Эволюционная разработка многопоточных программ с применением процедурно-параметрической парадигмы программирования

П.В. Косов[1], А.И. Легалов[1], В.С. Васильев[2]

[1]НИУ «Высшая школа экономики»,
[2]Сибирский федеральный университет

При создании параллельных программ, используемых во встроенных системах и в Интернете вещей зачастую применяются методы гибкой разработки. Одним из подходов к повышению гибкости является расширение функциональности за счет добавления новых программных объектов без изменения ранее написанного кода. Использование процедурно-параметрической парадигмы программирования позволяет во многих ситуациях безболезненно и независимо добавлять как данные, так и функции, предоставляя дополнительные возможности и технические приемы для разработки параллельных программ. В работе рассматривается использование данной парадигмы для многопоточного программирования на процедурно-параметрическом расширении языка программирования С.

Ключевые слова: параллельное программирование, многопоточное программирование, процедурно-параметрическая парадигма программирования, полиморфизм, гибкая разработка программного обеспечения.

1. Введение

При разработке программного обеспечения учитываются различные критерии качества. Не все они непосредственно связаны с параллелизмом, что обуславливается различными требованиями, предъявляемыми к конечному продукту. Вместе с тем, при создании параллельных программ обычно используются дополнительные библиотеки, которые прямо или косвенно могут влиять на достижение требуемых критериев. Это необходимо учитывать при применении различных подходов и парадигм программирования.

Расширение программ без изменения ранее написанного кода или с его минимальными изменениями является одним из критериев, определяющих гибкую разработку. Его важность обуславливается неполным знанием об окончательной функциональности программ во время их создания и начальной эксплуатации, а также необходимостью ускорить выход продуктов на рынок за счет инкрементального наращивания функциональности в каждом последующем релизе. В подобных ситуациях добавление новых конструкций, изменяющих уже написанный код, зачастую ведет к появлению неожиданных ошибок и непредсказуемому поведению. Ситуация может дополнительно усугубляться при использовании взаимодействующих параллельных процессов. Как следствие это ведет к увеличению стоимости разработки и сопровождения программной системы.

Одним из подходов, поддерживающих эволюционное расширение является динамическое связывание альтернативных программных объектов, обеспечивающее гибкое изменение данных во время выполнения. Он определяет специфику динамического полиморфизма, который отличается от статического полиморфизма, ориентированного на выявление альтернатив на этапе компиляции. Динамический полиморфизм, широко используется в объектно-ориентированном программировании (ООП). Основным решением в объектно-ориентированных (ОО) языках со статической типизацией является совместное использование наследования и виртуализации. Наследование поддерживает идентичность интерфейсов родительского и дочернего классов, а виртуализация позволяет подменять реализации методов в потомках. Подобный подход используется во многих современных ОО языках

программирования. В качестве примера можно привести языки C++ [1], Java [2] и другие. Поддержка динамического полиморфизма на основе статической утиной типизации включена и в процедурные языки. В Go [3] это механизм интерфейсов, позволяющий использовать для обработки альтернатив функции, связанные со структурами данных. Близкий механизм на основе типажей реализован в языке программирования Rust [4]. Вместе с тем, все эти подходы напрямую не поддерживают эволюционного расширения программ в случае множественного полиморфизма (мультиметодов) [5].

2. Процедурно-параметрическая парадигма

Еще один подход к реализации динамического полиморфизма предложен в рамках процедурно-параметрической парадигмы программирования (4П) [6]. Она предлагает гибкое эволюционное расширение как данных, так и функций, используя методы формирования обобщений и их специализаций на основе параметрического механизма. Подход может быть интегрирован в уже существующие языки процедурного и функционального программирования, а также использоваться при разработке новых языков. В настоящее время процедурно-параметрический (ПП) механизм реализован как расширение языка программирования C [7]. Данный язык широко используется в предметных областях, где ОО подход не является эффективным. Поэтому его дальнейшее развитие в направлении, связанном с поддержкой гибкой разработки программного обеспечения может оказаться перспективным.

В настоящее время методы кодирования, применяемые при разработке программ на языке C, могут приводить к ошибкам за счет отсутствия в ряде случаев контроля за типами данных во время компиляции. Такие ситуации возникают из-за того, что язык поддерживает произвольные и неконтролируемые преобразования типов. Например, в библиотеке потоков POSIX threads [8] многие функции для передачи параметров и возврата результатов используют указатели на `void`. Это приводит к ненадежному и неконтролируемому преобразованию типов. Подобный стиль кодирования снижает безопасность программ, хотя зачастую и позволяет достичь высокой производительности. Другой проблемой языка программирования C является отсутствие жесткой зависимости между альтернативными данными и идентифицирующими их признаками, что наблюдается, например, при использовании объединений (`union`). Признаки альтернатив могут произвольно использоваться программистом и приводить к ошибкам, выявляемым только во время выполнения.

