ИНСТИТУТ ПРИКЛАДНОЙ МАТЕМАТИКИ
им. М.В. КЕЛДЫША
РОССИЙСКОЙ АКАДЕМИИ НАУК

А.М.Горелик

ОБЪЕКТНО-ОРИЕНТИРОВАННОЕ ПРОГРАММИРОВАНИЕ НА СОВРЕМЕННОМ ФОРТРАНЕ

Москва
2002

Работа выполнена при финансовой поддержке Российского фонда фундаментальных исследований (проект 00-01-00043).

Объектно-ориентированное программирование на современном Фортране

А.М.Горелик

АННОТАЦИЯ

В работе рассматриваются основные концепции объектно-ориентированного программирования (ООП): абстрактные типы данных, инкапсуляция, перегрузка, наследование, статический и динамический полиморфизм. Обсуждаются конкретные средства языка Фортран 90/95, которые поддерживают большинство из этих концепций.

This work was supported by Russian Foundation for Fundamental Investigations (Project 00-01-00043).

Object-oriented programming in modern Fortran

A.M.Gorelik

The preprint of the Keldysh Institute of Applied Mathematics,

Russian Academy of Sciences

ABSTRACT

This paper considers the basic object-oriented concepts: abstract data types, encapsulation, overloading, inheritance, static and dynamic dispatching. Besides, techniques for implementation the most of them in Fortran 90/95 program are discussed.

Содержание

1. Введение

Современные стандарты Фортрана (Фортран 90/95) [1-6], по сравнению с Фортраном 77, содержат много новых средств, которые позволяют использовать новые технологии программирования, включая средства объектно-ориентированного программирования (ООП).

Для небольших программ не очень существенно, какая методология программирования выбрана. При разработке больших программ применение современных технологий, в т.ч. ООП, дает существенный выигрыш.

В то же время программисты, использующие Фортран 77 и более ранние версии (которые были разработаны еще в 60-е и 70-е годы прошлого века), вынуждены использовать старые технологии.

Переход к новым технологиям с сохранением многолетнего вклада в разработку Фортран-программ – это требование времени. Переход на современный Фортран актуален еще и потому, что средства для использования параллельных компьютеров также ориентированы на Фортран 90/95. Поскольку Фортран 90/95 является расширением Фортрана 77, некоторые из новых концепций могут быть реализованы путем модификации старых программ.

Следует отметить, что при программировании на современном Фортране реже требуется использовать элементы ООП, чем для С-программ, так как Фортран содержит больше встроенных абстракций. Отметим также тот факт, что переход от Фортрана 77 (для которого накоплен огромный фонд прикладных программ) к Фортрану 90/95 гораздо проще, чем переход к C++ [7].

Эти (и не только эти) обстоятельства привели к осознанию того, что необходимо развивать средства поддержки ООП в современном Фортране.

Действующий стандарт Фортран 95 [1] (как и его предшественник Фортран 90 [2-6]) поддерживает значительную часть методологии объектно-ориентированного программирования (ООП), хотя и не обладает полным набором средств ООП.

В настоящее время на международном уровне ведется работа по дальнейшему развитию языка: разрабатывается проект будущего стандарта (рабочее название - Фортран 2000; завершение планируется в 2004г.). Этот проект предусматривает весьма существенные нововведения, одним из важных направлений является развитие средств, обеспечивающих полную поддержку всех концепций ООП.

В данной работе рассматриваются основные концепции ООП и конкретные средства языка Фортран 90/95, которые поддерживают те или иные элементы ООП.

2. Основные концепции объектно-ориентированного программирования

Объектно-ориентированное программирование (ООП) – это современная методология программирования.

ООП позволяет не только использовать встроенные абстракции (как в традиционном программировании), но и определять собственные абстракции и тем самым позволяет описать программу в терминах близких к прикладной области. В то же время ООП не препятствует использованию других технологий программирования (например, можно использовать структурное программирование).

Методология ООП ориентирована прежде всего для написания очень больших программ или набора программ, которые разрабатывают группы программистов. При использовании старых традиционных методов и старых языков программирования, такие программы или системы трудны для понимания, сопровождения и модификации. При разработке больших систем обычно использовались дополнительные инструментальные средства или такие методы, как наборы правил и соглашений, которые должны выполняться членами группы. Современные языки программирования предлагают новые методы.

Метод ООП не только облегчает разделение работы между программистами при проектировании больших программ, но и обеспечивает возможность многократного использования кодов. Кроме того, объектно-ориентированная программа более надежная, легче для понимания и модификации (включая расширения).

Имеются языки, специально ориентированные на ООП: Simula 67, Lisp, Smalltalk, Eiffel, Java и др. Если говорить о более универсальных языках, то в языке Ada 83 была лишь частичная поддержка ООП, в C - таких средств не было. Более поздние версии Ada 95 и C++ реализуют полный механизм ООП. Языковая поддержка ООП для C++ и Ада 95 обсуждается в книге [8]. Язык Фортран в книге отнесен к языкам с низким уровнем абстракции. Это справедливо только в отношении “старого” Фортрана.

