru аспирант.

Для цитирования: A. I. Legalov and P. V. Kosov, "C Language Extension to Support Procedural-Parametric Polymorphism", *Modeling and analysis of information systems*, vol. 30, no. 1, pp. 40-62, 2023.

Введение

При разработке программных систем необходимо учитывать различные дополнительные факторы, многие из которых определяются как критерии качества программного обеспечения (ПО). Расширение программы без изменения ранее написанного кода является одним из них. Оно обуславливается как неполным знанием окончательной функциональности программ во время ее создания и начальной эксплуатации, так и необходимостью ускорить выход продукта на рынок за счет реализации только базового набора функций с последующим его наращиванием. В подобных ситуациях добавление новых программных конструкций в код, сопровождаемое его изменением, зачастую ведет к появлению ошибок и непредсказуемому поведению.

Эволюционное расширение программ во многом связано с использованием динамического полиморфизма, который обеспечивает обработку ветвей программы во время ее выполнения, не используя при этом явной проверки альтернативных вариантов. Это отличает его от статического полиморфизма, при котором альтернативные вычисления выявляются во время компиляции. Изначально решения, связанные с реализацией динамического полиморфизма, были предложены для объектно-ориентированной (ОО) парадигмы. Например, в бестиповых и динамически типизированных ОО языках используется «утиная типизация». Ее идея заключается в том, что если любые, даже несвязанные объекты, имеют одинаковый по сигнатуре метод, они, каждый по своему, могут обработать обращение к нему. Это решение реализовано в языках программирования Smalltalk [1], Python [2] и многих других.

В ОО языках со статической типизацией ключевым решением стало совместное использование механизмов наследования и виртуализации. Наследование обеспечило идентичность интерфейсов родительского и дочернего классов, а виртуализация позволила подменять реализацию методов в дочерних классах. Этот подход реализован в языках ОО программирования C++ [3], Java [4], C# [5] и других.

Поддержку динамического полиморфизма, основанную на иных принципах, стали включать и в процедурные языки. В языке Go [6], реализован механизм интерфейсов, позволяющий использовать в качестве альтернативных обработчиков функции, связанные со структурами данных. Аналогичный механизм на основе типажей реализован в языке программирования Rust [7].

Инструментальная поддержка эволюционной разработки во многом способствовала популярности объектно-ориентированного программирования (ООП). В частности, были предложены паттерны проектирования [8], обеспечивающие гибкое расширение программы. Вместе с тем следует отметить, что безболезненное расширение программы в рамках ООП имеет определенные ограничения. Например, невозможно прямое расширение мультиметодов. Предлагаемые для этого решения [9—11] в основном ведут к использованию дополнительных конструктивов и алгоритмов, при которых, чаще всего, объектно-ориентированный стиль смешивается с процедурным. Аналогичные проблемы, связанные с реализацией мультиметодов, существуют также в языках программирования Go и Rust.

Это ограничение обусловлено тем, что мультиметод по сути является внешней функцией, поддерживающей множественный динамический полиморфизм над своими аргументами, в отличии от виртуального метода класса, интрефейсов Go и типажей Rust, которые поддерживают только одиночный полиморфизм над своими программными объектами, являясь по сути монометодами. Следует отметить, что множественный динамический полиморфизм был реализован в языке ОО программирования CLOS [12]. Однако эта реализация оказалась излишне громоздкой, что не привело к ее заимствованию в других ОО языках, в частности, в C++ [13].

Для инструментальной поддержки эволюционно расширяемого множественного полиморфизма была предложена процедурно-параметрическая парадигма программирования [14]. Это позволило повысить возможности процедурного подхода в области разработки ПО. Решение базируется

на параметрическом механизме формирования отношений между данными и обрабатывающими их процедурами, который может быть реализован различными способами [15]. Это обеспечивает безболезненное расширение как данных, так функций, используя при этом статическую типизацию. Для апробации идеи разработан язык O2M, расширяющий язык программирования Оберон-2 [16]. Проведенные на его основе эксперименты по написанию кода позволили проанализировать возможности процедурно-параметрического программирования (ППП). Язык включает все возможности Оберона-2, и позволяет использовать его модули. Данная реализация также показала, что подход может быть достаточно безболезненно интегрирован как в уже существующие языки процедурного и функционального программирования, так и использоваться при разработке новых языков. Также показано, что процедурно-параметрический полиморфизм обеспечивает более гибкое эволюционное расширение программ по сравнению с другими методами поддержки динамического полиморфизма [17].

Вместе с тем следует отметить, что процедурно-параметрическая парадигма не получила распространения. Во многом это обуславливается как отсутствием широкого практического использования языков семейства Оберон, так и тем, что основные идеи и технологии базируются на других языках программирования, получивших более широкое применение. Нельзя сказать, что процедурное программирование в настоящее время находится в полном упадке. Например, язык программирования C входит в пятерку наиболее популярных языков по различным рейтингам, что обуславливается его гибкостью и относительной простотой, а также использованием в предметных областях, где ОО подход не является эффективным. В связи с этим в работе рассматривается решение, направленное на включение в язык программирования C конструкций, поддерживающих ППП. Предлагаются дополнительные синтаксические конструкции, ориентированные на поддержку предлагаемого подхода. Описываются их семантика, возможности и особенности отображения в более простые конструкции, применяемые в архитектурных решениях уровня системы команд.

1. Основные концепции процедурно-параметрической парадигмы

При описании особенностей процедурно-параметрической парадигмы используются следующие понятия:

- основа специализации;
- специализация обобщения;
- параметрическое обобщение;
- экземпляр параметрического обобщения или специализированная переменная;
- обобщающая функция;
- обработчик параметрической специализации или специализированная функция;
- вызов параметрической процедуры.

1.1. Основа специализации

Под основой специализации понимается любой независимый базовый или составной тип имеющий имя. Практически любой тип может использоваться в качестве типа специализации обобщения. Также в качестве основы может использоваться и параметрическое обобщение, которое по сути тоже является составным типом. Понятие основы не накладывает никаких ограничений на самостоятельное использование этих типов в программе и вводится только для терминологической стыковки с другими вводимыми понятиями. Они, как и ранее, остаются обычными типами.

В качестве примеров структур, используемых далее в качестве основ специализаций, можно привести представления треугольника, прямоугольника и круга:

```
struct Triangle {int a, b, c;};   // Треугольник
struct Rectangle {int x, y;};     // Прямоугольник
struct Circle {int r;};           // Круг
```

1.2. Специализация обобщения

Под специализацией обобщения (или просто «специализацией») понимается любая из основ специализации, включенная в качестве составной части в параметрическое обобщение. Каждая специализация задает в обобщении один из альтернативных вариантов реализации. По своей сути специализация близка по семантике к альтернативному полю объединения языка программирования C. Помимо этого, при определенных условиях, в качестве специализаций в обобщение могут включаться и неименованные структурные типы.

