Бесплатная электронная книга

УЧУСЬ

Fortran

Free unaffiliated eBook created from
Stack Overflow contributors.

#fortran

1
1: Fortran 2
2
2
Examples 2
2
, 3
4
5
2: C 6
Examples 6
C 6
C 7
3: I / O 8
8
Examples 8
- 8
9
10
4: 12
12
Examples 12
PACK , 12
5: 14
Examples 14
14
14
16
16
18
6: 20

Examples....20
....20
SELECT CASE....21
DO....22
WHERE....24
7: 26
Examples....26
....26
....27
....27
: 30
....30
....30
....31
....31
....31
, ...32
....32
....32
....33
....34
....34
....34
....35
....35
....35
: 36
....36
....37
....37
8: - 38
Examples....38

38
38
39
40
41
9: 43
Examples 43
Fortran 43
44
44
45
45
46
10: - 49
49
Examples 49
49
50
51
51
52
11: (.f, .f90, .f95, ...) , 55
55
Examples 55
55
12: 57
Examples 57
57
if 58
DO 59
59
61

63

GOTO....65

GOTO....65

....65

....67

13: **69**

Examples....69

....69

....70

....71

....73

....75

....76

....77

....78

14: **79**

Examples....79

/ 79

....80

....**82**

Около

You can share this PDF with anyone you feel could benefit from it, downloaded the latest version from: fortran

It is an unofficial and free Fortran ebook created for educational purposes. All the content is extracted from Stack Overflow Documentation, which is written by many hardworking individuals at Stack Overflow. It is neither affiliated with Stack Overflow nor official Fortran.

The content is released under Creative Commons BY-SA, and the list of contributors to each chapter are provided in the credits section at the end of this book. Images may be copyright of their respective owners unless otherwise specified. All trademarks and registered trademarks are the property of their respective company owners.

Use the content presented in this book at your own risk; it is not guaranteed to be correct nor accurate, please send your feedback and corrections to info@zzzprojects.com

глава 1: Начало работы с Fortran

замечания

Фортран - это язык, широко используемый в научном сообществе из-за его пригодности для численного расчета. Особенно привлекательной является его интуитивная запись в виде массива, которая упрощает запись быстрых векторизованных вычислений.

Несмотря на свой возраст, Fortran по-прежнему активно развивается с многочисленными реализациями, включая GNU, Intel, PGI и Cray.

Версии

Версия	Заметка	Релиз
ФОРТРАН 66	Первая стандартизация ASA (теперь ANSI)	1966-03-07
ФОРТРАН 77	Фиксированная форма, историческая	1978-04-15
Фортран 90	Бесплатная форма, стандарт ISO, операции с массивом	1991-06-15
Фортран 95	Чистые и элементарные процедуры	1997-06-15
Fortran 2003	Объектно-ориентированное программирование	2004-04-04
Fortran 2008	Co-Массивы	2010-09-10

Examples

Установка или настройка

Fortran - это язык, который может быть скомпилирован с использованием компиляторов, поставляемых многими поставщиками. Различные компиляторы доступны для разных аппаратных платформ и операционных систем. Некоторые компиляторы являются свободным программным обеспечением, некоторые могут использоваться бесплатно, а некоторые требуют покупки лицензии.

Наиболее распространенным свободным компилятором Fortran является GNU Fortran или gfortran. Исходный код доступен из GNU в составе GCC, сборника компиляторов GNU. Бинарные файлы для многих операционных систем доступны по адресу https://gcc.gnu.org/wiki/GFortranBinaries . Распределения Linux часто содержат gfortran в своем диспетчере пакетов.

Другие компиляторы доступны, например:

- EKOPath by PathScale
- LLVM (бэкэнд через DragonEgg)
- Oracle Developer Studio

- Компилятор Absoft Fortran
- Компилятор Intel Fortran
- Компилятор Fortran NAG
- Компиляторы PGI

В HPC-системах часто доступны специализированные компиляторы, доступные системному провайдеру, например, компиляторы IBM или Cray .

Все эти компиляторы поддерживают стандарт Fortran 95. Обзор состояния Fortran 2003 и статуса Fortran 2008 различными компиляторами предлагается Форумом ACM Fortran и доступен в Вики-версии Fortran.

Привет, мир

Любая программа Fortran должна включать `end` качестве последнего оператора. Поэтому простейшая программа Fortran выглядит так:

```
end
```

Вот несколько примеров программ «привет, мир»:

```
print *, "Hello, world"
end
```

С помощью оператора `write` :

```
write(*,*) "Hello, world"
end
```

Для ясности теперь принято использовать инструкцию `program` для запуска программы и присвоения ей имени. Затем оператор `end` может ссылаться на это имя, чтобы сделать его очевидным, о чем идет речь, и дать компилятору проверить правильность кода. Кроме того, все программы Fortran должны содержать `implicit none` оператор. Таким образом, минимальная программа Fortran должна выглядеть следующим образом:

```
program hello
  implicit none
  write(*,*) 'Hello world!'
end program hello
```

Следующий логический шаг от этого момента - это увидеть результат приветственной

мировой программы. В этом разделе показано, как добиться этого в среде Linux. Мы предполагаем, что у вас есть некоторые основные понятия команд оболочки , в основном вы знаете, как добраться до терминала оболочки. Мы также предполагаем, что вы уже настроили среду `fortran` . Используя предпочтительный текстовый редактор (блокнот, блокнот ++, vi, vim, emacs, gedit, kate и т. Д.), Сохраните программу приветствия выше (скопируйте и вставьте) в файл с именем `hello.f90` в вашем домашнем каталоге. `hello.f90` - ваш исходный файл. Затем перейдите в командную строку и перейдите в каталог (домашний каталог?), Где вы сохранили исходный файл, затем введите следующую команду:

```
>gfortran -o hello hello.f90
```

Вы только что создали свою исполняемую программу hello world. В техническом плане вы просто скомпилировали свою программу. Чтобы запустить его, введите следующую команду:

```
>./hello
```

На вашем терминале оболочки вы должны увидеть следующую строку.

```
> Hello world!
```

Поздравляем, вы только что написали, скомпилировали и запустили программу Hello World.

Квадратное уравнение

Сегодня Fortran в основном используется для численного расчета. Этот очень простой пример иллюстрирует основную программную структуру для решения квадратичных уравнений:

```
program quadratic
  !a comment

  !should be present in every separate program unit
  implicit none

  real :: a, b, c
  real :: discriminant
  real :: x1, x2

  print *, "Enter the quadratic equation coefficients a, b and c:"
  read *, a, b, c

  discriminant = b**2 - 4*a*c

  if ( discriminant>0 ) then

    x1 = ( -b + sqrt(discriminant)) / (2 * a)
    x2 = ( -b - sqrt(discriminant)) / (2 * a)
```

```
    print *, "Real roots:"
    print *, x1, x2

    ! Comparison of floating point numbers for equality is often not recommended.
    ! Here, it serves the purpose of illustrating the "else if" construct.
  else if ( discriminant==0 ) then

    x1 = - b / (2 * a)
    print *, "Real root:"
    print *, x1
  else

    print *, "No real roots."
  end if
end program quadratic
```

Нечувствительность к регистру

Прописные и строчные буквы алфавита эквивалентны в наборе символов Fortran. Другими словами, Fortran *нечувствителен к регистру* . Такое поведение контрастирует с чувствительными к регистру языками, такими как C ++ и многие другие.

Как следствие, переменные `a` и `A` являются `a` и той же переменной. В принципе можно написать программу следующим образом

```
pROgrAm MYproGRaM
..
enD mYPrOgrAM
```

Это хороший программист, чтобы избежать таких уродливых выборов.

Прочитайте Начало работы с Fortran онлайн: https://riptutorial.com/ru/fortran/topic/904/начало-работы-с-fortran

глава 2: C совместимость

Examples

Вызов C из Фортрана

В Fortran 2003 были введены языковые функции, которые могут гарантировать совместимость между C и Fortran (и до большего количества языков с использованием C в качестве посредника). Эти функции в основном доступны через встроенный модуль `iso_c_binding` :

```
use, intrinsic :: iso_c_binding
```

`intrinsic` ключевое слово здесь обеспечивает правильный модуль, а не созданный пользователем модуль с тем же именем.

`iso_c_binding` дает доступ к параметрам типа *совместимого* типа:

```
integer(c_int) :: foo    ! equivalent of 'int foo' in C
real(c_float) :: bar     ! equivalent of 'float bar' in C
```

Использование параметров типа типа C гарантирует, что данные могут быть переданы между программами C и Fortran.

Взаимодействие символов C char и Fortran, вероятно, является темой для себя и поэтому не обсуждается здесь

Чтобы на самом деле вызвать функцию C из Fortran, сначала должен быть объявлен интерфейс. Это по существу эквивалентно прототипу функции C и позволяет компилятору узнать о количестве и типе аргументов и т. Д. Атрибут `bind` используется, чтобы сообщить компилятору имя функции в C, которое может отличаться от Fortran название.

geese.h

```
// Count how many geese are in a given flock
int howManyGeese(int flock);
```

geese.f90

```
! Interface to C routine
interface
  integer(c_int) function how_many_geese(flock_num) bind(C, 'howManyGeese')
    ! Interface blocks don't know about their context,
    ! so we need to use iso_c_binding to get c_int definition
    use, intrinsic :: iso_c_binding, only : c_int
```

```
    integer(c_int) :: flock_num
  end function how_many_geese
end interface
```

Программа Fortran должна быть связана с библиотекой C (здесь *зависит от компилятора?*), `howManyGeese()` включает реализацию `howManyGeese()` , а затем `how_many_geese()` можно вызвать из Fortran.