Что касается современного Фортрана, отметим, что в описании стандарта явно не указывается, какие средства обеспечивают поддержку тех или иных элементов ООП (это и не требуется в официальном документе). Однако, как показали проведенные исследования, такие средства имеются, и эти средства весьма полезны не только в связи с ООП. Проблемы ООП для Фортрана 90 обсуждаются в работах [9-10].

Ниже приводится перечень основных концепций ООП.

- Расширяемость типов, т.е. возможность добавлять определяемые пользователем типы.
- Расширяемость операций, т.е. возможность описывать новые операции.
- Возможность статического контроля соответствия типов и соответствия формальных и фактических аргументов при вызове процедур.
- Механизм инкапсуляции для реализации абстрактных типов данных.
- Наследование, т.е. возможность для порожденного объекта наследовать свойства (описание типа и операций) порождающего (родительского) объекта.
- Статический полиморфизм, т.е. возможность для программиста использовать какой-либо объект более чем одним способом, причем конкретизация способа использования выполняется на этапе компиляции.
- Динамический полиморфизм (динамическое связывание), т.е. конкретизация способа использования объекта реализуется на этапе выполнения.

В следующих разделах рассматривается, как эти концепции реализуются средствами Фортрана 90/95.

3. Статический контроль соответствия типов и соответствия аргументов

Для ООП важным является наличие в языке возможности статического контроля соответствия типов и соответствия формальных и фактических аргументов при вызове процедур.

Для языка Фортран традиционно был характерен принцип умолчания, т.е. автоматическое приписывание объектам типа, если тип объекта не указан явно. В Фортран 90/95 введен оператор IMPLICIT NONE. При наличии такого оператора типы объектов в данной программной единице должны быть объявлены явно. Явное объявление объектов обеспечивает возможность контроля типов, что, очевидно, полезно не только для ООП, но и для повышения надежности создаваемых программ. IMPLICIT NONE является более гибким инструментом, чем просто обязательность описаний, что имеет место в других языках.

Для обеспечения статического контроля важной чертой является возможность проверки соответствия типов формальных и фактических аргументов. Для этих целей в современных стандартах имеется механизм, который обеспечивает явный интерфейс при вызове процедур.

В Фортране 77 интерфейс с вызываемой внешней процедурой был только неявным, т.е. при обращении к процедуре отсутствует или является неполной

информация о формальных аргументах вызываемой процедуры. Поскольку программные единицы, содержащие внешние процедуры, обычно компилируются раздельно и независимо, неявный интерфейс, очевидно, не позволяет контролировать правильность обращений к процедурам на этапе компиляции.

В Фортран 90/95 для внешних процедур может быть задан явный интерфейс, который обеспечивается с помощью интерфейсного блока; интерфейсный блок содержит заголовок процедуры и спецификации формальных аргументов вызываемой процедуры и результатов функции. Тем самым явный интерфейс позволяет компилятору организовать проверку правильности обращения к процедуре (в частности, проверить соответствие типов аргументов) уже на этапе компиляции.

В ряде случаев интерфейсный блок является обязательным, так как позволяет правильно сгенерировать вызов процедуры.

Некоторые ситуации использования интерфейсного блока и примеры будут приведены ниже (см. 5.2., 5.3., 6.2., 7.4. и 7.5.).

4. Расширяемость типов

Для ООП в языке необходимы средства, позволяющие добавлять определяемые пользователем типы данных. Такие типы должны быть столь же удобны в использовании, как и встроенные. В Фортране 90/95 для этих целей имеются производные (derived) типы.

Производные типы (структуры) представляют собой совокупность компонент. Компоненты производного типа могут иметь встроенный или ранее описанный производный тип; компонентами могут быть скаляры, массивы и указатели. Компонент производного типа может иметь такой же тип как описываемый, если он является указателем (т.е. имеет атрибут POINTER). Это полезно для создания связанных списков.

Скалярный объект производного типа представляет собой структуру. Допускаются массивы объектов производного типа.

Для ссылки на отдельные компоненты производного типа используется символ %. Компоненты объекта производного типа могут использоваться в обычных выражениях и операторах.

Для объектов одинакового производного типа определена операция присваивания; кроме того, такие объекты могут использоваться в операторах ввода/вывода, в качестве аргументов процедур и как результат функции.

Для данных производного типа не существует встроенных операций, но пользователь может определить операции и присваивания (см. 5.). Производные типы могут быть описаны в модуле (см. 6.2.).