1.3. Параметрическое обобщение

Параметрическое обобщение (или просто «обобщение») может содержать несколько специализаций. При этом обобщение может расширяться путем добавления новых специализаций в различных единицах компиляции, что и отличает его от объединения языка C. В предлагаемом расширении данного языка оно строится на основе модификации структуры и называется обобщенной структурой, синтаксис которой можно описать следующим образом:

```
ОбобщеннаяСтруктура = Структура
     "<" СписокСпециализаций ">" ["const"] |
     "<" [ШаблонПризнака] ">".
Шаблон признака = ":".
СписокСпециализаций = СписокСпециализацийПоТипу | СписокСпециализацийПоПризнаку.
СписокСпециализацийПоТипу = ИмяТипа ";" {ИмяТипа ";"}.
СписокСпециализацийПоПризнаку = Признак {"," Признак} ":" Тип ";"
                                {Признак {"," Признак} ":" Тип ";"}.
Признак = идентификатор.
Тип = ИмяТипа | ОписаниеНеименованногоТипа.
```

Как и при описании обычных структур, обобщенные структуры также можно использовать в описании `typedef`.

В отличие от обычных структур обобщение дополнительно содержит список специализаций. Уникальность при этом может определяться уникальностью типов, задающих различные специализации, или уникальностью признаков. Во втором случае допускается множество альтернатив имеющих одинаковый тип, а также использование в качестве типов неименованных структур, непосредственно определяемых в обобщении.

Если необходимо сформировать обобщенную структуру, которая в дальнейшем не должна изменяться, то для этого можно использовать квалификатор `const`, следующий непосредственно после описания специализаций. В качестве примера можно привести обобщенную структуру, задающую дни недели:

```
// Дни недели
struct WeekDay {int week_number;}
    <Monday, Tuesday, Wednesday, Thursday, Friday, Saturday, Sunday: void;>
    const;
```

В примере, за счет использования типа `void`, осуществляется имитация перечисления. Различные признаки, предшествующие типу, позволяют задать только семь альтернатив, с каждой из которой можно впоследствии связать свой полиморфный обработчик специализации. При отсутствии квалификатора `const` можно сымитировать эволюционно расширяемый перечислимый тип. Общее поле в основной структуре задает номер недели в году.

Зачастую появление в программе новых специализаций может осуществляться не одновременно, а в процессе ее разработки. В таких случаях целесообразно иметь возможность расширения набора специализаций обобщения по мере необходимости. Пусть на первом этапе необходимо сформировать специализацию обобщения для треугольника. Это можно осуществить созданием следующей первоначальной обобщенной структуры:

```
struct Figure {}<struct Triangle;>;
```

Допускается использовать первую часть обобщенной структуры без задания полей, что указывает на отсутствие общих данных у всех формируемых обобщений.

Последующее расширение обобщения может осуществляться как путем одиночного, так и группового включения новых специализаций. Описание их подключения осуществляется в соответствии со следующими синтаксическими правилами:

```
ДобавлениеСпециализаций = ССылкаНаОбобщеннуюСтруктуру
                        "+" "<" СписокСпециализаций ">".
```

Тогда добавление прямоугольника и круга может выглядеть следующим образом:

```
struct Figure + <struct Rectangle; struct Circle;>;
```

Возможна ситуация, когда первоначальное обобщение создается пустым. В этом случае, при задании признаков специализаций типами, их список будет отсутствовать, а его наполнение начнется с добавлением специализаций. Для текущего примера первоначальное формирование специализации – треугольника может быть реализовано следующим образом:

```
// В обобщенной структуре отсутствуют специализации
struct Figure {}<>;
// Включение треугольника в обобщенную фигуру
struct Figure + <struct Triangle;>;
```

Использование идентификации посредством признаков специализаций позволяет многократно использовать один и тот же тип. Пусть структура прямоугольника используется для задания ромба через его диагонали, а для задания отрезка через его длину используем структуру, определяющую круг. Помимо этого определим дополнительные имена типов основ специализаций с применением `typedef`:

```
typedef struct Triangle {int a, b, c;} Triangle;   // Треугольник
typedef struct Rectangle {int x, y;} Rectangle;    // Прямоугольник
typedef struct Circle {int r;} Circle;             // Круг
```

При изначально пустом наборе специализаций обобщения и использовании для идентификации специализаций признаков шаблон будет выглядеть следующим образом:

```
typedef struct Figure2 {}<:> Figure2;
```

Дальнейшее расширение обобщения, можно описать следующим образом:

```
// Добавление прямоугольника, ромба, отрезка
Figure2 + <rect, rhomb: Rectangle; section: Circle;>;
// Добавление треугольника и круга
Figure2 + <trian: Triangle; circ: Circle;>;
```

На первом этапе обобщение Figure2 расширяется за счет прямоугольника, ромба и отрезка указанной длины. В следующим описании добавляются треугольник и круг. Расширения обобщения могу осуществляться в разных единицах компиляции, в каждой из которых будут видны только свои специализации.

1.4. Экземпляры параметрических обобщений

Экземпляры параметрических обобщений являются переменными, сформированными на основе специализаций (специализированными переменными). Для каждой из таких переменных задается тип специализации. В ходе их описания допускается начальная инициализация. Возможно формирование как скалярных специализированных переменных, так и массивов различной размерности. Помимо этого можно задавать описание указателей на специализированные переменные, а также создавать массивы таких указателей. Указатели на специализации при этом могут ссылаться только на соответствующие специализированные переменные в отличие от указателей на параметрические обобщения, которые могут указывать на любые специализации, сформированные от соответствующего обобщения.

```
ОписаниеСпециализированнойПеременнной =
    ССылкаНаОбобщеннуюСтруктуру "<" [ ИмяТипа | Признак ] ">"
                        СписокСпециализированныхПеременных.
СписокСпециализированныхПеременных =
    СпециализированнаяПеременная {"," СпециализированнаяПеременная}.
СпециализированнаяПеременная =
    {"*"} ИмяСпециализированнойПеременной {"[" Размерность "]"}
        ["=" Инициализатор].
ССылкаНаОбобщеннуюСтруктуру = struct ИмяОбобщенной структуры | ИмяTypedef.
```

Размерность массива задается аналогично описанию размерности для обычных переменных языка C. Инициализация определяется в зависимости от размерности и типа формируемой переменной.