Многие из этих проблем эффективно решаются в ОО языках программирования, а также в Go и Rust за счет использования динамического полиморфизма. ПП парадигма, наряду с решением аналогичных проблем, позволяет еще и эффективно поддерживать множественный полиморфизм, обеспечивая эволюционное расширение мультиметодов. Моделирование конструкций, расширяющих язык C, подтвердило возможности их реализации [7]. На основе данных, полученных в ходе моделирования, процедурно-параметрические конструкции были интегрированы в семейство компиляторов `clang` [9]. Описание предлагаемых расширений, обновляемое по мере вводимых изменений, размещено в сети Интернет [10].

3. Сопоставление объектно-ориентированного и процедурно-параметрического подходов

Основное отличие ПП подхода по поддержке динамического полиморфизма заключается в ином техническом решении, обеспечивающем формирование отношений между альтернативными данными и функциями как на уровне языковых конструкций, так и при их реализации в семантической модели языка. Это отличие можно рассмотреть в сравнении с методами поддержки ОО полиморфизма. Общие схемы, демонстрирующие отличие ОО и ПП подходов, представлены на рис. 1.

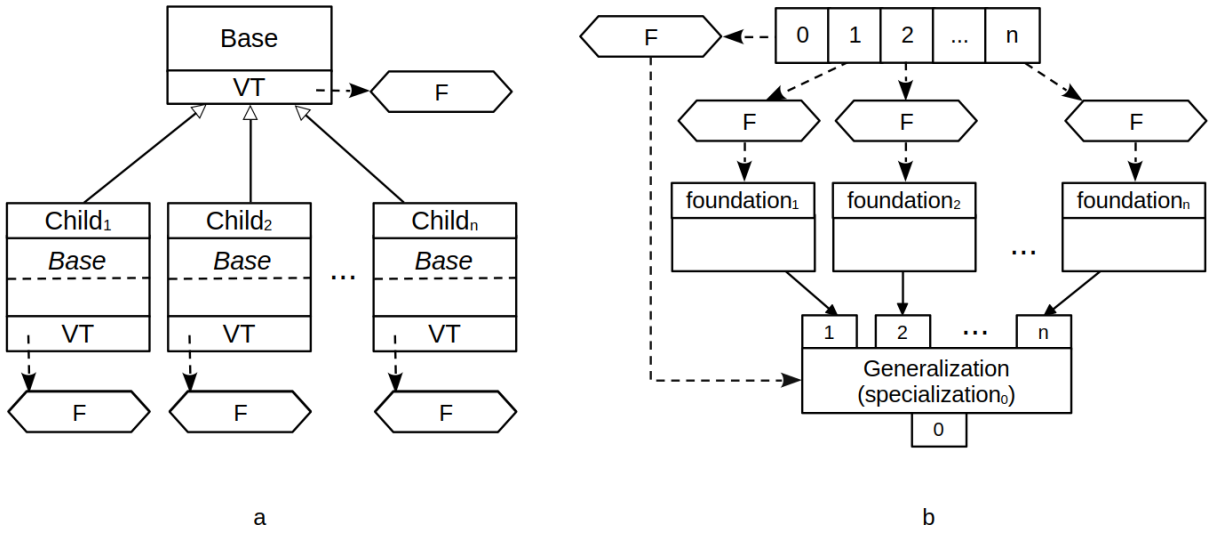

Рис. 1. Реализация объектно-ориентированного (a) и процедурно-параметрического (b) полиморфизма

Объектно-ориентированный полиморфизм (рис. 1a) формируется путем совместного использования наследования и виртуализации. Виртуализация при этом обычно реализуется за счет таблиц виртуальных методов (VT). Таблица базового класса $Base$, выступающего в роли обобщения, содержит указатели на один или несколько методов (F), которые обычно переопределяются в виртуальных таблицах производных классов, являющихся специализациями обобщения ($Child_1$–$Child_n$). При этом каждый производный класс, расширяя базовый класс, имеет собственный тип, что выражается в уникальном имени класса ($Child_i$). Производные классы формируются независимо друг от друга. Это позволяет эволюционно расширять альтернативные программные объекты, каждый из которых, при наличии одинаковых интерфейсов, может иметь иную функциональность, обеспечивая тем самым реализацию динамического ОО полиморфизма. Однако добавление нового виртуального метода требует модификации всей иерархии классов. Также данный механизм напрямую не поддерживает множественного полиморфизма (мультиметодов), для реализации которых ОО решением является использование диспетчеризации, не способствующей безболезненному расширению классов. Зачастую для гибкой реализации мультиметодов в ОО языках используют дополнительный код, который не связан с вычислениями, определяемыми решаемой задачей, и предназначен только для организации дополнительных взаимодействий [11, 12].