Структуры C в Фортране

Атрибут `bind` также может применяться к производным типам:

geese.h

```
struct Goose {
   int flock;
   float buoyancy;
}

struct Goose goose_c;
```

geese.f90

```
use, intrinsic :: iso_c_binding, only : c_int, c_float

type, bind(C) :: goose_t
  integer(c_int) :: flock
  real(c_float) :: buoyancy
end type goose_t

type(goose_t) :: goose_f
```

Теперь данные могут передаваться между `goose_c` и `goose_f` . C-процедуры, которые принимают аргументы типа `Goose` могут быть вызваны из Fortran с `type(goose_t)` .

Прочитайте C совместимость онлайн: https://riptutorial.com/ru/fortran/topic/2184/c-совместимость

глава 3: I / O

Синтаксис

- `WRITE(unit num, format num)` выводит данные после скобок в новой строке.
- `READ(unit num, format num)` вводится в переменную после скобок.
- `OPEN(unit num, FILE=file)` открывает файл. (Есть больше возможностей для открытия файлов, но они не важны для ввода-вывода.
- `CLOSE(unit num)` закрывает файл.

Examples

Простой ввод-вывод

В качестве примера ввода ввода и вывода мы возьмем реальное значение и вернем значение и его квадрат, пока пользователь не введет отрицательное число.

Как указано ниже, команда `read` принимает два аргумента: номер устройства и спецификатор формата. В приведенном ниже примере мы используем `*` для номера единицы (который указывает stdin) и `*` для формата (который в данном случае указывает значение по умолчанию для реалов). Мы также указываем формат для оператора `print` . В качестве альтернативы можно использовать `write(*,"The value....")` или просто игнорировать форматирование и использовать его как

```
print *,"The entered value was ", x," and its square is ",x*x
```

что, вероятно, приведет к некоторым странно разнесенным строкам и значениям.

```
program SimpleIO
   implicit none
   integer, parameter :: wp = selected_real_kind(15,307)
   real(kind=wp) :: x

   ! we'll loop over until user enters a negative number
   print '("Enter a number >= 0 to see its square. Enter a number < 0 to exit.")'
   do
      ! this reads the input as a double-pricision value
      read(*,*) x
      if (x < 0d0) exit
      ! print the entered value and it's square
      print '("The entered value was ",f12.6,", its square is ",f12.6,".")',x,x*x
   end do
   print '("Thank you!")'

end program SimpleIO
```

Чтение с некоторой проверкой ошибок

Современный пример Фортрана, который включает проверку ошибок и функцию для получения нового номера устройства для файла.

```
module functions

contains

    function get_new_fileunit() result (f)
        implicit none

        logical    :: op
        integer    :: f

        f = 1
        do
            inquire(f,opened=op)
            if (op .eqv. .false.) exit
            f = f + 1
        enddo

    end function

end module

program file_read
    use functions, only : get_new_fileunit
    implicit none

    integer            :: unitno, ierr, readerr
    logical            :: exists
    real(kind(0.d0))   :: somevalue
    character(len=128) :: filename

    filename = "somefile.txt"

    inquire(file=trim(filename), exist=exists)
    if (exists) then
        unitno = get_new_fileunit()
        open(unitno, file=trim(filename), action="read", iostat=ierr)
        if (ierr .eq. 0) then
            read(unitno, *, iostat=readerr) somevalue
            if (readerr .eq. 0) then
                print*, "Value in file ", trim(filename), " is ", somevalue
            else
                print*, "Error ", readerr, &
                        " attempting to read file ", &
                        trim(filename)
            endif
        else
            print*, "Error ", ierr ," attempting to open file ", trim(filename)
            stop
        endif
    else
        print*, "Error -- cannot find file: ", trim(filename)
        stop
    endif
```

```
end program file_read
```

Передача аргументов командной строки

Если аргументы командной строки поддерживаются, их можно прочитать через встроенный `get_command_argument` (введенный в стандарте Fortran 2003). Свойство `command_argument_count` позволяет узнать количество аргументов, предоставленных в командной строке.

Все аргументы командной строки считываются как строки, поэтому для числовых данных должно быть выполнено преобразование внутреннего типа. В качестве примера этот простой код суммирует два числа, указанные в командной строке:

```
PROGRAM cmdlnsum
IMPLICIT NONE
CHARACTER(100) :: num1char
CHARACTER(100) :: num2char
REAL :: num1
REAL :: num2
REAL :: numsum

!First, make sure the right number of inputs have been provided
IF(COMMAND_ARGUMENT_COUNT().NE.2)THEN
  WRITE(*,*)'ERROR, TWO COMMAND-LINE ARGUMENTS REQUIRED, STOPPING'
  STOP
ENDIF

CALL GET_COMMAND_ARGUMENT(1,num1char)   !first, read in the two values
CALL GET_COMMAND_ARGUMENT(2,num2char)

READ(num1char,*)num1                    !then, convert them to REALs
READ(num2char,*)num2

numsum=num1+num2                        !sum numbers
WRITE(*,*)numsum                        !write out value

END PROGRAM
```

Аргумент number в `get_command_argument` полезен в диапазоне от `0` до результата `command_argument_count` . Если значение равно `0` тогда указывается имя команды (если поддерживается).

Многие компиляторы также предлагают нестандартные встроенные функции (например, `getarg`) для доступа к аргументам командной строки. Поскольку они нестандартны, следует проконсультироваться с соответствующей документацией компилятора.

Использование `get_command_argument` может быть расширено за пределами приведенного выше примера аргументами `length` и `status` . Например, с

```
character(5) arg
```

```
integer stat
call get_command_argument(number=1, value=arg, status=stat)
```

значение `stat` будет равным `-1` если существует первый аргумент и имеет длину больше 5. Если есть какая-то другая трудность при извлечении аргумента, значение `stat` будет некоторым положительным числом (и `arg` будет состоять полностью из пробелов). В противном случае его значение будет равно `0` .

Аргумент `length` может быть объединен с переменной длины отложенной длины, например, в следующем примере.

```
character(:), allocatable :: arg
integer arglen, stat
call get_command_argument(number=1, length=arglen)  ! Assume for simplicity success
allocate (character(arglen) :: arg)
call get_command_argument(number=1, value=arg, status=stat)
```

Прочитайте I / O онлайн: https://riptutorial.com/ru/fortran/topic/6778/i---o

глава 4: Внутренние процедуры

замечания

Многие из доступных внутренних процедур имеют общие типы аргументов. Например:

- логический аргумент `MASK` который выбирает элементы входных массивов для обработки
- целочисленный скалярный аргумент `KIND` который определяет вид результата функции
- целочисленный аргумент `DIM` для функции сокращения, который управляет размером, над которым выполняется сокращение

Examples

Использование PACK для выбора элементов, удовлетворяющих условию

Внутренняя функция `pack` объединяет массив в вектор, выбирая элементы на основе заданной маски. Функция имеет две формы

```
PACK(array, mask)
PACK(array, mask, vector)
```

(т. е. `vector` аргумент необязателен).

В обоих случаях `array` - это массив, а также `mask` логического типа и соответствующая `array` (либо скаляр, либо массив той же формы).

В первом случае результатом является массив ранга 1 типов и типов `array` с числом элементов, являющимся числом истинных элементов в маске.

```
integer, allocatable :: positive_values(:)
integer :: values(5) = [2, -1, 3, -2, 5]
positive_values = PACK(values, values>0)
```

приводит к `positive_values` являющимся массивом `[2, 3, 5]` .

При наличии аргумента `vector` rank-1 результат теперь представляет собой размер `vector` (у которого должно быть как минимум столько элементов, сколько есть истинных значений в `mask` .

Эффект с `vector` заключается в том, чтобы вернуть этот массив с исходными элементами этого массива, перезаписанными маскированными элементами `array` . Например

```
integer, allocatable :: positive_values(:)
integer :: values(5) = [2, -1, 3, -2, 5]
positive_values = PACK(values, values>0, [10,20,30,40,50])
```

приводит к `positive_values` являющимся массивом `[2,3,5,40,50]` .

Следует отметить, что независимо от формы `array` аргументов результат всегда является массивом ранга-1.

В дополнение к выбору элементов массива, удовлетворяющих условию маскировки, часто бывает полезно определить индексы, для которых выполняется условие маскировки. Эта общая идиома может быть выражена как

```
integer, allocatable :: indices(:)
integer i
indices = PACK([(i, i=1,5)], [2, -1, 3, -2, 5]>0)
```

что приводит к тому, что `indices` составляют массив `[1,3,5]` .

Прочитайте Внутренние процедуры онлайн: https://riptutorial.com/ru/fortran/topic/2643/внутренние-процедуры

глава 5: Использование модулей

Examples

Синтаксис модуля

Модуль представляет собой набор объявлений типов, деклараций и процедур данных. Основной синтаксис:

```
module module_name
  use other_module_being_used

  ! The use of implicit none here will set it for the scope of the module.
  ! Therefore, it is not required (although considered good practice) to repeat
  ! it in the contained subprograms.
  implicit none

  ! Parameters declaration
  real, parameter, public :: pi = 3.14159
  ! The keyword private limits access to e parameter only for this module
  real, parameter, private :: e = 2.71828

  ! Type declaration
  type my_type
    integer :: my_int_var
  end type

  ! Variable declaration
  integer :: my_integer_variable

! Subroutines and functions belong to the contains section
contains

  subroutine my_subroutine
    !module variables are accessible
    print *, my_integer_variable
  end subroutine

  real function my_func(x)
    real, intent(in) :: x
    my_func = x * x
  end function my_func
end module
```

Использование модулей из других программных модулей

Для доступа к объектам, объявленным в модуле из другого программного модуля (модуля, процедуры или программы), модуль должен *использоваться* с оператором `use` .

```
module shared_data
  implicit none
```

```
  integer :: iarray(4) = [1, 2, 3, 4]
  real :: rarray(4) = [1., 2., 3., 4.]
end module


program test

  !use statements most come before implicit none
  use shared_data

  implicit none

  print *, iarray
  print *, rarray
end program
```

Оператор `use` поддерживает импорт только выбранных имен

```
program test

  !only iarray is accessible
  use shared_data, only: iarray

  implicit none

  print *, iarray

end program
```

К объектам также можно обращаться под другим именем, используя список *переименований* :

```
program test

  !only iarray is locally renamed to local_name, rarray is still acessible
  use shared_data, local_name => iarray

  implicit none

  print *, local_name

  print *, rarray

end program
```

Кроме того, переименование может быть объединено с `only` вариантом

```
program test
  use shared_data, only : local_name => iarray
end program
```

так что доступен только объект `iarray` модуля, но он имеет локальное имя `local_name` .