Примеры

```
! Описание типа PERSON
TYPE PERSON
  INTEGER AGE
  CHARACTER (LEN = 10) NAME
END TYPE PERSON

! Объявление скаляров и массива описанного типа
TYPE (PERSON):: STUDENT, TEACHER, MEMBER (10)
! Операция над компонентами производного типа
N = TEACHER%AGE - STUDENT%AGE
! Компонент элемента массива производного типа
MEMBER (1) % AGE = 50
! Присваивание объекту производного типа
STUDENT = (20, '    IVANOV')

! Пример связанного списка
TYPE LINK
  REAL VALUE
  TYPE (LINK), POINTER :: PREVIOUS
  TYPE (LINK), POINTER :: NEXT
END TYPE LINK
```

(Все примеры в работе приводятся в свободном формате и используются заглавные буквы; фиксированный формат и строчные буквы также допустимы.).

5. Определяемые операции и присваивания

5.1. Общие сведения

В языке имеется аппарат, позволяющий программисту распространить для производных типов встроенные операции (перегрузка операций, см. 7.3.) или описать новые операции для данных встроенных и производных типов. Кроме того, можно описать новые операторы присваивания. Эти возможности позволяют программисту определить абстрактные типы данных (см. 6.), что является одним из элементов объектно-ориентированного программирования.

Ниже рассматривается как эти средства реализуются в языке.

5.2. *Определяемые операции*

В программе можно описать и затем использовать в выражениях определяемые унарные и бинарные операции. Они описываются с помощью функций; если такая функция имеет один формальный аргумент, она описывает унарную операцию, если два – бинарную; при этом аргументы должны быть входными, т.е. INTENT(IN). Такая функция может быть помещена в модуль (см. 6.2.).

Знак определяемой операции либо совпадает со знаком одной из встроенных операций, либо представляет собой последовательность букв, заключенную в точки.

Примеры знаков определяемых операций:

```
+     .ADD.     .INVERSE.     .EQ.     .PLUS.
```

Если знак определяемой операции совпадает со знаком встроенной, то типы аргументов должны отличаться от типов соответствующей встроенной операции.

В качестве примера определяемой операции рассмотрим описание операции сложения в интервальной арифметике. Напомним, что объект в интервальной арифметике, называемый интервалом, может быть представлен двумя вещественными числами – границами интервала. Над объектами-интервалами определены операции, подобные обычным арифметическим операциям. Операции над интервалами описываются через арифметические операции над границами.

Пример

```
! Функция для описания операции сложения
FUNCTION ADD (A, B) OPERATOR (+)
    INTENT (IN) A, B
!   Описание типа INTERVAL
    TYPE INTERVAL
       REAL LOWER, UPPER
    END TYPE
    TYPE (INTERVAL) ADD, A, B
     ADD % LOWER = A % LOWER + B % LOWER
     ADD % UPPER = A % UPPER + B % UPPER
END FUNCTION ADD
```

Если в программной единице в выражении используется операция, определяемая программистом во внешней функции, то должен быть включен интерфейсный блок. Такая функция может быть помещена в модуль (см. 6.2.). Ниже приведен фрагмент программы, в котором используется определенная в предыдущем примере операция сложения.

```
! Фрагмент вызывающей программной единицы
!   Интерфейсный блок со спецификацией функции,
!    реализующей определяемую операцию
INTERFACE OPERATOR (+)          ! Знак операции “+”
  FUNCTION ADD (A, B) OPERATOR (+)
    INTENT (IN) A, B
    TYPE (INTERVAL) ADD, A, B
  END FUNCTION ADD
END INTERFACE

!   Использование определяемой операции
TYPE (INTERVAL) Z, X, Y
Z = X + Y
```

К функциям, описывающим определяемую операцию, можно обращаться либо обычным образом с помощью имени функции (в приведенном примере - ADD), либо используя знак операции (в примере – “+”).

5.3. *Определяемое присваивание*

Определяемый оператор присваивания описывается с помощью подпрограммы. Такая подпрограмма имеет два аргумента, соответствующие левой и правой частям присваивания; первый аргумент является переменной с атрибутом OUT или INOUT, второй – должен иметь атрибут IN. В программной единице, использующей определяемое присваивание, описанное во внешней процедуре, должен быть задан явный интерфейс. Такая процедура может быть помещена в модуль (см. 6.2.).

Определяемый оператор присваивания имеет ту же форму, что и встроенный оператор присваивания, и отличается от него тем, что для характеристик его левой и правой частей не выполняются правила, установленные для встроенного оператора присваивания.

Распознавание того, что оператор вида *v=e* является определяемым оператором присваивания, а не встроенным, осуществляется следующим образом. Такой оператор является определяемым, если имеется интерфейсный блок подпрограммы присваивания с формальными аргументами *x* и *y*, типы которых соответствуют типам *v* и *e* - левой и правой части оператора присваивания.