В случае ПП подхода реализация отношений между обобщением ($Generalization$) и его специализациями осуществляется иным способом (рис. 1b). Процедурно-параметрический полиморфизм обеспечивает формирование альтернатив ($specialization_1$–$specialization_n$), используя для этого основы специализаций, которые являются независимыми абстрактными типами данных ($foundation_1$–$foundation_n$). Добавление альтернативных специализаций к обобщающему их типу данных может осуществляться в произвольные моменты времени и в различных единицах компиляции. Формируемые при этом композиции являются специализациями обобщения и принадлежат к тому же типу, что и обобщающий тип ($Generalization$). Основа специализации трактуется как подтип обобщения. Само обобщение также является одной из специализаций, не содержащей основу ($specialization_0$), но имеющей свои внутренние данные. Механизм идентификации специализаций локализован внутри каждого обобщения с использованием внутренней индексации (параметрических индексов).

Обработка сформированных альтернатив осуществляется с использованием внешних

независимых специальных функций — обработчиков специализаций. Они расширяют обобщающую их функцию. Обработчики специализаций, как и сами специализации, могут добавляться независимо друг от друга. Механизм идентификации специализаций используется для автоматического выбора обработчиков через параметрические таблицы, каждая их которых связана со своим набором обобщающих функций. Добавление новых обобщающих функций и их обработчиков может осуществляться в любой момент времени независимо от уже существующих данных и функций, что характерно для добавления любых новых функций при процедурном подходе. Эти функции могут содержать произвольное число обобщающих аргументов, обеспечивая тем самым безболезненное расширение мультиметодов [7].

В предлагаемом расширении языка С основами специализаций могут быть разнообразные типы данных 2. Их можно сформировать с использованием базовых типов данных, а также на основе именованных и неименованных структур. Помимо этого специализации могут формироваться на основе указателей на различные типы данных. Для идентификации подключаемых основ специализаций используются признаки, задаваемые идентификаторами. Они являются уникальными внутри каждого из формируемых обобщений. Также в качестве признака может выступать имя типа используемой основы специализации, что может трактоваться как применение в качестве признака имени типа. Подробное описание реализованных ПП расширений языка программирования С представлено в [10].

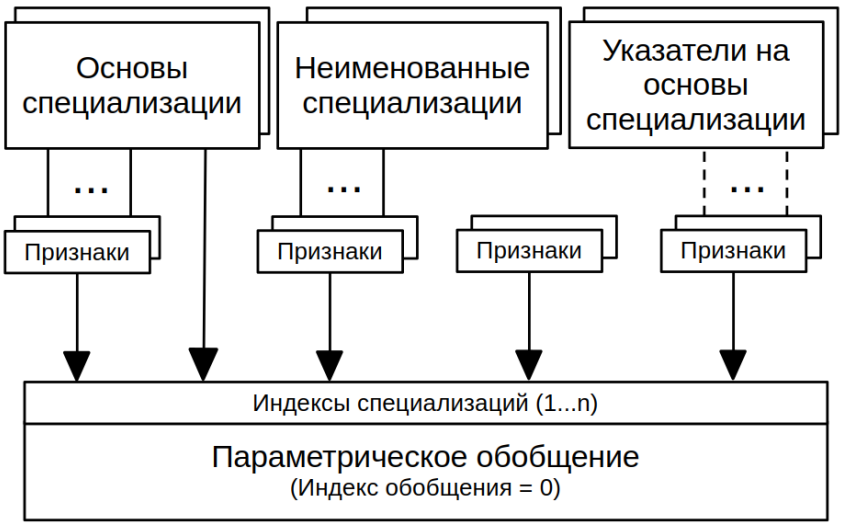

Рис. 2. Варианты процедурно-параметрических специализаций в расширении языка С

Примеры, демонстрирующие различные варианты прямого эволюционного расширения как для ОО, так и для ПП подходов приведены в каталоге `evolution` репозитория на Гитхаб [17].