Если для импорта имен отмечены как *частные,* вы не можете импортировать их в свою программу.

Внутренние модули

Fortran 2003 представил встроенные модули, которые предоставляют доступ к специальным именованным константам, производным типам и процедурам модулей. В настоящее время существует пять стандартных встроенных модулей:

- `ISO_C_Binding` ; поддержка совместимости C;
- `ISO_Fortran_env` ; подробное описание среды Fortran;
- `IEEE_Exceptions` , `IEEE_Arithmetic` и `IEEE_Features` ; поддерживая так называемый арифметический объект IEEE.

Эти встроенные модули являются частью библиотеки Fortran и доступны как другие модули, за исключением того, что оператор `use` может иметь внутренний характер:

```
use, intrinsic :: ISO_C_Binding
```

Это гарантирует, что встроенный модуль используется, когда доступен доступный пользователю модуль с таким же именем. Наоборот

```
use, non_intrinsic :: ISO_C_Binding
```

обеспечивает доступ к тому же самому доступному пользователю модулю (который должен быть доступен), а не к внутреннему модулю. Без характера модуля, указанного в

```
use ISO_C_Binding
```

доступный неагрессивный модуль будет предпочтительнее встроенного модуля.

Внутренние модули IEEE отличаются от других модулей тем, что их доступность в модуле определения области видимости может изменять поведение кода там даже без ссылки на любое из сущностей, определенных в них.

Контроль доступа

Доступность символов, объявленных в модуле, может контролироваться с использованием `private` и `public` атрибутов и операторов.

Синтаксис формы заявления:

```
!all symbols declared in the module are private by default
private

!all symbols declared in the module are public by default
public

!symbols in the list will be private
```

```
private :: name1, name2

!symbols in the list will be public
public :: name3, name4
```

Синтаксис формы атрибута:

```
integer, parameter, public :: maxn = 1000

real, parameter, private :: local_constant = 42.24
```

Доступ к общедоступным символам можно получить из модулей программы с использованием модуля, но частные символы не могут.

Если спецификация не используется, значение по умолчанию является `public` .

Спецификация доступа по умолчанию, использующая

```
private
```

или же

```
public
```

может быть изменено путем указания другого доступа с помощью *объекта-объявления-списка*

```
public :: name1, name2
```

или используя атрибуты.

Это управление доступом также влияет на символы, импортированные из другого модуля:

```
module mod1
  integer :: var1
end module

module mod2
  use mod1, only: var1

  public
end module

program test
  use mod2, only: var1
end program
```

возможно, но

```
module mod1
```

```
  integer :: var1
end module

module mod2
  use mod1, only: var1

  public
  private :: var1
end module

program test
  use mod2, only: var1
end program
```

невозможно, потому что `var` является приватным в `mod2` .

Защищенные объекты модуля

Помимо того, что модули модуля имеют доступ к управлению доступом (как `public` или `private`), объекты-объекты также могут иметь атрибут `protect` . Объект, защищенный общественностью, может быть связан с использованием, но используемый объект подлежит ограничениям на его использование.

```
module mod
  integer, public, protected :: i=1
end module

program test
  use mod, only : i
  print *, i   ! We are allowed to get the value of i
  i = 2        ! But we can't change the value
end program test
```

Объекту, защищенному общественностью, не разрешается указывать вне его модуля

```
module mod
  integer, public, target, protected :: i
end module mod

program test
  use mod, only : i
  integer, pointer :: j
  j => i   ! Not allowed, even though we aren't changing the value of i
end program test
```

Для общедоступного защищенного указателя в модуле ограничения разные. Защищен статус ассоциации указателя

```
module mod
  integer, public, target :: j
  integer, public, protected, pointer :: i => j
end module mod

program test
```

```
  use mod, only : i
  i = 2   ! We may change the value of the target, just not the association status
end program test
```

Как и с указателями переменной, указатели на процедуры также могут быть защищены, что снова предотвращает изменение целевой ассоциации.

Прочитайте Использование модулей онлайн: https://riptutorial.com/ru/fortran/topic/1139/использование-модулей

глава 6: Контроль выполнения

Examples

Если конструкция

Конструкция `if` (называемая блочным IF-заявлением в FORTRAN 77) является общей для многих языков программирования. Он условно выполняет один блок кода, когда логическое выражение оценивается как true.

```
[name:] IF (expr) THEN
    block
[ELSE IF (expr) THEN [name]
    block]
[ELSE [name]
   block]
END IF [name]
```

где,

- **name** - имя конструкции if (необязательно)
- **expr** - скалярное логическое выражение, заключенное в круглые скобки
- **block** - последовательность из нуля или более операторов или конструкций

Имя конструкции в начале оператора `if then` должно иметь то же значение, что и имя конструкции в `end if` , и оно должно быть уникальным для текущего модуля определения области видимости.

В операторах `if` , (in) равенства и логические выражения, оценивающие оператор, могут использоваться со следующими операторами:

```
.LT.  which is <   ! less than
.LE.           <=  ! less than or equal
.GT.           >   ! greater than
.GE.           >=  ! greater than or equal
.EQ.           =   ! equal
.NE.           /=  ! not equal
.AND.              ! logical and
.OR.               ! logical or
.NOT.              ! negation
```

Примеры:

```
! simplest form of if construct
if (a > b) then
    c =  b / 2
end if
!equivalent example with alternate syntax
```

```
if(a.gt.b)then
   c=b/2
endif

! named if construct
circle: if (r >= 0) then
    l = 2 * pi * r
end if circle

! complex example with nested if construct
block: if (a < e) then
    if (abs(c - e) <= d) then
        a = a * c
    else
        a = a * d
    end if
else
    a = a * e
end if block
```

Историческое использование конструкции `if` в так называемом «арифметическом if». Однако, поскольку это может быть заменено более современными конструкциями, это не рассматривается здесь. Более подробную информацию можно найти здесь .

Конструкция SELECT CASE

Конструкция `select case` условно выполняет один блок конструкций или операторов в зависимости от значения скалярного выражения в заявлении о `select case` . Эта конструкция управления может рассматриваться как замена для вычисленного `goto` .

```
[name:] SELECT CASE (expr)
[CASE (case-value [, case-value] ...) [name]
   block]...
[CASE DEFAULT [name]
   block]
END SELECT [name]
```

где,

- **name** - имя конструкции `select case` (необязательно)
- **expr** - скалярное выражение типа integer, логическое или символьное (заключено в круглые скобки)
- **case-value** - одно или несколько скалярных целочисленных, логических или символьных выражений инициализации, заключенных в круглые скобки
- **block** - последовательность из нуля или более операторов или конструкций

Примеры:

```
! simplest form of select case construct
select case(i)
case(:-1)
```

```
    s = -1
case(0)
    s = 0
case(1:)
    s = 1
case default
    print "Something strange is happened"
end select
```

В этом примере значение `(:-1)` case - это диапазон значений, соответствующий всем значениям, меньшим нуля, `(0)` соответствует нулям, и `(1:)` соответствует всем значениям выше нуля, раздел по `default` включает в себя, если другие разделы не выполняется.

Блок DO построить

Конструкция `do` представляет собой петлевую конструкцию, которая имеет ряд итераций, управляемых с помощью управления контуром

```
integer i
do i=1, 5
  print *, i
end do
print *, i
```

В приведенной выше форме переменная цикла `i` проходит через петлю 5 раз, принимая поочередно значения от 1 до 5. По завершении построения переменная цикла имеет значение 6, то есть **переменная цикла увеличивается еще раз после завершения цикла** .

В более общем плане конструкцию петли `do` можно понять следующим образом

```
integer i, first, last, step
do i=first, last, step
end do
```

Цикл начинается с `i` со значением `first` , увеличивая каждую итерацию на `step` пока `i` станет больше `last` (или меньше `last` если размер шага отрицательный).

Важно отметить, что поскольку Fortran 95, переменная цикла и выражения управления контуром должны быть целыми.

Итерация может быть завершена преждевременно с помощью инструкции `cycle`

```
do i=1, 5
  if (i==4) cycle
end do
```

и вся конструкция может прекратить выполнение с помощью оператора `exit`

```
do i=1, 5
  if (i==4) exit
end do
print *, i
```

`do` конструкции могут быть названы:

```
do_name: do i=1, 5
end do do_name
```

что особенно полезно, когда вложенные конструкторы `do`

```
do1: do i=1, 5
  do j=1,6
    if (j==3) cycle        ! This cycles the j construct
    if (j==4) cycle        ! This cycles the j construct
    if (i+j==7) cycle do1  ! This cycles the i construct
    if (i*j==15) exit do1  ! This exits the i construct
  end do
end do1
```

`do` конструкции могут также иметь неопределенный контур управление, либо «навсегда» или пока данное условие не будет выполнено

```
integer :: i=0
do
  i=i+1
  if (i==5) exit
end do
```

или же

```
integer :: i=0
do while (i<6)
  i=i+1
end do
```

Это также допускает бесконечный цикл `do` через `.true.` заявление

```
print *,'forever'
do while(.true.)
  print *,'and ever'
end do
```

Конструкция `do` может также оставлять порядок итераций неопределенным

```
do concurrent (i=1:5)
end do
```

отметив , что форма управления с обратной связью является таким же , как в `forall` контроля.