Запрещается создание переменных непосредственно от обобщения. То есть переменных, имеющих только структурную часть при отсутствии части, определяемой ее специализацией. Однако при необходимости можно добавлять в описание обобщений специализаций с признаком типа void или, при использовании явных признаков специализации, формировать аналогичный по назначению признак с типом void. Например, empty:void. Указатели на обобщения могут ссылаться на любые специализации данного обобщения, что указывает на возможность разыменования. Соответствующие действия могут осуществляться в ходе начальной инициализации или выполнения операции присваивания.

```
ОписаниеУказателейНаОбобщения = ТипОбобщения
                        СписокОбобщенныхУказателей.
СписокОбобщенныхУказателей = ОбобщенныйУказатель {"," ОбобщенныйУказатель}.
ОбобщенныйУказатель =
    "*"{"*"} ИмяОбобщенногоУказателя {"[" Размерность "]"}["=" Инициализатор].
```

Как и любые переменные языка C, обобщенные и специализированные переменные могут быть глобальными, статическими, локальными или динамическими. Их значения могут использоваться в качестве фактических параметров при вызове функций.

В качестве примера можно привести следующие варианты создания специализированных переменных и обобщенных указателей:

```
struct Figure<struct Triangle> t1 = {}<{3,4,5}>, t2;
struct Figure<struct Circle> c1 = {}<{10}>, *pc1 = &c1, c[10];
Figure2<rect> r1, r2[3] = {{}<{1,2}>,{}<{3,4}>,{}<{5,6}>},
              r3 = <{7,4}>, pr1 = &r1;
struct Figure *pf1 = &t1, **ppf1 = &pf1;
struct WeekDay<Monday> monday = {35}<>; // Меняется только номер недели
```

К особенностям инициализации специализаций можно отнести указание описываемых значений внутри угловых скобок. При этом для структурных значений необходимо также указывать дополнительные фигурные скобки, которые не нужны, если специализация является скаляром.

1.4.1. Рекурсивное расширение специализаций

Построение новых специализаций может также осуществляться на основе обобщенной структуры, что позволяет формировать цепочки уточнений произвольной длины. При этом следует отметить, что подобное введение новых специализаций возможно только в том случае, если предшествующий тип является обобщенным. Это позволяет контролировать процесс добавления новых уточнений во время компиляции. Например, для формирования новой ступени обобщения T можно включить в него обобщение T0:

```
struct T {int x, y;}<:>;
struct T + <t0: _Bool; t1: struct{double r; char s;}>;

struct T0 {int z;}<:>;
struct T + <t2: T0;>;
```

Тип T0 можно будет уточнять, добавляя для этого к нему новые специализации, которые также могут содержать обобщения, обеспечивающие их дальнейшее уточнение:

```
struct T00 {int a;}<:>;
struct T0 + <t00: T00;>;
```

Использование данного приема позволяет выстраивать сложные зависимости между типами, воспринимая при этом различные специализации как уточнения одного и того же типа. Для приведенного примера могут быть сформированы следующие специализации:

- из T ⟶ T<t0>, или T<t1> или T<t2>,
- из T<t2> ⟶ T<t2<t00>> и т. д.

Допускается также рекурсивное подключение к существующим обобщениям других обобщений, включая и подключение самого себя. При необходимости это позволяет выстраивать длинные статические цепочки, формируемые во время компиляции программы. В качестве примера можно расширить тип T специализацией, построенной на основе этого же типа:

```
struct T + <t3: T>;
T += t3: T;
```

Появляется возможность выстраивать специализации, обеспечивающие поддержку возможностей, эквивалентных декорированию:

```
T<t3>, T<t3<t3>>, T<t3<t3<t3<t3<t2<t00>>>>>>, ...
```

1.4.2. Операции над специализированными переменными

Над специализированными переменными, а также над специализированными и обобщенными указателями возможны операции доступа к отдельным полям для чтения данных и присваивания. Допускается манипуляция над отдельными полями как структурной части, так и специализации. При этом для полей структурной части специализированной переменной в качестве разделителя выступает, как обычно, точка (.). Поле обобщенной части отделяется от обозначения переменной восклицательным знаком (!). Например:

```
t1!a = 5;
t2 = t1;
pf1 = pc1;
monday.week_number = 24; // Понедельник 24-й недели
struct T<t0> b = {}<1>;  // Инициализация специализации базового типа
b! = 0;                  // Изменение значения специализации базового типа
```

В случаях, когда поля специализаций являются структурами, обращение к ним осуществляется с указанием имени поля, перед которым ставится точка. Обращение к структурной части специализированной переменной осуществляется также как и к обычной структурной переменной.

Операции над указателями на обобщения включают проверку типа специализации. Это напрямую позволяет определить основу специализации и осуществить корректное явное приведение к нужному типу для организации доступа к полям. Операция реализуется как функция _Spec_is, возвращающая булевское значение:

```
ПроверкаТипаСпециализации =
    "_Spec_is" "(" УказательНаОбобщение, ИмяТипа | Признак ")".
```

В случае, если признак специализации соответствует проверяемому типу, функция возвращает значение, равное 1. В противном случае возвращается 0.

1.5. Обобщающие параметрические функции

Обобщающие функции обеспечивают поддержку процедурно-параметрического полиморфизма. Их сигнатуры определяют единый интерфейс для обработчиков параметрических специализаций. Каждая обобщающая функция имеет от одного до нескольких обобщенных формальных параметров. Они задаются в виде указателей на обобщения. При вызове такой функции в нее передаются ссылки на специализированные переменные, комбинации которых и определяют конкретный обработчик специализации. Обобщающая параметрическая функция описывается следующими синтаксическими правилами:

```
ОбобщающаяФункция = ТипВозвращаемогоПараметра ИмяФункции
    "<" СписокОбобщенныхПараметров ">" "(" [ СписокФормальныхПараметров ] ")"
        ТелоОбобщающейФункции.
СписокОбобщенныхПараметров = ОбобщенныйПараметр {"," ОбобщенныйПараметр }.
ТелоОбобщающейФункции = ПустоеТело | ОбработчикПоУмолчанию.
ПустоеТело = "=" "0".
ОбобщенныйПараметр = ТипОбобщения "*" ИмяОбобщения.
```

Например, обобщающая функция вывода на печать геометрической фигуры может быть представлена следующим образом:

```
void PrintFigure<struct Figure *f>() = 0;
```

Обобщенные параметры задаются в угловых скобках (в отличие от прочих параметров, которые, как обычно, располагаются в круглых скобках при их наличии). Отсутствие тела в данном случае говорит о том, что необходимо написать обработчики специализаций для всех конкретных фигур, представленных параметрическим обобщением. Помимо этого может быть задан обработчик по умолчанию, который вызывается в тех случаях, когда для подставляемой комбинации специализаций не будет реализован свой обработчик. Простейшим вариантов подобного обработчика в таком случае может служить вызов функции прерывания программы:

```
void PrintFigure<struct Figure *f>() {
    printf("Incorrect Argument!!!\n")
    exit(-1);
}
```

Предполагается также, что перед вызовами обобщающих функций компилятором будет автоматически генерироваться код, проверяющий указатели.