4. Формирование оберток вокруг потоковых функций

Несмотря на появление новых перспективных языков, применение языка С в параллельном программировании остается популярным, что связано с его эффективным отображением на современные архитектуры. Использование языка актуально для встроенных систем, а также для приложений с критическим временем реакции на внешние события. Предлагаемые процедурно-параметрические расширения, на наш взгляд, позволяют повысить надежность и гибкость при создании многопоточных программ за счет создания соответствующих оберток над ненадежными конструкциями, а также за счет минимизации изменений в уже написанном коде.

Использование указателей на пустой тип `void` ведет к разыменованию типов и потере контроля над ними со стороны компилятора. Вместе с тем этот прием широко распространен при программировании на языке C и используется в различных библиотеках. Например, библиотека POSIX Threads [8] использует такой подход в функциях создания потока `pthread_create` и ожидания его завершения `pthread_join`:

```
int pthread_create( pthread_t *thread,
                    const pthread_attr_t *attr,
                    void *(*start_routine) (void *),
                    void *arg
);

int pthread_join( pthread_t thread, void **retval );
```

В связи с этим при вызове таких функций часто приходится осуществлять ненадежные преобразования типов данных.

4.1. Формирование объектно-ориентированной обертки

Преодоление этих недостатков в случае ОО программирования осуществляется за счет формирования абстрактной обертки, которая позволяет избавиться от прямого применения ненадежных конструкций. Пример простой обертки, реализованной в виде соответствующего класса [13], представлен ниже:

```
class Thread {
    public:
    virtual ~Thread () {}
    virtual void run () = 0;
    int start () {
        return pthread_create(
        &_ThreadId, nullptr,
        Thread::thread_func, this );
    }
    int wait () {
        return pthread_join( _ThreadId, NULL );
    }
    protected:
    pthread_t _ThreadId;
    static void* thread_func(void* d) {
        (static_cast <Thread*>(d))->run();
        return nullptr;
    }
};
```

На основе этой обертки можно создавать потоки, которые решают разнообразные задачи. В качестве примера рассмотрим класс, обеспечивающий вычисление периметра прямоугольника. Он определяет функционал соответствующего потока за счет наследования от базового класса `Thread`:

```
class ThreadRect : public Thread {
public:
    ThreadRect (int xx, int yy): x{xx}, y{yy} {}
    virtual void run () {
        p = double(2*(x+y));
    }
    void print(const char* str) {
        std::cout << str << ": Perimeter = " << p << "\n";
    }
protected:
    int x;
    int y;
    double p;
};
```

Использование этого и других подобных классов позволяет формировать и объединять множество потоков, избегая прямого использования функций библиотеки POSIX Threads:

```
...
ThreadRect thread1{3, 5};
ThreadRect thread2{13, 15};
thread1.start();  thread2.start();
thread1.wait();   thread2.wait();
thread1.print("Thread␣1");
thread2.print("Thread␣2");
...
```

Подобным образом в языке C++ реализована и более сложная обертка, определяющая стандартную библиотеку `thread` [14].

4.2. Формирование процедурно-параметрической обертки

Аналогичную обертку, подменяющую указатели на `void` на обобщения, обеспечивающие контроль над типами данных за счет специализаций, можно сформировать и при использовании ПП программирования. Одним из эквивалентных вариантов является создание обобщенной структуры, в которой специализации могут использоваться для передачи аргументов в формируемый поток и возврата из него результата.

```
typedef struct ThreadData {pthread_t threadId;}<> ThreadData;
```

Для работы с этой структурой можно создать абстрактную обобщенную функцию `RunThread`, не имеющую тела. На ее основе будут создаваться обработчики специализаций, формирующие требуемые потоковые функции. Данная функция в качестве обобщенного аргумента получает структуру с входными параметрами, передаваемые в поток, а также возвращаемыми из него:

```
void RunThread <ThreadData* d>() = 0;
```

Промежуточная потоковая функция `ThreadFunc`, скрытая от программиста внутри обертки, будет запускать различные обработчики специализаций этой обобщенной функции, формируемые программистом:

```
void* ThreadFunc(void* d) {
    RunThread<(Thread*)d>();
    return NULL;
}
```

Данная функция используется в качестве аргумента функции запуска потока, которая размещается внутри обертки `StartThread`, получающей данные, используемые для вычислений и возврата полученного результата:

```
int StartThread(ThreadData* td) {
    return pthread_create(td->threadId, NULL, ThreadFunc, td);
}
```

Слияние, обеспечивающее ожидание и завершение запущенных потоков, как и другие аналогичные функции библиотеки, тоже можно реализовать в соответствующей обертке:

```
int WaitThread(ThreadData* td) {
    return pthread_join(td->threadId, NULL);
}
```

Подобную обертку, используя процедурно-параметрическую парадигму, можно создать и над библиотекой `threads.h`, появившейся в стандарте языка C11 [15].