Существуют различные ограничения на операторы, которые могут выполняться в пределах диапазона `do concurrent` конструкции `do concurrent` которые предназначены для обеспечения отсутствия зависимостей между итерациями конструкции. Это явное указание программиста может обеспечить большую оптимизацию (включая распараллеливание) компилятором, которая может быть затруднена для определения иначе.

«Частные» переменные в рамках взаимодействия могут быть реализованы с использованием `block` конструкции внутри `do concurrent` :

```
do concurrent (i=1:5, j=2:7)
  block
    real tempval  ! This is independent across iterations
  end block
end do
```

В другой форме блока `do` create использует помеченный оператор `continue` вместо `end do` :

```
    do 100, i=1, 5
100 continue
```

Можно даже вложить такие конструкции с общим заявлением о завершении

```
    do 100, i=1,5
    do 100, j=1,5
100 continue
```

Обе эти формы, и особенно вторая (которая устаревает), как правило, следует избегать в интересах ясности.

Наконец, есть и не-блок `do` построить. Это также считается устаревшим и описано в `do` другом месте , а также методы по реструктуризации как блок `do` построить.

Конструкция WHERE

Конструкция `where` , доступная в Fortran90 и далее, представляет собой скрытую конструкцию `do` . Оператор маскирования следует тем же правилам инструкции `if` , но применяется ко всем элементам данного массива. Использование `where` позволяет выполнять операции над массивом (или несколькими массивами того же размера), элементы которого удовлетворяют определенному правилу. Это можно использовать для упрощения одновременных операций над несколькими переменными.

Синтаксис:

```
[name]: where (mask)
    block
[elsewhere (mask)
    block]
[elsewhere
    block]
end where [name]
```

Вот,

- **name** - это имя, присвоенное блоку (если указано)
- **mask** - это логическое выражение, применяемое ко всем элементам
- **block** - серия команд, которые должны выполняться

Примеры:

```
! Example variables
real:: A(5),B(5),C(5)
A = 0.0
B = 1.0
C = [0.0, 4.0, 5.0, 10.0, 0.0]

! Simple where construct use
where (C/=0)
    A=B/C
elsewhere
    A=0.0
end

! Named where construct
Block: where (C/=0)
    A=B/C
elsewhere
    A=0.0
end where Block
```

Прочитайте Контроль выполнения онлайн: https://riptutorial.com/ru/fortran/topic/1657/контроль-выполнения

глава 7: Массивы

Examples

Базовая нотация

Любой тип может быть объявлен как массив, используя либо атрибут *измерения,* либо просто указывая непосредственно `dimension` (ы) массива:

```
! One dimensional array with 4 elements
integer, dimension(4) :: foo

! Two dimensional array with 4 rows and 2 columns
real, dimension(4, 2) :: bar

! Three dimensional array
type(mytype), dimension(6, 7, 8) :: myarray

! Same as above without using the dimension keyword
integer :: foo2(4)
real :: bar2(4, 2)
type(mytype) :: myarray2(6, 7, 8)
```

Последний способ объявления многомерного массива позволяет объявлять массивы разного ранга / измерений одного типа в одной строке следующим образом

```
real :: pencil(5), plate(3,-2:4), cuboid(0:3,-10:5,6)
```

Максимальный ранг (количество измерений) разрешен 15 в стандарте Fortran 2008 и был 7 раньше.

Fortran хранит массивы в порядке *столбцов* . То есть элементы `bar` сохраняются в памяти следующим образом:

```
bar(1, 1), bar(2, 1), bar(3, 1), bar(4, 1), bar(1, 2), bar(2, 2), ...
```

В Fortran нумерация массивов начинается с **1** по умолчанию, в отличие от C, которая начинается с **0** . Фактически, в Fortran вы можете указать верхнюю и нижнюю границы для каждого измерения явно:

```
integer, dimension(7:12, -3:-1) :: geese
```

Это объявляет массив формы `(6, 3)` , первым элементом которого является `geese(7, -3)` .

Нижняя и верхняя границы по 2 (или более) размерам доступны по внутренним функциям `ubound` и `lbound` . Действительно, `lbound(geese,2)` вернется `-3` , тогда как `ubound(geese,1)`

вернется `12` .

Доступ к `size` массива можно получить с помощью встроенного функционального `size` . Например, `size(geese, dim = 1)` возвращает размер первого измерения, равный 6.

Распределяемые массивы

Массивы могут иметь атрибут *allocatable* :

```
! One dimensional allocatable array
integer, dimension(:), allocatable :: foo
! Two dimensional allocatable array
real, dimension(:,:), allocatable :: bar
```

Это объявляет переменную, но не выделяет для нее никакого пространства.

```
! We can specify the bounds as usual
allocate(foo(3:5))

! It is an error to allocate an array twice
! so check it has not been allocated first
if (.not. allocated(foo)) then
  allocate(bar(10, 2))
end if
```

Как только переменная больше не нужна, ее можно *освободить* :

```
deallocate(foo)
```

Если по какой - либо причине `allocate` утверждение не удается, программа остановится. Это можно предотвратить, если статус проверен с помощью ключевого слова `stat` :

```
real, dimension(:), allocatable :: geese
integer :: status

allocate(geese(17), stat=status)
if (stat /= 0) then
  print*, "Something went wrong trying to allocate 'geese'"
  stop 1
end if
```

Оператор `deallocate` имеет ключевое слово `stat` :

```
deallocate (geese, stat=status)
```

`status` - целочисленная переменная, значение которой равно 0, если выделение или освобождение было успешным.

Конструкторы массивов

Значение массива ранга-1 может быть создано с использованием *конструктора массива* с синтаксисом

```
(/ ... /)
[ ... ]
```

Форма `[...]` была введена в Fortran 2003 и обычно считается более понятной для чтения, особенно в сложных выражениях. Эта форма используется исключительно в этом примере.

Значения, отображаемые в конструкторе массива, могут быть скалярными значениями, значениями массива или подразумеваемыми циклами.

Параметры типа и типа построенного массива соответствуют значениям в конструкторе массива

```
[1, 2, 3]      ! A rank-1 length-3 array of default integer type
[1., 2., 3.]   ! A rank-1 length-3 array of default real type
["A", "B"]     ! A rank-1 length-2 array of default character type

integer, parameter :: A = [2, 4]
[1, A, 3]      ! A rank-1 length-4 array of default integer type, with A's elements

integer i
[1, (i, i=2, 5), 6]  ! A rank-1 length-6 array of default integer type with an implied-do
```

В приведенных выше формах все приведенные значения должны быть одного и того же типа и типа. Смешивание типов или параметров типа запрещено. Следующие примеры **недействительны**

```
[1, 2.]      ! INVALID: Mixing integer and default real
[1e0, 2d0]   ! INVALID: Mixing default real and double precision
[1., 2._dp]  ! INVALID: Allowed only if kind `dp` corresponds to default real
["Hello", "Frederick"]  ! INVALID: Different length parameters
```

Для построения массива с использованием разных типов должна быть задана спецификация типа для массива

```
[integer :: 1, 2., 3d0]    ! A default integer array
[real(dp) :: 1, 2, 3._sp]  ! A real(dp) array
[character(len=9) :: "Hello", "Frederick"]  ! A length-2 array of length-9 characters
```

Эта последняя форма для массивов символов особенно удобна, чтобы избежать заполнения пространства, например, альтернативы

```
["Hello    ", "Frederick"]  ! A length-2 array of length-9 characters
```

Размер массива с именем constant может подразумеваться конструктором массива, используемым для установки его значения

```
integer, parameter :: ids(*) = [1, 2, 3, 4]
```

и для параметризованных по длине типов параметр длины может быть принят

```
character(len=*), parameter :: names(*) = [character(3) :: "Me", "You", "Her"]
```

Спецификация типа также требуется при построении массивов нулевой длины. От

```
[ ] ! Not a valid array constructor
```

параметры типа и типа не могут быть определены из несуществующего набора значений. Чтобы создать массив целых чисел по умолчанию нулевой длины:

```
[integer :: ]
```

Конструкторы массива строят только массивы ранга-1. Время от времени, например, при установке значения именованной константы, в выражении также требуются массивы более высокого ранга. Массивы более высокого ранга могут быть взяты из результата `reshape` с построенным массивом ранга-1

```
integer, parameter :: multi_rank_ids(2,2) = RESHAPE([1,2,3,4], shape=[2,2])
```

В конструкторе массива значения массива в порядке элементов с любыми массивами в списке значений являются, как если бы отдельные элементы были предоставлены самим порядком элементов массива. Таким образом, предыдущий пример

```
integer, parameter :: A = [2, 4]
[1, A, 3]      ! A rank-1 length-4 array of default integer type, with A's elements
```

эквивалентно

```
[1, 2, 4, 3]   ! With the array written out in array element order
```

Обычно значения в конструкторе могут быть произвольными выражениями, включая конструкторы вложенных массивов. Для такого конструктора массива, который удовлетворяет определенным условиям, например, как константа или выражение спецификации, ограничения применяются к составным значениям.

Хотя это не конструктор массива, некоторые значения массивов также могут быть удобно созданы с использованием встроенной функции `spread` . Например

```
[(0, i=1,10)]  ! An array with 10 default integers each of value 0
```

также является результатом ссылки на функцию

```
SPREAD(0, 1, 10)
```

Спецификация характера массива: ранг и форма

Атрибут `dimension` для объекта указывает, что этот объект является массивом. В Fortran 2008 существует пять массивов: [1]

- явная форма
- предполагаемая форма
- предполагаемый размер
- отсроченная форма
- подразумеваемая форма

Возьмите три ранга-1 массива [2]

```
integer a, b, c
dimension(5) a    ! Explicit shape (default lower bound 1), extent 5
dimension(:) b    ! Assumed or deferred shape
dimension(*) c    ! Assumed size or implied shape array
```

С их помощью можно видеть, что необходим дальнейший контекст для полного определения характера массива.