1.6. Обработчики параметрических специализаций

Обработчики параметрических специализаций (или специализированные функции) вызываются при обращении к обобщающей функции. Они обеспечивают непосредственную манипуляцию конкретными специализациями, включенными в состав параметрических обобщений. Синтаксис обработчиков:

```
СпециализированнаяФункция = ТипВозвращаемогоПараметра ИмяФункции
    "<" СписокСпециализаций ">" "(" СписокФормальных параметров ")"
        ТелоСпециализированнойФункции.
СписокСпециализаций = СпециализированныйПараметр
    {"," СпециализированныйПараметр }.
СпециализированныйПараметр = ТипСпециализации "*" ИмяСпециализации.
```

Выбор обработчика осуществляется в зависимости от значений параметрических аргументов, подставляемых вместо формальных параметров. Они могут располагаться в разных единицах компиляции, добавляясь, например, по мере создания очередной специализации, и для обобщающей функции печати фигуры выглядеть следующим образом:

```
// Вывод параметров прямоугольника
void PrintFigure<struct Figure<struct Rectangle> *r>() {
    printf("Rectangle: x = %d, y = %d", r->!x, r->!y);
}
// Вывод параметров треугольника
void PrintFigure<struct Figure<struct Triangle> *t>() {
    printf("Triangle: a = %d, b = %d, c = %d", (*t)!a, (*t)!b, (*t)!c);
}
// Вывод параметров круга
void PrintFigure<struct Figure<struct Circle> *r>() {
    printf("Circle: r = %d", c->!r);
}
```

Следует отметить, что восклицательный знак, определяющий доступ к полю обобщенной части, ставится после знака указателя.

1.7. Вызовы параметрических функций

Вызов функции по сути запускает обработчик для конкретной комбинации специализаций. Он отличается от вызова обычной функции только наличием списка дополнительных фактических параметров, указывающих на специализации, по которым автоматически осуществляется выбор обработчика, что и определяет динамический полиморфизм. Вызов параметрической функции имеет следующий синтаксис:

```
ВызовПараметрическойФункции = ИмяФункции
        "<" СписокУказателейНаСпециализации ">"
        "(" [СписокФактическихПараметров] ")".
```

В качестве примеров можно привести вызов функции печати фигуры для различных выше описанных специализированных переменных:

```
// печать параметров треугольника t1
PrintFigure<&t1>();
// печать параметров круга c1
PrintFigure<&c1>();
// печать параметров круга c1 через указатель pc1
PrintFigure<pc1>();
// печать параметров круга c[5]
PrintFigure<&c[5]>();
// печать параметров круга c[5] через смещение указателя
PrintFigure<c+5>();
// печать параметров треугольника t1 через обобщенный указатель pf1
PrintFigure<pf1>();
// печать параметров треугольника t1 через указатель на указатель *ppf1
PrintFigure<*ppf1>();
```

2. Эволюционная поддержка множественного полиморфизма

Основное достоинство процедурно-параметрической парадигмы проявляется при эволюционном расширении мультиметодов. В качестве примера рассмотрим создание мультиметода, проверяющего возможность размещения первой фигуры внутри второй. Пусть изначально имеется информация только о специализациях прямоугольника и треугольника, заданных с использованием признаков и описанием typedef. В этом случае обобщающая функция может иметь следующий вид:

```
// Обобщенная функция, требующая обязательного переопределения
// для всех специализаций
_Bool Multimethod<Figure2 *first, Figure2 *second>() = 0;
```

Для ее видимости в других единицах компиляции, используемых для определения обработчиков специализаций достаточно прототипа:

```
_Bool Multimethod<Figure2*, Figure2*>();
```

В одной или нескольких единицах компиляции можно сформировать четыре функции, осуществляющие обработку различных комбинаций специализаций. Каждый обработчик специализации описывает одну комбинацию параметров и определяет метод ее обработки:

```
// Прямоугольник разместится внутри прямоугольника
_Bool Multimethod<Figure2<rect> *r1, Figure2<rect> *r2>() {
    return ((r1->!x < r2->!x) && (r1->!y < r2->!y)) ||
           ((r1->!x < r2->!y) && (r1->!y> < r2->!x));
}
// Прямоугольник разместится внутри треугольника
_Bool Multimethod<Figure2<rect> *r1, Figure2<trian> *t2>() {...}
// Треугольник разместится внутри прямоугольника
_Bool Multimethod<Figure2<trian> *t1, Figure2<rect> *r2>() {...}
// Треугольник разместится внутри треугольника
_Bool Multimethod<Figure2<trian> *t1, Figure2<trian> *t2>() {...}
```

Более сложные алгоритмы заменены многоточием. Приведенный пример показывает, что специализации, тип которых известен во время компиляции, обрабатываются так же, как и обычные переменные эквивалентного типа.

Добавление в программу специализаций для новых геометрических фигур осуществляется без изменений существующих единиц компиляции. Например, появление круга приведет к реализации дополнительных обработчиков специализаций в новых файлах, подключаемых к проекту:

```
// Прямоугольник разместится внутри круга
_Bool Multimethod<Figure2<rect> r1*, Figure2<circ> *c2>() {
    return ((r1->!x*r1->!x + r1->!y*r1->!y) < (c2->!r*c2->!r));
}
// Треугольник разместится внутри круга
_Bool Multimethod<Figure2<trian> *t1, Figure2<circ> *c2>() {...}
// Круг разместится внутри прямоугольника
_Bool Multimethod<Figure2<circ> *c1, Figure2<rect> *r2>() {...}
// Круг разместится внутри треугольника
_Bool Multimethod<Figure2<circ> *c1, Figure2<trian> *t2>() {...}
// Круг разместится внутри круга
_Bool Multimethod<Figure2<circ> *c1, Figure2<circ> *c2>() {
    return c1->!r < c2->!r;
}
```

3. Отображение процедурно-параметрических конструкций в низкоуровневое представление

Перед реализацией генератора кода, обеспечивающего трансформацию процедурно-параметрических конструкций в машинный код необходимо провести анализ возможных вариантов окончательно формируемого представления. Для этого можно непосредственно использовать язык программирования C, программные объекты которого практически однозначно отображаются на компьютерную память. Помимо этого использование данного языка позволяет гораздо проще оценить эффективность различных вариантов отображения. Основной идеей при этом, как и в объектно-ориентированных системах, является создание программных объектов и связей между ними, обеспечивающих поддержку новой парадигмы, до запуска функции `main`. Это обеспечивается за счет возможностей операционной системы Linux, использовать аналоги конструкторов, выполняемых до запуска `main`, и деструкторов, запускаемых после ее завершения [18].