Рассмотренные оберточные функции можно применять для формирования различных потоков, рассматриваемых в качестве специализаций обобщения `RunThread`. В частности, использование прямоугольника и определение данных, необходимых для вычисления его периметра, будут выглядеть следующим образом:

```
typedef struct Rectangle{
    int x, y; // Sides of a rectangle
} Rectangle;

typedef struct RectPerimeter {
    Rectangle r;     // Rectangle
    double p;        // Perimeter as result
} RectPerimeter;

// Formation of specialization for a generalized thread
ThreadData + <RectPerimeter;>;
```

Обработчик специализаци, вычисляющий периметр, может быть реализован следующим образом:

```
void RunThread<ThreadData.RectPerimeter *rp>() {
    rp->@p = (double)((rp->@r.x+rp->@r.y)*2);
}
```

Он получает в качестве параметра сформированную специализированную структуру, возвращая в поле периметра полученный результат. Функция, осуществляющая непосредственный вывод значения периметра, выделяет из нее данные, размещенные в структуре `RectPerimeter`:

```
void PrintRectPerimeter(RectPerimeter* rp, const char* str) {
    printf("Perimeter␣of␣%s␣=␣%f\n", str, rp->p);
}
```

Главная функция, запускающая несколько потоков, скрывает особенности библиотеки POSIX Thread аналогично тому, как это делается и в ОО программе:

```
...
struct ThreadData.RectPerimeter thread1;
thread1.threadId = 0;
thread1.@r.x = 3;
thread1.@r.y = 5;
struct ThreadData.RectPerimeter thread2;
thread2.threadId = 1;
thread2.@r.x = 7;
thread2.@r.y = 4;

StartThread((ThreadData*)&thread1);
StartThread((ThreadData*)&thread2);

WaitThread((ThreadData*)&thread1);
WaitThread((ThreadData*)&thread2);

PrintRectPerimeter(&(thread1.@), "Thread␣1");
PrintRectPerimeter(&(thread2.@), "Thread␣2");
...
```

После формирования данных, передаваемых в потоки, производится их запуск в функциях (`StartThread`). Ожидание завершения потоков происходит на функциях `WaitThread`, а затем осуществляется вывод периметров. Символ «`@`» отделяет данные специализации от основной структуры, описывающей поток.

5. Эволюционное расширение многопоточных ПП программ

Использование ПП парадигмы позволяет добавлять в многопоточную программу, как новые альтернативные данные, так и новые функции, осуществляющие их обработку. В этом плане использование методов эволюционного расширения практически ничем не отличаются от их применения в последовательных ПП программах. Однако в случае многопоточности допускается параллельная обработка с сохранением гибкости по расширению ранее написанного кода.

5.1. Добавление новых альтернативных данных

Рассмотрим особенности расширения программ на следующем простом примере. Пусть имеется массив хранящий различные геометрические фигуры, которые изначально могут быть прямоугольниками и треугольниками. Необходимо вычислить периметр каждой из фигур после чего подсчитать общую сумму. Для вычисления периметра каждой фигуры предполагается использование отдельного потока.

Прямое применение традиционного процедурного подхода в языке C обычно ведет к представлению фигур либо с использованием объединений, либо применяется разыменование типов за счет использование указателей на пустой тип (`void*`). В первом случае ключ, определяющий тип фигуры инструментально не связан с хранимой фигурой. Во втором варианте приходится использовать разыменование типов. В обоих случаях имеются проблемы, связанные с обеспечением надежности программы. В случае объединений описание обобщенной фигуры, задающей прямоугольник и треугольник, можно представить следующим образом:

```
typedef struct Rectangle {
    int x, y;
} Rectangle;

typedef struct Triangle {
    int a, b, c;
} Triangle;

typedef enum key {RECTANGLE, TRIANGLE} key;
typedef struct Figure {
    key k;
    union {
        Rectangle r;
        Triangle t;
    };
} Figure;
```

Вычисление периметра в этом случае обычно реализуется с использованием единой централизованной функции, в которой по ключу осуществляется выбор хранимой фигуры:

```
double FigurePerimeter(Figure *s) {
    switch(s->k) {
        case RECTANGLE:
            return RectanglePerimeter(&(s->r));
            break;
        case TRIANGLE:
            return TrianglePerimeter(&(s->t));
            break;
        default:
            return 0.0;
    }
}
```

Если в разрабатываемую процедурную программу потребуется добавить еще одну или несколько разновидностей фигур, например круг, то ранее написанный код придется изменить. Это затрудняет эволюционную разработку и последующее сопровождение. Проблемы, связанные с изменением обработчиков, особенно проявляются при большом числе используемых в них альтернативных данных. Следует также отметить, что подобное решение в случае ООП позволяет без труда добавлять новые альтернативы за счет производных классов и обеспечивать для них формирование соответствующих методов, используя ОО полиморфизм. Но при этом добавление новых методов ведет к изменению как базового, так и производных классов.