Явная форма

Явный массив формы всегда является формой его объявления. Если массив не объявлен как локальный для подпрограммы или конструкции `block` , определяющая форму границы должна быть постоянными выражениями. В других случаях явный массив формы может быть автоматическим объектом, используя экстенты, которые могут различаться при каждом вызове подпрограммы или `block` .

```
subroutine sub(n)
  integer, intent(in) :: n
  integer a(5)   ! A local explicit shape array with constant bound
  integer b(n)   ! A local explicit shape array, automatic object
end subroutine
```

Предполагаемая форма

Предполагаемый массив формы - это фиктивный аргумент без атрибута `allocatable` или `pointer` . Такой массив принимает форму от фактического аргумента, с которым он связан.

```
integer a(5), b(10)
call sub(a)   ! In this call the dummy argument is like x(5)
```

```
call sub(b)   ! In this call the dummy argument is like x(10)

contains

  subroutine sub(x)
    integer x(:)    ! Assumed shape dummy argument
  end subroutine sub

end
```

Когда фиктивный аргумент принял форму, область, ссылающаяся на процедуру, должна иметь явный интерфейс, доступный для этой процедуры.

Предполагаемый размер

Предполагаемый размерный массив - это фиктивный аргумент, размер которого определяется из его фактического аргумента.

```
subroutine sub(x)
  integer x(*)   ! Assumed size array
end subroutine
```

Предполагаемые массивы размеров ведут себя по-разному, чем предполагаемые массивы форм, и эти различия описаны в другом месте.

Отложенная форма

Отложенный массив формы представляет собой массив, который имеет атрибут `allocatable` или `pointer` . Форма такого массива определяется его назначением или назначением указателя.

```
integer, allocatable :: a(:)
integer, pointer :: b(:)
```

Подразумеваемая форма

Массив подразумеваемой формы - это именованная константа, которая принимает форму из выражения, используемого для установления его значения

```
integer, parameter :: a(*) = [1,2,3,4]
```

Последствия этих объявлений массива для фиктивных аргументов должны быть задокументированы в другом месте.

---

[1] Техническая спецификация, расширяющая Fortran 2008, добавляет шестой характер

массива: предполагаемый ранг. Это не рассматривается здесь.

[2] Они могут быть эквивалентно записаны как

```
integer, dimension(5) :: a
integer, dimension(:) :: b
integer, dimension(*) :: c
```

или же

```
integer a(5)
integer b(:)
integer c(*)
```

Целые массивы, элементы массива и массивы

Рассмотрим массив, объявленный как

```
real x(10)
```

Тогда у нас есть три аспекта, представляющие интерес:

1. Весь массив `x` ;
2. Элементы массива, такие как `x(1)` ;
3. Секции массива, такие как `x(2:6)` .

Целые массивы

В большинстве случаев весь массив `x` относится ко всем элементам массива как к единому объекту. Он может отображаться в исполняемых инструкциях, таких как `print *, SUM(x)` , `print *, SIZE(x)` или `x=1` .

Целый массив может ссылаться на массивы, которые не имеют явной формы (например, `x` выше):

```
function f(y)
  real, intent(out) :: y(:)
  real, allocatable :: z(:)

  y = 1.         ! Intrinsic assignment for the whole array
  z = [1., 2.,]  ! Intrinsic assignment for the whole array, invoking allocation
end function
```

Массив предполагаемого размера также может отображаться как целый массив, но только в ограниченных обстоятельствах (для документирования в другом месте).

Элементы массива

Элемент массива называется заданием целочисленных индексов, по одному для каждого ранга массива, обозначающим местоположение во всем массиве:

```
real x(5,2)
x(1,1) = 0.2
x(2,4) = 0.3
```

Элемент массива является скаляром.

Секции массивов

Раздел массива является ссылкой на несколько элементов (возможно, только один) из целого массива, используя синтаксис, включающий двоеточия:

```
real x(5,2)
x(:,1) = 0.         ! Referring to x(1,1), x(2,1), x(3,1), x(4,1) and x(5,1)
x(2,:) = 0.         ! Referring to x(2,1), x(2,2)
x(2:4,1) = 0.       ! Referring to x(2,1), x(3,1) and x(4,1)
x(2:3,1:2) = 0.     ! Referring to x(2,1), x(3,1), x(2,2) and x(3,2)
x(1:1,1) = 0.       ! Referring to x(1,1)
x([1,3,5],2) = 0.   ! Referring to x(1,2), x(3,2) and x(5,2)
```

В приведенной выше форме используется *векторный индекс* . Это зависит от ряда ограничений, помимо других разделов массива.

Каждая секция массива сама является массивом, даже если упоминается только один элемент. То есть `x(1:1,1)` является массивом ранга 1, а `x(1:1,1:1)` является массивом ранга 2.

Секции массива вообще не имеют атрибута всего массива. В частности, где

```
real, allocatable :: x(:)
x = [1,2,3]     ! x is allocated as part of the assignment
x = [1,2,3,4]   ! x is dealloacted then allocated to a new shape in the assignment
```

назначение

```
x(:) = [1,2,3,4,5]   ! This is bad when x isn't the same shape as the right-hand side
```

не допускается: `x(:)` , хотя раздел массива со всеми элементами `x` , не является выделяемым массивом.

```
x(:) = [5,6,7,8]
```

отлично, когда `x` имеет форму правой части.

Массивные элементы массивов

```
type t
   real y(5)
end type t

type(t) x(2)
```

Мы можем также ссылаться на целые массивы, элементы массива и разделы массива в более сложных настройках.

Из вышесказанного `x` является целым массивом. У нас также есть

```
x(1)%y        ! A whole array
x(1)%y(1)     ! An array element
x%y(1)        ! An array section
x(1)%y(:)     ! An array section
x([1,2]%y(1)  ! An array section
x(1)%y(1:1)   ! An array section
```

В таких случаях нам не разрешается иметь более одной части ссылки, состоящей из массива ранга 1. Следующие, например, недопустимы

```
x%y             ! Both the x and y parts are arrays
x(1:1)%y(1:1)   ! Recall that each part is still an array section
```

Операции с массивами

Из-за своих вычислительных целей математические операции над массивами прямолинейны в Fortran.

Сложение и вычитание

Операции над массивами одинаковой формы и размера очень похожи на матричную алгебру. Вместо прогона всех индексов с циклами можно записать сложение (и вычитание):

```
real, dimension(2,3) :: A, B, C
real, dimension(5,6,3) :: D
A    = 3.    ! Assigning single value to the whole array
B    = 5.    ! Equivalent writing for assignment
C    = A + B ! All elements of C now have value 8.
D    = A + B ! Compiler will raise an error. The shapes and dimensions are not the same
```

Также справедливы массивы с нарезкой:

```
integer :: i, j
real, dimension(3,2) :: Mat = 0.
```

```
real, dimension(3)   :: Vec1 = 0., Vec2 = 0., Vec3 = 0.
i = 0
j = 0
do i = 1,3
  do j = 1,2
    Mat(i,j) = i+j
  enddo
enddo
Vec1 = Mat(:,1)
Vec2 = Mat(:,2)
Vec3 = Mat(1:2,1) + Mat(2:3,2)
```

функция

Точно так же большинство внутренних функций могут использоваться неявно, предполагая компонентную работу (хотя это не является систематическим):

```
real, dimension(2) :: A, B
A(1) = 6
A(2) = 44 ! Random values
B    = sin(A) ! Identical to B(1) = sin(6), B(2) = sin(44).
```

Умножение и деление

Необходимо соблюдать осторожность при работе с продуктом и разделением: встроенные операции с использованием символов `*` и `/` имеют элементы:

```
real, dimension(2) :: A, B, C
A(1) = 2
A(2) = 4
B(1) = 1
B(2) = 3
C = A*B ! Returns C(1) = 2*1 and C(2) = 4*3
```

Это не должно ошибаться при работе с матричными операциями (см. Ниже).

Матричные операции

Матричные операции являются внутренними процедурами. Например, матричный продукт массивов предыдущего раздела записывается следующим образом:

```
real, dimension(2,1) :: A, B
real, dimension(1,1) :: C
A(1) = 2
A(2) = 4
B(1) = 1
B(2) = 3
C = matmul(transpose(A),B) ! Returns the scalar product of vectors A and B
```

Сложные операции позволяют инкапсулировать функции, создавая временные массивы.

Хотя это разрешено некоторыми компиляторами и параметрами компиляции, это не рекомендуется. Например, можно записать произведение, включающее транспонирование матрицы:

```
real, dimension(3,3) :: A, B, C
A(:) = 4
B(:) = 5
C = matmul(transpose(A),matmul(B,matmul(A,transpose(B)))) ! Equivalent to A^t.B.A.B^T
```

Расширенные разделы массива: триплеты индексов и векторные индексы

Как упоминалось в другом примере , можно ссылаться на подмножество элементов массива, называемое секцией массива. Из этого примера мы можем иметь

```
real x(10)
x(:)   = 0.
x(2:6) = 1.
x(3:4) = [3., 5.]
```

Однако секции массива могут быть более общими, чем это. Они могут иметь форму индексных триплетов или векторных индексов.