Пример описания и использования определяемого присваивания.

```
! Описание присваивания значения логического типа
!          переменной символьного типа
      SUBROUTINE LOG_TO_SYM (SYMB_VAR, LOG_EXPR)
         CHARACTER, INTENT (OUT) :: SYMB_VAR
         LOGICAL, INTENT (IN)    :: LOG_EXPR
         SYMB_VAR = 'F'
         IF (LOG_EXPR == .TRUE.) SYMB_VAR = 'T'
      END SUBROUTINE LOG_TO_SYM

! Фрагмент программной единицы, где используется
!          определяемое присваивание
      PROGRAM EXAMPLE
        CHARACTER A
        LOGICAL B

!     Интерфейсный блок со спецификацией подпрограммы,
!          описывающей определяемое присваивание
      INTERFACE  ASSIGNMENT (=)
         SUBROUTINE LOG_TO_SYM (SYMB_VAR, LOG_EXPR)
            CHARACTER, INTENT (OUT) :: SYMB_VAR
            LOGICAL, INTENT (IN)    :: LOG_EXPR
          END SUBROUTINE LOG_TO_SYM
      END INTERFACE

!     Оператор определяемого присваивания
      A = B
```

6. Инкапсуляция и абстракция данных

6.1. *Общие сведения*

Фундаментальной идеей ООП является инкапсуляция данных и соответствующих операций. Инкапсуляция в Фортране 90/95 реализуется с помощью

модулей (см. 6.2.) и различных механизмов динамического размещения объектов (см. 6.3.).

Инкапсуляция объектов означает возможность скрывать внутренние детали реализации при предоставлении открытого интерфейса к некоторому объекту. Объекты, обладающие таким свойством, часто называют абстрактными. Например, для абстрактных данных пользователю достаточно знать только тип и набор операций над данными такого типа, но он может не знать, как описан тип и как реализованы эти операции. Если поместить в модуль описания определяемых пользователем типов и операций над данными такого типа, это может обеспечить их инкапсуляцию.

Здесь можно провести аналогию, которая традиционно существует для встроенных операций. Например, используя операцию .AND. для логических операндов, программист не интересуется, как представлен логический тип в машине, и как выполняются операции.

Пример, часто приводимый в литературе, - стек. Тип “стек” может быть реализован (описан), например, как массив фиксированной длины или как связанный список. Можно описать операции push (поместить в стек) и pop (извлечь из стека). Если описание стека изменено и, естественно, изменено описание операций, программист все равно может их использовать без необходимости изменять код этих операций.

Указанная черта повышает надежность и безопасность программы и, кроме того, делает программные единицы легче для использования, так как позволяет изменять внутренние детали, не влияя на пользователя.

6.2. *Модули*

Модуль – это новый вид программных единиц в Фортране. Наличие программных единиц-модулей, производных типов, определяемых пользователем операций и атрибутов PUBLIC и PRIVATE обеспечивают полную поддержку инкапсуляции и абстракции данных.

Внутри модуля могут быть описаны производные типы и процедуры, определяющие операции над объектами этого типа. Их реализация может быть скрыта внутри модуля, и их можно менять в случае необходимости. Программная единица, использующая модуль, может не знать внутреннее представление типа, и ей могут быть недоступны отдельные компоненты. В то же время сами типы и операции могут быть доступны программным единицам, которые используют модуль, и изменения структуры типов и описаний операций не влияют на

программные единицы, использующие этот модуль (пример см. ниже в данном разделе).

Модули могут содержать глобальные объекты, которые могут быть использованы другими программными единицами без явного объявления их внутри этих программных единиц.

Объекты модуля, которые используются в других программных единицах, должны иметь атрибут PUBLIC (явно описанный или по умолчанию). Те объекты модуля, которые предназначены для использования лишь внутри данного модуля, должны быть объявлены с атрибутом PRIVATE.

В программной единице, использующей объекты некоторого модуля, должен присутствовать оператор USE с указанием имени модуля.

Аппарат модулей позволяет создавать пакеты (модули) для различных приложений. Фрагмент подобного модуля для интервальной арифметики (см. 5.2.) воспроизведен ниже.

Пример

```
MODULE INTERVAL_ARITHMETIC
    TYPE INTERVAL
        PRIVATE
        REAL LOWER, UPPER
    END TYPE
    INTERFACE OPERATOR (+)
        MODULE PROCEDURE ADD
    END INTERFACE
    CONTAINS   ! Оператор отделяет модульные процедуры
!    Модульная процедура
      FUNCTION ADD (A, B) OPERATOR (+)
        TYPE (INTERVAL) ADD
        TYPE (INTERVAL), INTENT (IN) :: A, B
         ADD % LOWER = A % LOWER + B % LOWER
         ADD % UPPER = A % UPPER + B % UPPER
      END FUNCTION ADD
!    Модульные процедуры для описания других операций
        . . .
END MODULE INTERVAL_ARITHMETIC
```

Использование модуля интервальной арифметики показано в следующем примере.

```
PROGRAM INT_ARITH_EXAMPLE
        USE INTERVAL_ARITHNETIC
         TYPE (INTERVAL) X, Y, Z
          . . .
        Z = X + Y
         . . .
END PROGRAM
```

В приведенном примере главная программная единица может использовать тип INTERVAL и операции (сложение и другие описанные в модуле операции), но не имеет доступ к отдельным компонентам типа INTERVAL, так как описание этого типа содержит оператор PRIVATE.