Процедурно-параметрическое расширение языка C аналогично по возможностям ОО подходу. Оно обеспечивает гибкое добавление как новых данных, так и новых обработчиков специализаций, так как альтернативы, являющиеся специализациями уже существующего общего типа формируются независимо друг от друга. Это же касается и обработчи-

ков специализаций, каждый из которых независимо расширяет обобщенную функцию. При использовании ранее приведенных описаний прямоугольника и треугольника обобщенная фигура будет формироваться на основе структуры, в которой отсутствуют специализации:

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

Далее независимо друг от друга формируются специализации этого обобщения, определяющие фигуру–прямоугольник и фигуру–треугольник:

```
Figure + < rect: Rectangle; >;
Figure + < trian: Triangle; >;
```

Каждая из специализаций может быть описана в своем заголовочном файле, что при добавлении следующих фигур, обеспечивает их независимость и неизменность ранее написанного кода. Например, добавление круга будет связано только с его описанием и зависимостью от основы и обобщенной фигуры:

```
typedef struct Circle {
    int r;
} Circle;

Figure + < circ: Circle; >;
```

Аналогичным образом в независимых единицах компиляции осуществляется формирование обобщенной функции вычисления периметра и ее различных обработчиков, что позволяет постепенно и эволюционно наращивать функциональность программы:

```
double FigurePerimeter<Figure *f>() = 0;

void FigurePerimeter<Figure.rect *f>() {
    RectanglePerimeter(&(f->@));
}

void FigurePerimeter<Figure.trian *f>() {
    TrianglePerimeter(&(f->@));
}
```

Для вычисления периметра в многопоточном режиме необходимо сформировать структуру, позволяющую передавать в потоковую функцию аргументы и возвращать результат. Эта структура используется в качестве специализации потоковой функции, вычисляющей периметр обобщенной фигуры:

```
typedef struct FigurePerimeterData {
    int iThread;  // Thread id
    Figure *f;    // Pointer to figure
    double p;     // Figure perimeter as result
} FigurePerimeterData;

struct ThreadData + <FigurePerimeterData;>;

void RunThread<ThreadData.FigurePerimeterData *figurePerimeter>() {
    figurePerimeter->@p = FigurePerimeter<figurePerimeter->@f>();
}
```

Для хранения различных геометрических фигур можно создать простейший контейнер, например, на основе одномерного массива:

```
enum {max_len = 100};

typedef struct Container {
    int len;
    struct Figure *cont[max_len];
} Container;
```

Обработка всего массива может быть организована запуском потоков для каждого из его элементов:

```
double CalcFigurePerimetersInContainer(Container *c) {
    struct ThreadData.FigurePerimeterData perimeterArray[c->len];
    struct ThreadData.FigurePerimeterData *pElement;
    for(int i = 0; i < c->len; ++i) {
        // Specialization of the memory area
        // allocated for the thread data
        init_spec(ThreadData.FigurePerimeterData, (perimeterArray+i));
        pElement = (perimeterArray+i);
        pElement->@iThread = i;
        pElement->@f = c->cont[i];
        pElement->@p = 0.0;
        StartThread((ThreadData*)(perimeterArray+i));
    }

    double perimeterSum = 0.0;
    for(int i = 0; i < c->len; ++i) {
        pElement = (perimeterArray+i);
        WaitThread((ThreadData*)(perimeterArray+i));
        perimeterSum += pElement->@p;
        printf("The␣thread␣%d:␣perimeter␣=␣%lf\n",
        pElement->@iThread, pElement->@p);
    }
    printf("Common␣Figures␣Thread␣Perimenter␣=␣%lf\n", perimeterSum);
    return perimeterSum;
}
```

В представленном примере вспомогательный массив переменных, передаваемых в потоковую функцию, формируется на стеке, что позволяет избежать динамического выделения памяти с использованием специальной функции `create_spec`. Однако каждая из формируемых специализаций, размещаемая в массиве `perimeterArray`, имеет свой признак, который не устанавливается автоматически. Поэтому для формирования признаков специализаций используется специальная функция инициализации `init_spec`, аргументами которой являются тип специализации и адрес области памяти, выделяемой под инициализируемую специализацию. Описание дополнительных функций, используемых для поддержки процедурно-параметрического программирования в расширении языка C, приведено в [?].