Подстрочные триплеты

Подстрочный тройной принимает вид `[bound1]:[bound2][:stride]` . Например

```
real x(10)
x(1:10) = ...   ! Elements x(1), x(2), ..., x(10)
x(1:) = ...     ! The omitted second bound is equivalent to the upper, same as above
x(:10) = ...    ! The omitted first bound is equivalent to the lower, same as above
x(1:6:2) = ...  ! Elements x(1), x(3), x(5)
x(5:1) = ...    ! No elements: the lower bound is greater than the upper
x(5:1:-1) = ... ! Elements x(5), x(4), x(3), x(2), x(1)
x(::3) = ...    ! Elements x(1), x(4), x(7), x(10), assuming omitted bounds
x(::-3) = ...   ! No elements: the bounds are assumed with the first the lower, negative
stride
```

Когда задан шаг (который не должен быть равен нулю), последовательность элементов начинается с указанной первой привязки. Если шаг положительный (соответственно отрицательный), выбранные элементы следуют за последовательностью, увеличенной (соответственно уменьшенной) по шагу до тех пор, пока последний элемент не будет больше (соответственно меньше), чем вторая граница. Если шаг опущен, он рассматривается как один.

Если первая оценка больше второй, а шаг - положительный, элементы не указаны. Если первая оценка меньше второй, а шаг отрицательный, элементы не указаны.

Следует отметить, что `x(10:1:-1)` не совпадает с `x(1:10:1)` хотя каждый элемент `x` появляется в обоих случаях.

Векторные индексы

Векторным индексом является целочисленный массив ранга-1. Это обозначает последовательность элементов, соответствующих значениям массива.

```
real x(10)
integer i
x([1,6,4]) = ...     ! Elements x(1), x(6), x(4)
x([(i,i=2,4)]) = ... ! Elements x(2), x(3) and x(4)
print*, x([2,5,2])   ! Elements x(2), x(5) and x(2)
```

Раздел массива с векторным индексом ограничен в том, как он может быть использован:

- это может быть не аргумент, связанный с фиктивным аргументом, который определен в процедуре;
- он может не быть целью в инструкции присваивания указателя;
- он не может быть внутренним файлом в инструкции передачи данных.

Кроме того, такой раздел массива может не отображаться в инструкции, которая включает в себя ее определение, когда один и тот же элемент выбирается дважды. Сверху:

```
print*, x([2,5,2])   ! Elements x(2), x(5) and x(2) are printed
x([2,5,2]) = 1.      ! Not permitted: x(2) appears twice in this definition
```

Секции массива более высокого ранга

```
real x(5,2)
print*, x(::2,2:1:-1)  ! Elements x(1,2), x(3,2), x(5,2), x(1,1), x(3,1), x(5,1)
```

Прочитайте Массивы онлайн: https://riptutorial.com/ru/fortran/topic/996/массивы

глава 8: Объектно-ориентированное программирование

Examples

Определение производного типа

В Fortran 2003 появилась поддержка объектно-ориентированного программирования. Эта функция позволяет использовать современные методы программирования. Производные типы определяются с помощью следующей формы:

```
TYPE [[, attr-list] :: ] name [(name-list)]
   [def-stmts]
   [PRIVATE statement or SEQUENCE statement]. . .
   [component-definition]. . .
   [procedure-part]
END TYPE [name]
```

где,

- **attr-list** - список спецификаторов атрибутов
- **name** - имя производного типа данных
- **name-list** - список имен параметров типа, разделенных запятыми
- **def-stmts** - одно или несколько объявлений INTEGER параметров типа, названных в списке имен
- **определение компонента** - один или несколько операторов объявления типа или операторов указателя процедуры, определяющих компонент производного типа
- **procedure-part** - оператор CONTAINS, необязательно сопровождаемый оператором PRIVATE, и один или несколько операторов привязки к процедуре

Пример:

```
type shape
    integer :: color
end type shape
```

Процедуры типа

Чтобы получить поведение класса, тип и связанные с ним процедуры (подпрограмма и функции) должны быть помещены в модуль:

Пример:

```
module MShape
```

```
    implicit none
    private

    type, public :: Shape
    private
        integer :: radius
    contains
        procedure :: set   => shape_set_radius
        procedure :: print => shape_print
    end type Shape

contains
    subroutine shape_set_radius(this, value)
        class(Shape), intent(in out) :: self
        integer, intent(in)          :: value

        self%radius = value
    end subroutine shape_set_radius

    subroutine shape_print(this)
        class(Shape), intent(in) :: self

        print *, 'Shape: r = ', self%radius
    end subroutine shape_print
end module MShape
```

Позже, в коде, мы можем использовать этот класс Shape следующим образом:

```
! declare a variable of type Shape
type(Shape) :: shape

! call the type-bound subroutine
call shape%set(10)
call shape%print
```

Абстрактные производные типы

Расширяемый производный тип может быть *абстрактным*

```
type, abstract :: base_type
end type
```

Такой производный тип никогда не может быть инстанцирован, например,

```
type(base_type) t1
allocate(type(base_type) :: t2)
```

но полиморфный объект может иметь это как объявленный тип

```
class(base_type), allocatable :: t1
```

или же

```
function f(t1)
  class(base_type) t1
end function
```

Абстрактные типы могут содержать компоненты и процедуры с привязкой к типу

```
type, abstract :: base_type
  integer i
contains
  procedure func
  procedure(func_iface), deferred :: def_func
end type
```

Процедура `def_func` - это процедура *отложенного* типа с интерфейсом `func_iface` . Такая отложенная процедура, связанная с типом, должна выполняться каждым расширяющимся типом.

Расширение типа

Производный тип является *расширяемым,* если он не имеет ни атрибута `bind` ни атрибута `sequence` . Такой тип может быть расширен другим типом.

```
module mod

  type base_type
    integer i
  end type base_type

  type, extends(base_type) :: higher_type
    integer j
  end type higher_type

end module mod
```

Полиморфная переменная с объявленным типом `base_type` совместима с типом типа `higher_type` и может иметь это как динамический тип

```
class(base_type), allocatable :: obj
allocate(obj, source=higher_type(1,2))
```

Совместимость типов спускается через цепочку детей, но тип может распространяться только на один другой тип.

Расширяющийся производный тип наследует процедуры привязки типа от родителя, но это может быть переопределено

```
module mod

  type base_type
  contains
    procedure :: sub => sub_base
```

```
  end type base_type

  type, extends(base_type) :: higher_type
  contains
    procedure :: sub => sub_higher
  end type higher_type

contains

  subroutine sub_base(this)
    class(base_type) this
  end subroutine sub_base

  subroutine sub_higher(this)
    class(higher_type) this
  end subroutine sub_higher

end module mod

program prog
  use mod

  class(base_type), allocatable :: obj

  obj = base_type()
  call obj%sub

  obj = higher_type()
  call obj%sub

end program
```

Тип конструктора

Пользовательские конструкторы могут быть созданы для производных типов, используя интерфейс для перегрузки имени типа. Таким образом, аргументы ключевого слова, которые не соответствуют компонентам, могут использоваться при построении объекта такого типа.

```
module ball_mod
  implicit none

  ! only export the derived type, and not any of the
  ! constructors themselves
  private
  public :: ball

  type :: ball_t
     real :: mass
  end type ball_t

  ! Writing an interface overloading 'ball_t' allows us to
  ! overload the type constructor
  interface ball_t
     procedure :: new_ball
  end interface ball_t
```

```
contains

  type(ball_t) function new_ball(heavy)
    logical, intent(in) :: heavy

    if (heavy) then
       new_ball%mass = 100
    else
       new_ball%mass = 1
    end if

  end function new_ball

end module ball_mod

program test
  use ball_mod
  implicit none

  type(ball_t) :: football
  type(ball_t) :: boulder

  ! sets football%mass to 4.5
  football = ball_t(4.5)
  ! calls 'ball_mod::new_ball'
  boulder = ball_t(heavy=.true.)
end program test
```

Это можно использовать для создания более чистого API, чем с использованием отдельных процедур инициализации:

```
subroutine make_heavy_ball(ball)
  type(ball_t), intent(inout) :: ball
  ball%mass = 100
end subroutine make_heavy_ball

...

call make_heavy_ball(boulder)
```

Прочитайте Объектно-ориентированное программирование онлайн: https://riptutorial.com/ru/fortran/topic/2374/объектно-ориентированное-программирование

глава 9: Программные единицы и макет файла

Examples

Программы Fortran

Полная программа Fortran состоит из нескольких отдельных программных модулей. Программные единицы:

- основная программа
- функция или подпрограмма подпрограммы
- модуль или подмодуль
- блок данных блока данных

Основная программа и некоторые подпрограммы (функции или подпрограммы) могут быть предоставлены языком, отличным от Fortran. Например, основная программа C может вызывать функцию, определенную подпрограммой функции Fortran, или основная программа Fortran может вызывать процедуру, определенную C.

Эти программные модули Fortran могут быть представлены отдельными файлами или в одном файле.

Например, мы можем видеть два файла:

prog.f90

```
program main
  use mod
end program main
```

mod.f90

```
module mod
end module mod
```

И компилятор (вызванный правильно) сможет связать основную программу с модулем.

Один файл может содержать множество программных единиц

everything.f90

```
module mod
end module mod
```

```
program prog
  use mod
end program prog

function f()
end function f()
```

В этом случае, однако, следует отметить, что функция `f` по-прежнему является *внешней функцией* в отношении основной программы и модуля. Однако модуль будет доступен основной программе.

Правила области ввода применяются к каждому отдельному программному модулю, а не к файлу, в котором они содержатся. Например, если мы хотим, чтобы в каждой области видимости не было неявного ввода текста, указанный выше файл должен быть записан как

```
module mod
  implicit none
end module mod

program prog
  use mod
  implicit none
end program prog

function f()
  implicit none
  <type> f
end function f
```

Модули и подмодули

Модули документированы в другом месте .