3.1. Трансформация обобщений

Параметрические обобщения преобразуются в структуры, содержащие общие для всех специализаций поля. Помимо этого они содержат дополнительное поле, в котором фиксируется признак специализации. Признак задается числом от нуля до количества специализаций, окончательное число которых может формироваться различным образом: либо на этапе сборки программы, либо во время ее запуска, но до выполнения функции `main`. Например, описанное выше обобщение, задающее геометрическую фигуру, можно трансформировать в следующие программные объекты языка C:

```
typedef struct figure {
  unsigned tag;       // тег, задающий признак специализации
  struct {} head;     // первичная структура, возможен список полей
} figure;
```

Помимо этого для каждого обобщения создается статическая переменная, в которой фиксируется количество специализаций, сформированных в различных единицах компиляции:

```
static unsigned _figure_number_ = 0;
```

Для получения ее значения и изменения при регистрации новых специализаций порождаются функции, через которые и осуществляется доступ:

```
// Регистрация очередной специализации
void _figure_spec_register(void (*set_spec_tag)(unsigned));
// Получение текущего числа специализаций
unsigned get_figure_number();
// Возврат признака специализации через указатель на обобщение
unsigned get_figure_tag(figure *fig);
```

Прототипы этих функций определены в заголовочном файле вместе с описанием структуры обобщения и подключаются к единицам компиляции, отвечающим за установку параметров специализаций. Переменная `_figure_number_` и определения выше объявленных функций находятся в отдельной единице компиляции:

```
// Регистрация очередной специализации фигуры
void _figure_spec_register(void (*set_spec_tag)(unsigned)) {
    // Текущее значение становится тегом специализации
    set_spec_tag(_figure_number_);
    ++_figure_number_;  // изменение числа зарегистрированных фигур
}
// Возврат текущего количества зарегистрированных специализаций
unsigned get_figure_number() {
    return _figure_number_;
}
// Возврат признака специализации через указатель на обобщение
unsigned get_figure_tag(figure *fig) {
    return fig->tag;
}
```

Передаваемая через указатель `set_spec_tag` функция используется для получения каждой разновидности специализации своего тега. Эта функция непосредственно связана с переменной, идентифицирующей конкретную специализацию. После регистрации очередной специализации их общее количество увеличивается на единицу.

3.2. Трансформация специализаций

Специализации обобщений формируются по одному и тому же подходу. Описание каждой специализации и ее реализация находятся в заголовочном файле и в файле реализации. В качестве примера рассмотрим специализацию фигуры как прямоугольника. Пусть данная специализация использует в качестве основы ранее описанный прямоугольник:

```
struct rectangle {int x, y;};   // основа специализации
```

Будем считать, что при создании специализации обобщения формируется уникальное имя, не совпадающее с именами конструкций, определяемых программистом. Формируемый специализированный прямоугольник может напрямую не использовать исходную фигуру, повторяя при этом ее поля:

```
typedef struct figure_rectangle {
    // Повторение полей обобщения
    unsigned tag;   // поле тега на том же месте
    struct {} head; // поле структуры, идентичное структуре обобщения
    struct rectangle tail; // поле, определяющее содержание специализации
} figure_rectangle;
```

Помимо структуры задаются прототипы функций, используемые данной специализацией:

```
// Установщик признака специализации, передаваемый регистратору
void set_figure_rectangle_tag(unsigned tag);
// Получение признака специализации. Необходим при регистрации обработчиков
unsigned get_figure_rectangle_tag();
// Инициализация признака спецпеременной перед использованием
void init_figure_rectangle(figure_rectangle *p_fr);
```

Реализация этих функций и статической переменной `_figure_rectangle_tag`, хранящей признак данной специализации, размещаются в отдельной единице компиляции:

```
// Признак специализации. Доступен через интерфейсные функции
static unsigned _figure_rectangle_tag = -1;

void set_figure_rectangle_tag(unsigned tag) {
    _figure_rectangle_tag = tag;
}
unsigned get_figure_rectangle_tag() {
    return _figure_rectangle_tag;
}
void init_figure_rectangle(figure_rectangle *p_fr) {
    p_fr->tag = _figure_rectangle_tag;
}
```

В этой же единице компиляции определяется функция `register_figure_rectangle_tag`, осуществляющая регистрацию данной специализации. Именно она формирует значение признака до запуска функции `main`, используя специфику запуска программы операционной системой:

```
// Конструктор, обеспечивающий создание признака специализации
void  __attribute__ ((constructor(101))) register_figure_rectangle_tag() {
    _figure_spec_register(set_figure_rectangle_tag);
}
```

Число 101 определяет приоритет. Он одинаков для всех конструкторов данного типа, осуществляющих регистрацию специализаций, так как порядок их регистрации не играет значения. Основным ограничением является то, что приоритет должен быть больше 100, так как меньшие значения зарезервированы за операционной системой.

3.3. Трансформация обобщающих функций

Обобщающие функции предоставляют общий интерфейс для соответствующих обработчиков специализаций, предоставляя тем самым общую основу для динамического полиморфизма. Помимо этого они используются для формирования действий, связанных с обработкой по умолчанию. Последнее осуществляется в том случае, если обработчик специализации отсутствует. Основная идея обобщающей функции заключается в вызове обработчика специализации в соответствии с комбинацией специализаций, используемых в качестве фактических параметров. При этом идентификация нужного обработчика определяется по признакам специализаций. Возможны различные варианты реализации обобщающих функций и их подмены обработчиками специализаций.

1. Обобщающая функция вызывает нужные действия, обращаясь к обработчикам специализаций через массив указателей на соответствующие функции. Данный подход обеспечивает эффективность, эквивалентную скорости доступа к виртуальным методам при объектно-ориентированном подходе. Но в тех случаях, когда приоритетным является использование обработчика по умолчанию, хранение указателей в массиве может оказаться неэффективным.
2. Выбор нужной специализации может осуществляться через обход списка указателей с поиском нужных комбинаций признаков, например, используя для ускорения хеширование или древовидные схемы. Однако в данном случае скорость поиска нужной комбинации сильно зависит от числа обработчиков специализаций.
3. Возможна централизованная реализация с применением условных операторов или переключателей. Данный подход по сути является решением, типичным для процедурного программирования. Несмотря на свою простоту он требует знания всех специализаций и не поддерживает эволюционного расширения.