5.2. Добавление новых функций

При добавлении в ПП программу новой функциональности, например, для вычисления площади геометрических фигур в уже сформированный код также не нужно вносить дополнительных изменений создаются только новые функции, обеспечивающие вычисление площади по аналогии с вычислением периметра. Для формирования новых потоков создается обработчик специализации, содержащий данные, необходимые для вычисления площади. В этой ситуации процедурно-параметрический подход является более гибким по сравнению с объектно-ориентированным, при котором добавление нового метода ведет к изменению интерфейсов ранее написанных классов, а также включением в них новых реализаций, обеспечивающих вычисление площадей фигур уже описанных в программе.

6. Многопоточность и множественный полиморфизм

Аналогичным образом ПП подход позволяет расширять функции от нескольких полиморфных аргументов, а также добавлять новые мультиметоды в программу, формировать новые специализации, которые осуществляют обработку этих мультиметодов [7]. Использование многопоточного программирования в этом случае также не мешает гибкому расширению функциональности программ.

В качестве примера параллельного выполнения мультиметодов можно рассмотреть добавление функции осуществляющей вывод отношений между различными парами геометрических фигур, размещенных в контейнере. Для запуска каждого потока используется функция `TreadContainerMultimethodOut`. Она во вложенном цикле формирует данные для каждой из пар массива фигур. После чего запускает стартовую функцию потока `StartThread`.

```
// Output of pairs of figures from container
// to the specified thread using a multimethod
void TreadContainerMultimethodOut(Container *c, FILE* ofst) {
    struct ThreadData.MultimethodData mmArray[c->len][c->len];
    struct ThreadData.MultimethodData *pElement;
    int threadId = 0;
    for(int i = 0; i < c->len; i++) {
        for(int j = 0; j < c->len; j++) {
            init_spec(ThreadData.MultimethodData, &mmArray[i][j]);
            pElement = &mmArray[i][j];
            pElement->@iThread = threadId;
            pElement->@c = c;
            pElement->@figureIndex1 = i;
            pElement->@figureIndex2 = j;
            pElement->@ofst = ofst;
            pElement->@p = 0.0;
            StartThread((ThreadData*)(&mmArray[i][j]));
            threadId++;
        }
    }
    // Assembling the results returned by threads
    for(int i = 0; i < c->len; i++) {
        for(int j = 0; j < c->len; j++) {
            pElement = (&mmArray[i][j]);
            WaitThread((ThreadData*)&mmArray[i][j]);
        }
    }
}
```

Основа специализации используется для передачи в потоки информации об индексах каждой из фигур формируемой пары:

```
// The structure of data transmitted to a thread processing
// a one of combination of multimethod
typedef struct MultimethodData {
    int iThread; // Thread ID
    Container* c;
    // The figure container
    int figureIndex1; // The index of the first figure
    int figureIndex2; // The index of the second figure
    FILE* ofst;
} MultimethodData;
```

Специализация, сформированная с использованием этой основы, обеспечивает использование обобщенного обработчика, запускающего поток:

```
// Specialization for the thread used in the multimethod
ThreadData + <MultimethodData;>;
```

Переопределяемая поточная функция `RunThread`, использует сформированную специализацию и вызывает функцию `MultimethodLauncher`, осуществляющую непосредственный вызов одного из обработчиков мультиметодов, доступный через обобщающую функцию `Multimethod`.

```
// Implementation of a specialization handler that runs in a thread.
void RunThread<ThreadData.MultimethodData *mm>() {
    MultimethodLauncher(mm->@iThread,
                        mm->@c,mm->@figureIndex1,
                        mm->@figureIndex2,mm->@ofst);
}

// A function that starts the handler for figures combination
void MultimethodLauncher(int iThread,
Container *c,
int i, int j, FILE* ofst) {
    sleep(rand()%3); // The delay for generating a random execution time
    printf("%d [%d,%d]: ", iThread, i, j);
    Multimethod<c->cont[i], c->cont[j]>(ofst);
}
```