Компиляторы часто генерируют так называемые *файлы модулей* : обычно файл, содержащий

```
module my_module
end module
```

приведет к компилятору файл с именем `my_module.mod` . В таких случаях, чтобы модуль был доступен программному модулю, этот файл модуля должен быть видимым до того, как этот последний программный блок будет обработан.

Внешние процедуры

Внешняя процедура - это та, которая определяется вне другого программного модуля или другими средствами, отличными от Fortran.

Функция, содержащаяся в файле, подобном

```
integer function f()
  implicit none
end function f
```

является внешней функцией.

Для внешних процедур их существование может быть объявлено с помощью блока интерфейса (для указания явного интерфейса)

```
program prog
  implicit none
  interface
    integer function f()
  end interface
end program prog
```

или инструкцией объявления, чтобы дать неявный интерфейс

```
program prog
  implicit none
  integer, external :: f
end program prog
```

или даже

```
program prog
  implicit none
  integer f
  external f
end program prog
```

`external` атрибут не требуется:

```
program prog
  implicit none
  integer i
  integer f
  i = f()   ! f is now an external function
end program prog
```

Блоки данных блоков данных

Блоки данных блоков данных - это программные единицы, которые предоставляют начальные значения для объектов в общих блоках. Они намеренно оставлены без документов здесь и будут представлены в документации по историческим функциям Fortran.

Внутренние подпрограммы

Программный блок, который не является внутренней подпрограммой, может содержать

другие программные единицы, называемые *внутренними подпрограммами* .

```
program prog
  implicit none
contains
  function f()
  end function f
  subroutine g()
  end subroutine g
end program
```

Такая внутренняя подпрограмма имеет ряд особенностей:

- существует ассоциация хозяев между объектами в подпрограмме и внешней программой
- неявные правила набора текста наследуются (`implicit none` действует по `f` выше)
- внутренние подпрограммы имеют явный интерфейс, доступный на хосте

Подпрограммы модуля и внешние подпрограммы могут иметь внутренние подпрограммы, такие как

```
module mod
  implicit none
contains
  function f()
  contains
    subroutine s()
    end subroutine s
  end function f
end module mod
```

Файлы исходного кода

Файл исходного кода представляет собой (обычно) обычный текстовый файл, который обрабатывается компилятором. Файл исходного кода может содержать до одной основной программы и любое количество модулей и внешних подпрограмм. Например, файл исходного кода может содержать следующее

```
module mod1
end module mod1

module mod2
end module mod2

function func1()    ! An external function
end function func1

subroutine sub1()   ! An external subroutine
end subroutine sub1

program prog        ! The main program starts here...
end program prog    ! ... and ends here
```

```
function func2()    ! An external function
end function func2
```

Здесь мы должны напомнить, что, хотя внешние подпрограммы заданы в том же файле, что и модули и основная программа, внешние подпрограммы явно не известны никакому другому компоненту.

В качестве альтернативы отдельные компоненты могут быть распределены по нескольким файлам и даже скомпилированы в разное время. Документация компилятора должна быть прочитана о том, как объединить несколько файлов в одну программу.

Один файл исходного кода может содержать исходный код фиксированной формы или свободной формы: их нельзя смешивать, хотя несколько файлов, которые объединяются во время компиляции, могут иметь разные стили.

Чтобы указать компилятору исходную форму, обычно есть два варианта:

- выбор суффикса имени файла
- использование флагов компилятора

Флаг времени компиляции для указания источника фиксированной или свободной формы можно найти в документации компилятора.

Существенные суффиксы имен файлов также содержатся в документации компилятора, но, как правило, файл с именем `file.f90` берется содержать источник свободной формы, а файл `file.f` - содержать источник фиксированной формы.

Использование суффикса `.f90` для указания источника свободной формы (которое было введено в стандарте Fortran 90) часто заставляет программиста использовать суффикс, чтобы указать языковой стандарт, которому соответствует исходный код. Например, мы можем видеть файлы с `.f03` или `.f08` . Этого обычно следует избегать: большинство источников Fortran 2003 также совместимы с Fortran 77, Fortran 90/5 и Fortran 2008. Кроме того, многие компиляторы автоматически не учитывают такие суффиксы.

Компиляторы также часто предлагают встроенный препроцессор кода (обычно на основе cpp). Опять же, флаг времени компиляции может использоваться для указания того, что препроцессор должен быть запущен до компиляции, но суффикс файла исходного кода также может указывать на такое требование предварительной обработки.

Для файловых систем, чувствительных к `file.F` файл `file.F` часто воспринимается как исходный файл с фиксированной формой, который должен быть предварительно обработан, а `file.F90` - исходный файл свободной формы, который должен быть предварительно обработан. Как и прежде, следует проконсультироваться с документацией компилятора для таких флагов и суффиксов файлов.

Прочитайте Программные единицы и макет файла онлайн:
https://riptutorial.com/ru/fortran/topic/2203/программные-единицы-и-макет-файла

глава 10: Процедуры - Функции и подпрограммы

замечания

Функции и *подпрограммы* в сочетании с *модулями* являются инструментами для разбивки *программы* на единицы. Это делает программу более читаемой и управляемой. Каждый из этих блоков можно рассматривать как часть кода, который в идеале может быть скомпилирован и протестирован изолированно. Основная программа (ы) может вызывать (или вызывать) такие подпрограммы (функции или подпрограммы) для выполнения задачи.

Функции и подпрограммы различаются в следующем смысле:

- **Функции** возвращают один объект и, как правило, не изменяют значения его аргументов (т. Е. Действуют как математическая функция!);
- **Подпрограммы** обычно выполняют более сложную задачу, и они обычно изменяют свои аргументы (если они есть), а также другие переменные (например, те, которые указаны в модуле, который содержит подпрограмму).

Функции и подпрограммы коллективно идут под именем *процедур* . (В дальнейшем мы будем использовать глагол «вызов» как синоним «invoke», даже если технически процедуры, `call` ed, являются `subroutine` s, тогда как `function` s отображается как правая часть задания или в выражениях.)

Examples

Синтаксис функции

Функции могут быть записаны с использованием нескольких типов синтаксиса

```
function name()
  integer name
  name = 42
end function
```

```
integer function name()
  name = 42
end function
```

```
function name() result(res)
  integer res
  res = 42
end function
```

Функции возвращают значения через результат *функции* . Если оператор функции не имеет предложение `result` функции имеет то же имя, что и функция. В `result` функции получается из `result` . В каждом из первых двух примеров выше результат функции дается по `name` ; в третьем - `res` .

Результат функции должен быть определен во время выполнения функции.

Функции позволяют использовать некоторые специальные префиксы.

Чистая функция означает, что эта функция не имеет побочного эффекта:

```
pure real function square(x)
  real, intent(in) :: x
  square = x * x
end function
```

Элементная функция определяется как скалярный оператор, но она может быть вызвана с массивом как фактический аргумент, и в этом случае функция будет применяться по типу элемента. Если не указан `impure` префикс (введенный в Fortran 2008), *элементарная* функция также является *чистой* функцией.

```
elemental real function square(x)
  real, intent(in) :: x
  square = x * x
end function
```

Оператор возврата

Оператор `return` может использоваться для выхода из функции и подпрограммы. В отличие от многих других языков программирования он не используется для установки возвращаемого значения.

```
real function f(x)
  real, intent(in) :: x
  integer :: i

  f = x

  do i = 1, 10

    f = sqrt(f) - 1.0

    if (f < 0) then
      f = -1000.
      return
    end if

  end do
end function
```

Эта функция выполняет итеративное вычисление. Если значение `f` становится

отрицательным, функция возвращает значение -1000.

Рекурсивные процедуры

В функциях Fortran и подпрограммах должны быть явно объявлены как *рекурсивные* , если они должны называть себя снова, прямо или косвенно. Таким образом, рекурсивная реализация серии Фибоначчи может выглядеть так:

```
recursive function fibonacci(term) result(fibo)
  integer, intent(in) :: term
  integer :: fibo

  if (term <= 1) then
    fibo = 1
  else
    fibo = fibonacci(term-1) + fibonacci(term-2)
  end if

end function fibonacci
```

Другим примером является возможность вычисления факториала:

```
recursive function factorial(n)  result(f)
  integer :: f
  integer, intent(in) :: n

  if(n == 0) then
    f = 1
  else
    f = n * f(n-1)
  end if
end function factorial
```

Для функции, непосредственно рекурсивно ссылающейся на себя, ее определение должно использовать суффикс `result` . Функция не может быть как `recursive` и `elemental` .

Предположение о ложных аргументах

Атрибут `intent` фиктивного аргумента в подпрограмме или функции объявляет о своем намеренном использовании. Синтаксис является либо одним из

```
intent(IN)
intent(OUT)
intent(INOUT)
```

Например, рассмотрим эту функцию:

```
real function f(x)
  real, intent(IN) :: x

  f = x*x
end function
```

`intent(IN)` указывает, что фиктивный аргумент `x` (не указатель) `x` никогда не может быть определен или не определен во всей функции или ее инициализации. Если у фиктивного аргумента указателя есть `intent(IN)` атрибута `intent(IN)` , это относится к его ассоциации.

`intent(OUT)` для аргумента фиктивного указателя без указателя означает, что фиктивный аргумент становится неопределенным при вызове подпрограммы (за исключением любых компонентов производного типа с инициализацией по умолчанию) и должен быть установлен во время выполнения. Фактический аргумент, переданный как фиктивный аргумент, должен быть определяемым: передача именованной или литеральной константы или выражения не допускается.

Аналогично предыдущему, если указатель фиктивный аргумент является `intent(OUT)` статус ассоциации указателя становится неопределенным. Фактический аргумент здесь должен быть переменной указателя.

`intent(INOUT)` указывает, что фактический аргумент является определяемым и подходит как для передачи, так и для возврата данных из процедуры.