В объектно-ориентированных языках программирования используется понятие “класс”. Так, в C++ класс содержит новый тип и операции, которые могут выполняться этим классом. В такой трактовке в Фортране 90/95 аналогом класса является модуль, содержащий одно описание производного типа и описание операций над данными этого типа (однако понятие модуля в языке значительно шире).

Для читателя, не знакомого с Фортраном 90/95, приведем некоторую дополнительную информацию о модулях. Модули полезны не только в связи с ООП. В частности, такие средства описания глобальных объектов являются более общими и гибкими, чем оператор COMMON, так как предоставляют пользователю возможность работать не только с общими данными, но также с другими общими объектами - типами данных, заготовленными описаниями процедур и определяемых операций и др. Способ работы с глобальными данными, обеспечиваемый модулями, более гибкий, чем с помощью операторов COMMON: глобальные данные при этом способе доступны по имени, а не по положению внутри общего блока. Кроме того, в отличие от описания общих блоков в операторах COMMON, глобальные переменные надо описывать только в одном месте.

Оператор USE позволяет ограничить доступ к общедоступным объектам и переименовать доступные объекты. Поясним на примерах.

Примеры операторов USE.

```
USE MOD1
USE MOD2 NEWA => A, NEWB => B
USE MOD3 ONLY: X, Y, NEWZ => Z
```

Операторы, приведенные в примере, означают, что в данной программной единице доступны следующие объекты:

1) все объекты с атрибутом PUBLIC из модуля MOD1 (локальные имена этих объектов совпадают с их именами в модуле MOD1);

2) все объекты с атрибутом PUBLIC из модуля MOD2, причем объекты с именами A и B получают локальные имена NEWA и NEWB (остальные объекты сохраняют те же имена, что в модуле MOD2);

3) только объекты с именами X, Y, Z (они должны иметь атрибут PUBLIC) из модуля MOD3; причем объект Z получает локальное имя NEWZ (остальные объекты, т.е. X и Y сохраняют свои имена).

Переименование объектов бывает удобно, а иногда и необходимо, если в программной единице, использующей объекты модуля, имеются свои локальные объекты, имена которых совпадают с именами глобальных объектов модуля.

Еще одно преимущество модулей заключается в том, что обеспечивается возможность зависимой (т.е. контролируемой) раздельной компиляции программных единиц. Это объясняется тем что информация, специфицируемая в модуле, доступна процессору при компиляции программных единиц, использующих данный модуль. Зависимая и в то же время раздельная компиляция повышает надежность программ и позволяет компилятору выполнять более широкий спектр оптимизирующих преобразований.

6.3. *Динамические массивы*

Инкапсуляция реализуется не только с помощью модулей. Наличие различных механизмов динамического размещения объектов также обеспечивает поддержку инкапсуляции и абстракции данных.

Инкапсуляция достигается за счет того, что размер массива, создание и освобождение (уничтожение) производятся в самой процедуре и при вызове об этом можно не заботиться. Массивы, которые необходимы только в процедуре, скрыты внутри этой процедуры, и остальная программа не должна их касаться. Такой вид инкапсуляции имеет те же преимущества: автор процедуры может, например, менять размер массива без изменения тех программных единиц, которые

используют эту процедуру, что облегчает разделение работы между разработчиками.

В Фортране 77 не допускались массивы переменной длины. Поскольку программные единицы транслировались раздельно, длина массива должна была объявляться явно в вызывающей программной единице и передаваться в процедуру через аргументы или через COMMON-блоки.

В Фортране 90/95 длины могут быть объявлены только в процедуре, что упрощает вызов, уменьшает число аргументов. Это объясняется тем, что все массивы, которые нужны только внутри процедуры, могут быть скрыты в процедуре, и другие программные единицы могут о них ничего не знать.

В современных стандартах имеются следующие механизмы динамического размещения массивов:

- автоматические массивы, границы которых вычисляются при входе в процедуру;
- размещаемые массивы, границы которых вычисляются при выполнении программы;
- массивы, создаваемые в процессе выполнения программы, т.е. массивы, которые не были заранее явно объявлены в программе.

Эти средства позволяют пользователю организовать работу с массивами, размер которых заранее не известен, а также оперативно отводить память под массивы, требующиеся на том или ином этапе выполнения программы и освобождать память по мере необходимости.

6.3.1. Автоматические массивы

Автоматические массивы создаются при входе в процедуру, их можно использовать только внутри процедуры, и они уничтожаются (т.е. освобождается занимаемая ими память) при выходе; таким образом, автоматические массивы могут быть неизвестны вне процедуры.