Сам мультиметод, формируется на основе абстрактной обобщенной функции и множества раздельных обработчиков специализаций, определяющих все необходимые отношения между геометрическими фигурами, реализованными в программе.

```
// A generalizing function defining an abstract multimethod
void Multimethod<Figure* f1, Figure* f2>(FILE* ofst) = 0;

// Specialization handler for two rectangles
void Multimethod<Figure.rect* r1, Figure.rect* r2>(FILE* ofst) {
    fprintf(ofst, "Two␣Rectangles␣Combination\n");
}

// Specialization handler for rectangle and triangle
void Multimethod<Figure.rect* r1, Figure.trian* t2>(FILE* ofst) {
    fprintf(ofst, "Rectangle␣-␣Triangle␣Combination\n");
}

// Specialization handler for triangle and rectangle
void Multimethod<Figure.trian* t1, Figure.rect* r2>(FILE* ofst) {
    fprintf(ofst, "Triangle␣-␣Rectangle␣Combination\n");
}

// Specialization handler for two triangles
void Multimethod<Figure.trian* t1, Figure.trian* t2>(FILE* ofst) {
    fprintf(ofst, "Two␣Triangles␣Combination\n");
}
```

Добавление новой фигуры и ее обработчиков специализаций, обеспечивающих взаимодействие с другими фигурами осуществляется без изменения ранее написанного кода. Аналогичным образом возможно и добавление новых мультиметодов, реализующих другие алгоритмы взаимодействия между фигурами.

7. Заключение

Предлагаемое расширение языка программирования C конструкциями, обеспечивающими инструментальную поддержку процедурно-параметрической парадигмы программирования, позволяет разрабатывать гибкие и эволюционно расширяемые параллельные программы. При этом обеспечивается добавление как новых альтернативных данных, так и функций без изменения ранее написанного кода. Эволюционное расширение возможно и в случае множественного полиморфизма. Реализация проекта осуществляется путем интеграции в компилятор clang [9], что позволяет изменять как синтаксический анализатор, так и абстрактное синтаксическое дерево, адаптируя их под новые конструкции. Проект реализуется под открытой лицензией и размещен на Гитхаб [16]. Примеры, демонстрирующие возможности процедурно-параметрической парадигмы программирования, включающие и использование многопоточности, также размещены на гитхаб [17].

Литература

1. Gregoire M. Professional C++. John Wiley & Sons, 2018. 1122 p.
2. Sciore E. Java Program Design. Apress Media, 2019. 1122 p.
3. Freeman A. Pro Go: The Complete Guide to Programming Reliable and Efficient Software Using Golang. Apress, 2022. 1105 p.
4. Blandy J., Orendorff J., Tindall L.F. Programming Rust. O'Reilly Media, 2021. 735 p.
5. Элджер Дж. C++: библиотека программиста. СПб.: ЗАО «Издательство Питер», 1999. 320 с.

6. Легалов А.И. Процедурно-параметрическая парадигма программирования. Возможна ли альтернатива объектно-ориентированному стилю? Красноярск: 2000. Деп. рук. № 622-В00 Деп. в ВИНИТИ 13.03.2000. 43 с.
URL: http://www.softcraft.ru/ppp/pppfirst/. (дата обращения: 31.01.2025).

7. Легалов А.И., Косов П.В. Расширение языка С для поддержки процедурно-параметрического полиморфизма // Моделирование и анализ информационных систем. 2023. Т. 30, № 1. С. 40–62.
DOI: 10.18255/1818-1015-2023-1-40-62.

8. Butenhof D.R. Programming with POSIX Threads. Addison-Wesley, 1997. 398 p.

9. Clang: a C language family frontend for LLVM. URL: https://clang.llvm.org/ (дата обращения: 31.01.2025).

10. Легалов А.И., Косов П.В. Процедурно-параметрическое расширение языка программирования С. Синтаксис и семантика. URL: http://softcraft.ru/ppp/ppc/ (дата обращения: 31.01.2025).

11. Мейерс С. Наиболее эффективное использование C++. 35 новых рекомендаций по улучшению ваших программ и проектов. М.: ДМК Пресс, 2000. 304 с.

12. Легалов А.И. ООП, мультиметоды и пирамидальная эволюция // Открытые системы. 2002. № 3. С. 41–45.

13. POSIX Threads. URL: https://ru.wikipedia.org/wiki/POSIX_Threads (дата обращения: 31.01.2025).

14. Гримм Р. Параллельное программирование на современном языке C++. М.: ДМК Пресс, 2022. 616 с.

15. C. Concurrency support library. URL: https://en.cppreference.com/w/c/thread (дата обращения: 31.01.2025).

16. Размещение проекта с процедурно-параметрической версией языка программирования С на Гитхаб. URL: https://github.com/kpdev/llvm-project/tree/pp-extension-v2 (дата обращения: 31.01.2025).

17. Примеры на Гитхаб, написанные с использованием процедурно-параметрической версии языка С: URL: https://github.com/kreofil/evo-situations (дата обращения: 31.01.2025).