В качестве примера рассмотрим реализацию обобщающей функции вывода параметров геометрических фигур на основе массива указателей. Объявление обобщающей функции `print_figure` представлено в заголовочном файле:

```
// Одномерный массив указателей на обработчики специализаций
typedef void (*print_figure_pointer)(figure* p_f);
// Обобщающая функция
void print_figure(figure* p_f);
// Регистрация обработчика специализации в массиве обработчиков
void _print_figure_spec_register(
        unsigned (*get_spec_tag)(), print_figure_pointer spec);
```

Там же, для удобства использования, через объявление `typedef` приведено описание указателя `print_figure_pointer` на функции — обработчики специализаций, параметры которых совпадают с параметрами обобщающей функции. Помимо этого объявлен прототип функции, осуществляющей регистрацию обработчиков специализаций.

Описание указателя на массив обработчиков специализаций и определения приведенных выше функций представлено в отдельной единице компиляции. Указатель на массив обработчиков специализаций размещен в статической памяти и недоступен из других модулей программы:

```
static print_figure_pointer *_print_figure_array;
```

Доступ к нему осуществляется только через соответствующие функции, расположенные в этой же единице компиляции.

Сами массивы указателей создаются в конструкторах регистраторов обобщающих функций, формирующих массивы указателей на обработчики специализаций. Они имеют меньший приоритет (в данном случае он равен 201) по сравнению с конструкторами, регистрирующими специализации. Это позволяет создавать массивы, размерность которых соответствует количеству зарегистрированных специализаций. Конструктор, формирующий массив указателей на обработчики функций печати фигур реализован следующим образом:

```
void  __attribute__ ((constructor(201))) register_print_figure_array() {
    unsigned figure_number = get_figure_number();
    _print_figure_array =
        malloc(figure_number * sizeof(print_figure_pointer));
    for(unsigned i = 0; i < figure_number; ++i) {
        _print_figure_array[i] = print_figure_default;
    }
}
```

Получив количество зарегистрированных специализированных фигур, он формирует массив соответствующей размерности, заполняя его указателями на обработчик по умолчанию. Это позволяет осуществить корректный вызов обобщенной функции, если для каких-либо специализаций обработчик будет отсутствовать. Представленный ниже обработчик по умолчанию выводит сообщение об отсутствии функции вывода и завершает программу:

```
static void print_figure_default(figure* p_f) {
    printf("ERROR: print for figure is absent!\n");
    exit(1);
}
```

Выделение динамической памяти должно сопровождаться последующим ее освобождением по окончании выполнения функции `main` или выхода по функции `exit`. Для этого используется соответствующий деструктор:

```
void  __attribute__ ((destructor(201))) delete_print_figure_array() {
    free(_print_figure_array);
}
```

Регистрация обработчиков специализаций во многом аналогична регистрации специализаций. Имеется общий регистратор, которому передается указатель на функцию-обработчик:

```
void _print_figure_spec_register(
  unsigned (*get_spec_tag)(), print_figure_pointer spec) {
  // Передаваемый указатель на функцию фиксируется в массиве указателей
  unsigned tag = get_spec_tag();
  _print_figure_array[tag] = spec;
}
```

Запуск этого регистратора осуществляется конструктором специализации и рассмотрен ниже.

Обобщающая функция может вызвать любой из обработчиков специализаций. Она обращается к массиву обработчиков, используя в качества индексов теги специализаций, выступающих в роли обобщенных параметров. Для функции печати обобщенной фигуры функция выглядит следующим образом:

```
void print_figure(figure* p_f) {
    print_figure_pointer spec = _print_figure_array[p_f->tag];
    spec(p_f);
}
```

3.4. Трансформация специализаций

Приведенная схема позволяет полностью отделить обобщение от специализаций и их обработчиков, которые могут разрабатываться и эволюционно добавляться в отдельных единицах компиляции по одной и той же схеме. В качестве примера рассмотрим добавление в программу специализированного прямоугольника. Его основой служит структура, описывающая стороны прямоугольника:

```
struct rectangle {int x, y;};
```

Используя эту основу, в заголовочном файле можно описать специализацию:

```
typedef struct figure_rectangle {
  // Повторение полей обобщения
  unsigned tag;       // поле тега на том же месте
  struct {} head;     // поле структуры, идентичное структуре обобщения
  struct rectangle tail;  // поле, определяющее содержание специализации
} figure_rectangle;
```

В начале структуры, описывающей специализацию, повторяются поля, соответствующие обобщению. Это обеспечивает семантическую эквивалентность и возможность приведения к типу обобщенной фигуры при проведении различных манипуляций. После этого следуют данные, определяющие специализацию обобщения. В примере это структура, описывающая прямоугольник. Помимо этого в заголовочном файле описываются функции, связанные с обработчиком специализации:

```
// Установщик признака специализации
void set_figure_rectangle_tag(unsigned tag);
// Получение признака специализации. Необходимо в процессе регистрации фигуры
unsigned get_figure_rectangle_tag();
// Инициализация признака специализации перед использованием
void init_figure_rectangle(figure_rectangle *p_fr);
```

Эти функции определены в единице компиляции специализации.

Установщик признака передается на регистрацию специализации, которая выполняется с использованием функции `_figure_spec_register`.

```
void set_figure_rectangle_tag(unsigned tag) {
    _figure_rectangle_tag = tag;
}
```

Полученное в результате значение, являющееся общим для всех специализаций данного типа, сохраняется в статической переменной:

```
static unsigned _figure_rectangle_tag;
```

Доступ к значению переменной, для использования его в качестве признака, осуществляется посредством функции:

```
unsigned get_figure_rectangle_tag() {
    return _figure_rectangle_tag;
}
```

Помимо этого переменная используется для установки признака сформированного прямоугольника после его объявления в программе. Эта установка осуществляется функцией инициализации прямоугольника:

```
void init_figure_rectangle(figure_rectangle *p_fr) {
    p_fr->tag = _figure_rectangle_tag;
}
```

Сам процесс инициализации осуществляется конструктором, который, для корректного формирования количества специализаций, выполняется с наибольшим приоритетом:

```
void  __attribute__ ((constructor(101))) register_figure_rectangle_tag() {
    _figure_spec_register(set_figure_rectangle_tag);
}
```

3.5. Трансформация обработчиков специализаций

Добавление обработчиков специализаций связано с реализацией конкретных функций и их регистрацией в массивах обработчиков специализаций. В рассматриваемом примере функция вывода специализированного прямоугольника выглядит следующим образом:

```
static void print_figure_rectangle(figure* p_f) {
    figure_rectangle* p_fr = (figure_rectangle*)p_f;
    printf("rectangle: x= %d, y = %d\n", p_fr->tail.x, p_fr->tail.y);
}
```

В данном случае идет прямое обращение к специализации.