Пример

```
SUBROUTINE SWAP (X, Y)
REAL, DIMENSION (:) :: X, Y
REAL, DIMENSION (SIZE (X)) :: TEMP
!     TEMP – автоматический массив
TEMP = X
X = Y
Y = TEMP
END SUBROUTINE SWAP
```

Автоматические массивы не являются формальными аргументами, их границы и размер вычисляются при входе в процедуру.

Если в процедуре появляется временный массив, с переменными границами, который используется только внутри процедуры, в Фортране 77 он должен быть формальным аргументом. В Фортране 90/95, если такой массив не нужен на выходе, его можно исключить из списка аргументов, и вызывающая программа может о нем не знать. Так в приведенном примере массив TEMP в Фортране 77 должен быть формальным аргументом, и при вызове SWAP его размер должен быть определен. В современном Фортране информация о массиве TEMP при вызове процедуры не нужна.

Примечание

В приведенном примере массивы X и Y – массивы с передаваемой конфигурацией. Это тоже новая черта Фортрана 90/95; конфигурация формального аргумента перенимается у фактического.

6.3.2. Размещаемые массивы

Размещаемые массивы отличаются от автоматических тем, что создание и уничтожение выполняется полностью под контролем программиста. Такие массивы создаются при выполнении оператора ALLOCATE и уничтожаются при выполнении DEALLOCATE (или при выходе из процедуры, если они не имеют атрибута SAVE).

Пример

```
REAL, DIMENSION(:, : ) ALLOCATABLE :: AR1, AR2
. . .
READ  (*, *) NX, NY
ALLOCATE  AR1 (NX, NY), AR2 (NX, NY)
```

Размещаемые массивы должны быть объявлены с атрибутом ALLOCATABLE; при объявлении размещаемого массива указывается только его размерность (ранг). При выполнении оператора размещения ALLOCATE вычисляются границы массива, его размер и отводится память. Память, занимаемая массивом, освобождается при выполнении оператора DEALLOCATE.

Как и в случае автоматических массивов, такие массивы можно не включать в список аргументов, хотя они не являются массивами с постоянными границами.

6.4. *Обсуждение*

Подводя итог, еще раз отметим что инкапсуляция упрощает нескольким авторам разрабатывать программу независимо друг от друга, облегчает

модернизацию программы для другой архитектуры, или для других требований. Инкапсуляция также упрощает использование процедур. Это объясняется тем, что детали реализации скрыты от пользователя: даже если автор процедуры вносит изменения внутри, пользователь не должен менять свои коды. Использование механизма инкапсуляции особенно полезно при создании библиотек и пакетов, ориентированных на конкретную область.

7. Статический полиморфизм

7.1. *Общие сведения*

Полиморфизм означает возможность для программиста использовать какой-либо объект более чем одним способом.

Примером полиморфизма может служить операция деления в Фортране. Если операнды целого типа - выполняется целочисленное деление, если хотя бы один операнд вещественного типа - выполняется деление с плавающей точкой, для комплексных операндов выполняется другая операция. В таких случаях говорят, что операция перегружена.

При статическом полиморфизме конкретизация способа использования выполняется на этапе компиляции. Статический полиморфизм поддерживается в Фортране 90/95 следующим образом:

- явное и неявное приведение типов (см. 7.2.);
- перегрузка операций (см. 7.3.);
- перегрузка процедур (родовые процедуры и процедуры с необязательными аргументами; см. 7.4 и 7.5.).

Рассмотрим каждую из этих компонент более подробно.

7.2. *Приведение типов*

Приведение типов требуется в тех случаях, когда операнды одной операции (кроме возведения в степень) или левая и правая части оператора присваивания имеют разный тип. Неявное приведение типов выполняется автоматически компилятором (это традиционно для Фортрана), при этом выполняется автоматическое приведение к более широкому типу. Более надежным является явное приведение типов с помощью встроенных функций (REAL, DBLE, CMPLX, INT и др.), что также традиционно для Фортрана.

7.3. *Перегрузка операций*

Перегрузка операций означает использование одного и того же имени для нескольких разных операций в общей области действия. Реализация операции зависит от типов операндов.

В современном Фортране такие средства предусмотрены как для встроенных операций и присваиваний (это допускалось и в прежнем Фортране), так и для определяемых пользователем операций и присваиваний.

Если знак определяемой операции совпадает со знаком одной из встроенных операций, то типы операндов определяемой операции должны отличаться от типов операндов данной встроенной операции. Например, знак “+” можно расширить для сложения объектов типа INTERVAL (см. пример в 5.2. и 6.2.).

Перегрузка операций допускается и для операций с определяемыми знаками операций. Типы и параметры типа операндов определяемой операции должны соответствовать типам и параметрам типа формальных аргументов функции, описывающей данную операцию. Соответствие типа операнда и типа формального аргумента производного типа устанавливается по принципу именной эквивалентности. То же относится и к операторам присваивания. Примеры были приведены выше (см. 5.2. и 5.3.).

Операции над скалярами и массивами – еще один пример перегрузки операций. Любая встроенная операция и многие стандартные функции (называемые поэлементными) могут быть применимы как к скалярам, так и к массивам (согласованным по конфигурации, т.е. имеющим одинаковую размерность и размер по каждому измерению). Операции выполняются на поэлементной основе и вырабатывают результат – массив.

Пример

```
REAL, DIMENSION (100, 100) :: A, B, X, Z
Z = A + B* SIN (X)
```

7.4. *Перегрузка процедур*

Перегрузка процедур означает использование одного и того же имени процедуры (обощенного имени) для разных процедур.

В Фортране 77 такие средства были только для встроенных процедур (универсальные имена). Например, функция REAL (A) выполняет разные операции в зависимости от типа аргумента, хотя для этих операций существуют специфические имена FLOAT и SNGL.

В современном Фортране процедуры, определяемые пользователем, также могут быть настраиваемыми (для использования таких процедур требуется интерфейсный блок). Разные процедуры объединяются под одним обобщенным (родовым) именем.

Пример

```
INTERFACE  SWITCH
  SUBROUTINE INT_SWITCH (X, Y)
    INTEGER X, Y
  END SUBROUTINE INT_SWITCH
  SUBROUTINE REAL_SWITCH (X, Y)
    REAL X, Y
  END SUBROUTINE REAL_SWITCH
END INTERFACE
```

К каждой из процедур INT_SWITCH и REAL_SWITCH, указанных в этом примере, очевидно, можно обратиться с помощью оператора CALL, в котором указывается имя процедуры и список аргументов. Кроме того, интерфейсный блок в такой форме, как он задан в примере, позволяет обратиться к любой из процедур, с помощью родового имени SWITCH, например:

```
CALL SWITCH (P, R)
```

Если фактические аргументы P и R имеют целый тип, то данный вызов представляет собой обращение к процедуре INT_SWITCH, если вещественный, то к процедуре REAL_SWITCH.

Объединяемые процедуры могут отличаться не только типом аргументов, как в приведенном примере. Конкретный вызов должен быть применим не более чем к одной из специфицированных процедур.

7.5. *Процедуры с необязательными аргументами*

Необязательные аргументы также обеспечивают перегрузку процедур. Если формальный аргумент объявлен как необязательный, т.е. если он указан в операторе OPTIONAL или имеет атрибут OPTIONAL, то соответствующий фактический аргумент при обращении к процедуре может быть опущен.

Для вызова процедур с необязательными аргументами необходим интерфейсный блок.

Пример процедуры с необязательными аргументами:

```
SUBROUTINE S (F, X, METHOD, PR)
  INTEGER, OPTIONAL :: PR, METHOD
  ...
END SUBROUTINE S
```

Процедура, приведенная в примере, может быть вызвана, например, следующим образом:

```
CALL S (F1, Y, PR = 6)
```

Использование одной и той же процедуры с разным набором аргументов – еще один пример перегрузки процедур.

8. Наследование

Наследование означает, что объект может перенимать описание типа и операций от соответствующего родительского объекта. Кроме того, для порожденного объекта могут быть определены дополнительные свойства. Таким образом, наследование позволяет использовать общие свойства объекта и подогнать их под специфические потребности.

Наследование в Фортране 90/95 реализуется частично посредством производного типа; полное наследование в явном виде не поддерживается, но может быть смоделировано. Ниже обсуждается как это можно сделать.

Как уже отмечалось (см. 4.), компоненты производного типа могут иметь ранее описанный производный тип. Это означает, что производный тип может быть получен из другого производного типа добавлением компонент так, что новый тип получает копии значений и операций, которые были описаны (определены) для порождающего (родительского) типа. Порождающий тип называют также базовым типом. Базовый тип обладает общими свойствами.

Пример

```
! Порождающий (базовый) тип
TYPE PERSON
  INTEGER AGE
  CHARACTER (LEN = 15) NAME
  INTEGER ID_NUMB
END TYPE PERSON
```

```
! Порожденный тип
TYPE STUDENT
  TYPE (PERSON) STUD_A
  CHARACTER (LEN = 12) FACULITY
  INTEGER GROUP
  INTEGER YEAR
END TYPE STUDENT
```

В приведенном примере тип STUDENT содержит одну компоненту PERSON и три дополнительные компоненты (FACULITY, GROUP и YEAR).

Аналогичным образом можно создать другой тип из PERSON – тип TEACHER, который содержит новую компоненту SALARY.

```
TYPE TEACHER
    TYPE (PERSON)  TEACH_B
     INTEGER  SALARY
END TYPE TEACHER
```

Кроме того, можно создать типы студентов и преподавателей конкретного университета и таким способом создать иерархию типов и семейство типов. Таким образом, производные типы могут создаваться иерархически.