Для регистрации этой функции в массиве обработчиков обобщающей функции вывода используется соответствующий конструктор:

```
void  __attribute__ ((constructor(301))) register_print_figure_rectangle() {
    _print_figure_spec_register(get_figure_rectangle_tag,
                                print_figure_rectangle);
}
```

Для запуска после формирования массива указателей на специализации его приоритет задается ниже ранее описанных конструкторов.

3.6. Формирование процедурно-параметрических переменных

Специализированные переменные порождаются на основе абстракций, описывающих специализации. Их создание практически ничем не отличается от создания в языке C структурных переменных. Аналогичным образом для них осуществляется инициализация полей как общей, так и специализированной частей. Ключевым отличием является необходимость предварительной инициализации, при которой формируется признак специализации.

Инициализация внешних (глобальных и статических) переменных должна осуществляться до запуска функции `main`. Поэтому вызов соответствующей функции должен проходить внутри конструктора. Например, формирование переменной, описывающей прямоугольник можно представить следующим образом:

```
figure_rectangle r = {.tail.x=10, .tail.y=20};
// Конструктор, обеспечивающий инициализацию тега переменной r
void  __attribute__ ((constructor(401))) init_figure_rectangle_r() {
    init_figure_rectangle(&r);
}
```

Инициализация локальных переменных, а также переменных, размещаемых в динамической памяти осуществляется уже после запуска функции main. Например:

```
figure_rectangle r2 = {.tail.x=1, .tail.y=2};
init_figure_rectangle(&r2);

figure *p_f = (figure*)malloc(sizeof(figure_rectangle));
init_figure_rectangle((figure_rectangle*)p_f);
((figure_rectangle*)p_f)->tail.x = 100;
((figure_rectangle*)p_f)->tail.y = 200;
...
free(p_f);
```

3.7. Использование обобщающих функций

Процедурно-параметрический полиморфизм поддерживается использованием обработчиков специализаций, доступ к которым осуществляется через вызовы обобщающих функций. Каждая обобщающая функция обращается к специализациям через указатель на обобщение, запуская выбор нужного обработчика через признак специализации. При этом сам вызов нужного обработчика остается прозрачным и для представленных выше переменных может выглядеть следующим образом:

```
figure *p_f1 = (figure*)&r;
print_figure(p_f1); // Печать внешней специализированной переменной
p_f1 = (figure*)&r2;
print_figure(p_f1); // Печать локальной специализированной переменной
print_figure(p_f); // Печать переменной в динамически выделенной памяти
```

3.8. Варианты дальнейшего эволюционного расширения

Предлагаемый подход позволяет гибко наращивать функциональность программы, добавляя в нее как специализации, так и их обработчики без изменения ранее написанного кода. Например, добавление специализированного треугольника может формироваться в отдельных заголовочных файлах и соответствующих единицах компиляции по тому же принципу, что и формирование специализированного прямоугольника. Точно также в отдельной единице компиляции может быть сформирован обработчик, осуществляющий вывод специализированного треугольника.

Аналогичным образом осуществляется поддержка множественного полиморфизма. Допускается формирование обобщающих функций, аргументами которых являются два и более обобщения. Отличительной чертой в данном случае является то, что для регистрации обработчиков специализаций необходимо использовать многомерные массивы, размерность которых определяется количеством обобщенных аргументов функции. Вместе с тем их можно смоделировать через одномерные массивы с использованием для доступа формул, осуществляющих необходимый пересчет.

В качестве примера можно рассмотреть мультиметод, осуществляющий анализ вложенности первой геометрической фигуры во вторую. В заголовочном файле, как и ранее, описываются прототип обобщающей функции, осуществляющей запуск мультиметода, а также прототип регистратора

обработчиков специализаций. При этом регистратор размещает в массиве обработчики с использованием признаков первого и второго аргументов:

```
typedef void (*multimethod_figure_pointer)(figure* p_f1, figure* p_f2);
void multimethod_figure(figure* p_f1, figure* p_f2);
void _multimethod_figure_spec_register(unsigned (*get_spec_tag1)(),
                                       unsigned (*get_spec_tag2)(),
                                       multimethod_figure_pointer spec);
```

Эти функции определяются в соответствующей единице компиляции.

Помимо этого формируется одномерный массив, имитирующий матрицу обработчиков специализаций для двух аргументов. Размерность этого массива, а также формула, осуществляющая пересчет двумерной размерности в одномерную непосредственно прописаны в соответствующих функциях. Конструктор и деструктор реализуются аналогично тому, как это было сформировано для обобщающей функции вывода.

```
// Одномерный параметрический массив, имитирующий матрицу
static multimethod_figure_pointer *_multimethod_figure_array;
// Обобщающая функция, вызывающая обработчики специализаций
void multimethod_figure(figure* p_f1, figure* p_f2) {
  multimethod_figure_pointer spec =
    _multimethod_figure_array[get_figure_number()*p_f1->tag + p_f2->tag];
  spec(p_f1, p_f2);
}
// Обработчик по умолчанию, когда обработчик специализации отсутствует.
static void multimethod_figure_default(figure* p_f1, figure* p_f2) {
  printf("WARNING: Multimethod not implemented for this combination!\n");
}
// Конструктор, создающий матрицу указателей на обработчики специализаций
void
__attribute__ ((constructor(201))) register_multimethod_figure_array() {
  unsigned figure_number = get_figure_number();
  // Формируется матрица
  unsigned multimethod_array_size = figure_number * figure_number;
  _multimethod_figure_array =
    malloc(multimethod_array_size * sizeof(multimethod_figure_pointer));
  for(unsigned i = 0; i < multimethod_array_size; ++i) {
    _multimethod_figure_array[i] = multimethod_figure_default;
  }
}
// Деструктор, обеспечивающий освобождение памяти
void
__attribute__ ((destructor(201))) delete_multimethod_figure_array() {
  free(_multimethod_figure_array);
}
// Регистрация обработчика специализации в матрице
void _multimethod_figure_spec_register(unsigned (*get_spec_tag1)(),
                                       unsigned (*get_spec_tag2)(),
                                       multimethod_figure_pointer spec) {
```

```
  // printf("start _print_figure_spec_register\n");
  // Указатель на функцию фиксируется в массиве указателей
  unsigned tag = get_figure_number() * get_spec_tag1() + get_spec_tag2();
  _multimethod_figure_array[tag] = spec;
}
```

Добавление любого из обработчиков специализаций также осуществляется по единой схеме, обеспечивающей независимость от ранее написанного кода. Для регистрации мультиметода используется соответствующий конструктор

```
// Мультиметод для двух прямоугольников
static void
multimethod_figure_rectangle_rectangle(figure* p_f1, figure* p_f2) {
    printf("rectangle and rectangle\n");
}
// Регистрация мультиметода для двух прямоугольников
void  __attribute__
((constructor(301))) register_multimethod_figure_rectangle_rectangle() {
  _multimethod_figure_spec_register(get_figure_rectangle_tag,
                                    get_figure_rectangle_tag,
                                    multimethod_figure_rectangle_rectangle);
}
```

Дальнейшее расширение мультиметода осуществляется по аналогичной схеме.