Так, если для массивов типа PERSON были описаны некоторые операции (например, упорядочение по алфавиту или по возрасту элементов массивов указанного типа), то эти операции могут быть применимы и к данным, имеющим тип STUDENT или TEACHER. Для порожденных типов можно создать и новые операции и/или модифицировать (расширить) операции, описанные для порождающего типа. При этом изменения структуры порождающего типа и описанных для него операций могут не оказывать влияния на порожденный тип и дополнительные операции для него.

Целесообразность применения наследования заключается в том, чтобы избежать дублирования кодов при создании и использовании типов похожих друг на друга, имеющих общие свойства и принадлежащих одной предметной области.

9. Динамический полиморфизм (динамическое связывание)

Выше (см. раздел 7.) были описаны средства реализации статического полиморфизма, при котором одно и то же имя (операции или процедуры) можно использовать в разной интерпретации. При этом конкретизация способа использования могла быть определена на этапе компиляции. При динамическом

полиморфизме (иногда его называют динамическим связыванием) надо написать процедуры, которые могут выполняться для всех типов наследственной иерархии, и при этом выбор конкретной процедуры для исполнения, определяется во время выполнения программы, т.е. на этапе выполнения производится связывание имени с конкретной процедурой.

Динамический полиморфизм, как и наследование (см. 8.), полезен в тех случаях, когда имеются похожие друг на друга (но не идентичные) объекты, которые могут быть обработаны обобщенными процедурами. Например, существует много различных типов студентов и преподавателей. Задача заключается в том, чтобы избежать написания различных программ для каждого из них и в то же время иметь возможность обрабатывать отдельно каждый тип.

В явном виде динамический полиморфизм в Фортране 90/95 отсутствует, но может быть смоделирован, хотя не всегда это сделать просто.

Для реализации такого вида полиморфизма можно использовать аппарат указателей, который имеется в современных стандартах Фортрана. Надо создать производный тип, который содержит в качестве компонент указатели на каждый возможный тип в наследственной иерархии. Затем реализовать обощенную процедуру, которая будет проверять во время выполнения, связан ли в данный момент указатель с тем или иным объектом (это делается с помощью стандартной функции ASSOCIATED) и в зависимости от этого вызывать нужную процедуру. Пример реализации динамического полиморфизма (как и других концепций ООП) читатель может найти в работе [9].

10. Заключение и перспективы

При проектировании объектно-ориентированной программы проблема заключается в том, чтобы выделить объекты для конкретной прикладной задачи, выделить операции, которые требуется выполнить, и разбить программу на соответствующие модули. Решение этой проблемы во многом зависит от конкретной задачи. Рамки препринта не позволяют уделить достаточное внимание указанной проблеме; эта тема хорошо освещена в литературе. Наша цель была другая: показать как можно использовать основные идеи ООП при программировании на современном Фортране.

Мы показали, что Фортран 90/95 поддерживает многие черты полезные для ООП (производные типы, определяемые пользователем операции, модули, обеспечивающие инкапсуляцию и абстракцию объектов, перегрузка операций и

процедур, динамические массивы и др.). Однако в явном виде отсутствуют такие черты как наследование и динамический полиморфизм. Отсутствующие черты могут быть смоделированы без дублирования кодов. Это позволяет реализовать все важные концепции ООП, хотя в некоторых случаях с большими усилиями чем в объектно-ориентированных языках.

Во введении уже отмечалось, что в настоящее время разрабатывается проект будущего стандрта языка (рабочее название - Фортран 2000). Путем добавления нескольких небольших расширений в этом проекте определен полный набор средств поддержки ООП. В частности, для реализации механизма наследования в полном объеме вводятся расширяемые (extensible) типы. Вводятся также средства реализации динамического полиморфизма. В некоторых реализациях новые средства могут появиться и до официального утверждения нового стандарта.

Литература

1. ISO/IEC 1539-1: 1997 Information technology - Programming languages - Fortran - Part 1: Base Language
2. ISO/IEC 1539: 1991(E) Information technology - Programming languages - Fortran
3. Фортран 90. Международный стандарт. Перевод с англ. Дробышевич С.Г., редактор перевода Горелик А.М. М.: Финансы и статистика, 1998
4. Горелик А.М., Ушкова В.Л. Фортран сегодня и завтра. М.: Наука, 1990
5. Меткалф М., Рид Дж. Описание языка программирования Фортран 90. Перевод с английского. М.: Мир, 1995
6. Горелик А.М. Современные международные стандарты языка Фортран. //Программирование, 2001, N6
7. Пол А. Объектно-ориентированное программирование на C++. М.: ”Невский диалект” – “Изд. Бином”, 1999
8. Бен-Ари М. Языки программирования. Перевод с английского. М.: Мир, 2000
9. Decyk V., Norton C., Szymanski. How to Express C++ concepts in Fortran 90. //Scientific Programming, vol.6, №4, 1997.
10. Cary J., Shasharina S., Cummings J., Reinders J., Hinker P. Comparison of C++ and Fortran 90 for object-oriented Scientific programming. //Computer Physics Communications. 105, 1997.