Полный пример, демонстрирующий предлагаемых подход к реализации процедурно-параметрического полиморфизма с подробным описанием эволюционного расширения создаваемой программы представлен по ссылке [19].

4. Инструментальная поддержка процедурно-параметрической парадигмы

Приведенное выше моделирование эволюционного расширения программы на языке программирования C показывает, что основные подходы к реализации предлагаемых конструкций мало чем отличаются от тех, которые ранее были использованы при расширении языка Оберон-2, анализе методов реализации обобщенных записей, моделировании процедурно-параметрического полиморфизма на языке C++ [15—17, 20]. Вместе с тем, наличие для языка программирования C компиляторов и инструментальных средств, поддерживающих возможность интеграции с ними на уровне исходных текстов позволяет выбирать соответствующие инструменты и использовать их для повышения эффективности собственных разработок.

Одной из таких систем является семейство компиляторов clang [21]. Доступ к исходным текстам компилятора и предлагаемые программные интерфейсы позволяют непосредственно модифицировать синтаксический анализатор, обеспечивая добавление в него правил, распознающих синтаксис вводимых конструкций. На этапе семантического анализа имеется доступ к абстрактному синтаксическому дереву (АСД), структуру которого можно подкорректировать с учетом тех изменений, которые вносятся в семантику языка C процедурно-параметрическими расширениями. Эти модификации поддерживаются методами классов, используемых при формировании узлов АСД. При семантическом анализе предлагаемых конструкций в АСД добавляются объекты, расширяющие семантику обычных структур и функций до их процедурно-параметрических реализаций. При этом добавляемые объекты должны быть эквивалентны представленным выше конструкциям, описанным на языке C. Использование для изменения дерева классов, реализованных в `clang`, позволяет не изменять генератор кода из C в llvm [22], тем самым упрощая исходную задачу его формирования для дополнительно вводимых конструкций.

Заключение

Предлагаемое расширение языка программирования C конструкциями, обеспечивающими инструментальную поддержку процедурно-параметрической парадигмы программирования, позволяет разрабатывать эволюционно расширяемые программы. При этом обеспечивается добавление новых альтернативных данных, а также функций, позволяющих безболезненно для ранее написанного кода использовать множественный полиморфизм. Проведенное моделирование вносимых изменений, реализованное на языке программирования C в рамках операционной системы Linux подтверждает возможности такой реализации. При необходимости рассматриваемый подход может непосредственно использоваться для создания эволюционно расширяемых программ на процедурном языке программирования.

Реализация описанного решения возможна с использованием уже существующих инструментальных средств, таких как семейства компиляторов clang [21] и библиотеки libtooling [23], что позволяет осуществлять модификации как синтаксического анализатора, так и изменять в ходе анализа абстрактное синтаксическое дерево, адаптируя его под новые конструкции.

References

[1] D. Shafer and D. A. Ritz, *Practical Smalltalk. Using Smalltalk/V*, Springer-Verlag. 1991, p. 233.

[2] H. John, *Advanced Guide to Python 3 Programming*, Springer. 2019, p. 497.

[3] M. Gregoire, *Professional C++*, John Wiley & Sons. 2018, p. 1122.

[4] E. Sciore, *Java Program Design*, Apress Media. 2019, p. 1122.

[5] J. Albahari and B. Albahari, *C# 6.0 Pocket Reference: Instant Help for C# 6.0 Programmers*, O'Reilly Media. 2016, p. 224.

[6] A. Freeman, *Pro Go: The Complete Guide to Programming Reliable and Efficient Software Using Golang*, Apress. 2022, p. 1105.

[7] J. Blandy, J. Orendorff, and L. F. Tindall, *Programming Rust*, O'Reilly Media. 2021, p. 735.

[8] E. Gamma, R. Helm, R. Johnson, and J. Vlissides, *Design Patterns. Elements of Reusable Object-Oriented Software*, Addison-Wesley Professional. 1994, p. 416.

[9] A. Alexandrescu, *Modern C++ Design. Generic Programming and Design Patterns Applied.* Addison-Wesley Professional. 2001, p. 360.

[10] S. Meyers, *More effective C++. 35 New Ways to Improve Your Programs and Designs*, Addison-Wesley Professional. 1996, p. 318.

[11] A. Legalov, «Oop, multimethods and pyramidal evolution», *Open Systems*, no. 3, pp. 41–45, 2002, In Russian.

[12] L. Demichiel, «Overview: The common lisp object system», *Lisp and Symbolic Computation*, no. 1, pp. 227–244, 1989.

[13] B. Stroustrup, *Design and Evolution of C++*, Addison-Wesley Professional. 1994, p. 480.

[14] A. Legalov, *Procedurally-parametric programming paradigm. Is it possible as alternative to the object-oriented style?*, Dep. hands. Number 622-V00 Dep. VINITI 13.03.2000. 2000, p. 43, In Russian.

[15] I. Legalov, «Using of generalized records in procedural-parametric programming language», *Scientific Bulletin of the NSTU*, vol. 28, no. 3, pp. 25–37, 2007, In Russian.

[16] A. Legalov and D. Schvetc, «Procedural language with support for evolutionary design», *Scientific Bulletin of the NSTU*, vol. 15, no. 2, pp. 25–38, 2003, In Russian.

[17] A. Legalov and P. Kosov, «Evolutionary extension of programs using the procedural-parametric approach», *Computational Technologies*, vol. 21, no. 3, pp. 56–69, 2016, In Russian.

[18] *Linux x86 program start up or - how the heck do we get to main()?* [Online]. Available: http://dbp-consulting.com/tutorials/debugging/linuxProgramStartup.html.

[19] *An example of an evolutionary extension of a program using a procedural-parametric approach.* [Online]. Available: https://github.com/kreofil/c-evolution-example.

[20] A. Legalov, «Multimethods and paradigms», *Open Systems*, no. 5, pp. 33–37, 2002, In Russian.

[21] *Clang: A c language family frontend for llvm.* [Online]. Available: https://clang.llvm.org/.

[22] *The llvm compiler infrastructure.* [Online]. Available: https://llvm.org/.

[23] *Libtooling.* [Online]. Available: https://clang.llvm.org/docs/LibTooling.html.