Наконец, фиктивный аргумент может быть без атрибута `intent` . Такой фиктивный аргумент используется ограниченным фактическим аргументом.

Например, рассмотрим

```
integer :: i = 0
call sub(i, .TRUE.)
call sub(1, .FALSE.)

end

subroutine sub(i, update)
  integer i
  logical, intent(in) :: update
  if (update) i = i+1
end subroutine
```

Аргумент `i` может иметь никакого атрибута `intent` который разрешает оба вызова подпрограммы основной программы.

Ссылка на процедуру

Для полезности функции или подпрограммы она должна быть указана. Подпрограмма ссылается в заявлении о `call`

```
call sub(...)
```

и функцию внутри выражения. В отличие от многих других языков выражение не формирует полный оператор, поэтому ссылка на функцию часто встречается в инструкции присваивания или используется каким-то другим способом:

```
x = func(...)
y = 1 + 2*func(...)
```

Существует три способа указания ссылки на процедуру:

- как имя указателя процедуры или процедуры
- компонент процедуры объекта производного типа
- имя привязки процедуры привязки типа

Первое можно увидеть как

```
procedure(), pointer :: sub_ptr=>sub
call sub()   ! With no argument list the parentheses are optional
call sub_ptr()
end

subroutine sub()
end subroutine
```

и последние два

```
module mod
  type t
    procedure(sub), pointer, nopass :: sub_ptr=>sub
  contains
    procedure, nopass :: sub
  end type

contains

  subroutine sub()
  end subroutine

end module

use mod
type(t) x
call x%sub_ptr()   ! Procedure component
call x%sub()       ! Binding name

end
```

Для процедуры с фиктивными аргументами ссылка требует соответствующих *фактических* аргументов, хотя необязательные фиктивные аргументы могут не задаваться.

Рассмотрим подпрограмму

```
subroutine sub(a, b, c)
  integer a, b
  integer, optional :: c
end subroutine
```

На это могут ссылаться следующие два способа

```
call sub(1, 2, 3)   ! Passing to the optional dummy c
call sub(1, 2)      ! Not passing to the optional dummy c
```

Это так называемая *позиционная* ссылка: фактические аргументы связаны с позицией в списках аргументов. Здесь макет `a` связан с `1` , `b` с `2` и `c` (если указано) с `3` .

Альтернативно, привязка *ключевых слов* может использоваться, когда процедура имеет явный интерфейс, доступный

```
call sub(a=1, b=2, c=3)
call sub(a=1, b=2)
```

который является тем же самым, что и выше.

Однако с ключевыми словами фактические аргументы могут предлагаться в любом порядке

```
call sub(b=2, c=3, a=1)
call sub(b=2, a=1)
```

Позиционные и ссылки на ключевые слова могут использоваться как

```
call sub(1, c=3, b=2)
```

пока задано ключевое слово для каждого аргумента, следующего за первым появлением ключевого слова

```
call sub(b=2, 1, 3)  ! Not valid: all keywords must be specified
```

Значение ссылки на ключевые слова особенно заметно, когда имеется несколько необязательных аргументов фиктивного типа, как показано ниже, если в определении подпрограммы выше `b` также были необязательными

```
call sub(1, c=3)  ! Optional b is not passed
```

Списки аргументов для процедур с привязкой к типу или указатели процедур компонентов с переданным аргументом рассматриваются отдельно.

Прочитайте Процедуры - Функции и подпрограммы онлайн:
https://riptutorial.com/ru/fortran/topic/1106/процедуры---функции-и-подпрограммы

глава 11: Расширения исходного файла (.f, .f90, .f95, ...) и то, как они связаны с компилятором.

Вступление

Файлы Fortran подпадают под множество расширений, и каждый из них имеет отдельный смысл. Они определяют версию выпуска Fortran, стиль форматирования кода и использование директив препроцессора, аналогичных C-языку программирования.

Examples

Расширения и значения

Ниже перечислены некоторые из распространенных расширений, используемых в исходных файлах Fortran и функциях, над которыми они могут работать.

Нижний регистр f в расширении

Эти файлы не имеют функций директив препроцессора, аналогичных C-программированию. Их можно напрямую скомпилировать для создания объектных файлов. например: .f, .for, .f95

Верхний регистр F в расширении

Эти файлы имеют функции директив препроцессора, аналогичные языку C-программирования. Препроцессоры либо определяются в файлах, либо используют файлы заголовков C / C ++, как и те, и другие. Эти файлы должны быть предварительно обработаны, чтобы получить файлы расширения нижнего регистра, которые можно использовать для компиляции. например: .F, .FOR, .F95

.f, .for, .f77, .ftn

Они используются для файлов Fortran, которые используют **формат фиксированного стиля** и, таким образом, используют версию **Fortran 77** . Поскольку они являются расширением нижнего регистра, они не могут иметь директив препроцессора.

.F, .FOR, .F77, .FTN

Они используются для файлов Fortran, которые используют **формат фиксированного стиля** и, таким образом, используют версию **Fortran 77** . Поскольку они являются расширениями верхнего регистра, они могут иметь директивы препроцессора, и поэтому их

необходимо предварительно обработать, чтобы получить файлы расширения нижнего регистра.

.f90, .f95, .f03, .f08 Они используются для файлов Fortran, которые используют **формат Free style** и, таким образом, используют более поздние версии Fortran. Версия выпуска находится в названии.

- f90 - Fortran 90
- f95 - Fortran 95
- f03 - Fortran 2003
- f08 - Fortran 2008

Поскольку они являются расширением нижнего регистра, они не могут иметь директив препроцессора.

.F90, .F95, .F03, .F08 Они используются для файлов Fortran, которые используют **формат Free style** и, таким образом, используют более поздние версии Fortran. Версия выпуска находится в названии.

- F90 - Фортран 90
- F95 - Fortran 95
- F03 - Fortran 2003
- F08 - Fortran 2008

Поскольку они являются расширениями верхнего регистра, у них есть директивы препроцессора, и поэтому они должны быть предварительно обработаны для получения файлов расширения нижнего регистра.

Прочитайте Расширения исходного файла (.f, .f90, .f95, ...) и то, как они связаны с компилятором. онлайн: https://riptutorial.com/ru/fortran/topic/10265/расширения-исходного-файла---f---f90---f95-------и-то--как-они-связаны-с-компилятором-

глава 12: Современные альтернативы историческим особенностям

Examples

Неявные типы переменных

Когда Fortran была первоначально разработана, память была в отличной форме. Имена переменных и процедур могут содержать не более 6 символов, а переменные часто *неявно печатаются* . Это означает, что первая буква имени переменной определяет ее тип.

- переменные, начинающиеся с i, j, ..., n, являются `integer`
- все остальные (a, b, ..., h, o, p, ..., z) являются `real`

Возможны такие программы, как Fortran:

```
program badbadnotgood
  j = 4
  key = 5 ! only the first letter determines the type
  x = 3.142
  print*, "j = ", j, "key = ", key, "x = ", x
end program badbadnotgood
```

Вы даже можете определить свои собственные неявные правила с `implicit` выражением:

```
! all variables are real by default
implicit real (a-z)
```

или же

```
! variables starting with x, y, z are complex
! variables starting with c, s are character with length of 4 bytes
! and all other letters have their default implicit type
implicit complex (x,y,z), character*4 (c,s)
```

Неявная типизация больше не считается лучшей практикой. Очень легко сделать ошибку, используя неявное типирование, поскольку опечатки могут остаться незамеченными, например

```
program oops
  real :: somelongandcomplicatedname

  ...

  call expensive_subroutine(somelongandcomplEcatedname)
end program oops
```

Эта программа будет успешно работать и делать не то.

Чтобы отключить неявное типирование, можно использовать `implicit none` оператор.

```
program much_better
  implicit none
  integer :: j = 4
  real :: x = 3.142
  print*, "j = ", j, "x = ", x
end program much_better
```

Если мы не использовали `implicit none` в программе `oops` выше, компилятор заметил бы сразу, и возникает ошибка.

Арифметическое утверждение if

Арифметический оператор `if` позволяет использовать три ветви в зависимости от результата арифметического выражения

```
if (arith_expr) label1, label2, label3
```

Этот оператор `if` переносит поток управления на одну из меток кода. Если результатом `arith_expr` является отрицательная `label1` , то если результат равен нулю, используется `label2` , и если результат положительный, применяется последняя `label3` . Арифметика, `if` требует всех трех меток, но допускает повторное использование меток, поэтому это утверждение можно упростить до двух ветвей `if` .

Примеры:

```
if (N * N - N / 2) 130, 140, 130

if (X) 100, 110, 120
```

Теперь эта функция устарела с той же функциональностью, предлагаемых в , `if` заявление и `if-else` конструкция. Например, фрагмент

```
    if (X) 100, 110, 120
100 print*, "Negative"
    goto 200
110 print*, "Zero"
    goto 200
120 print*, "Positive"
200 continue
```

может быть записана как конструкция `if-else`

```
if (X<0) then
  print*, "Negative"
```

```
else if (X==0) then
  print*, "Zero"
else
  print*, "Positive"
end if
```

Замена оператора `if`

```
    if (X) 100, 100, 200
100 print *, "Negative or zero"
200 continue
```

может быть

```
if (X<=0) print*, "Negative or zero"
```

Неблокирующие конструкторы DO

Конструкция без блока `do` выглядит так:

```
    integer i
    do 100, i=1, 5
100 print *, i
```

То есть, если заявленный оператор терминации не является оператором `continue` . Существуют различные ограничения на утверждение, которое может использоваться как заявление о завершении, и все это, как правило, очень запутанно.

Такая неблокирующая конструкция может быть переписана в виде блока в виде

```
    integer i
    do 100 i=1,5
      print *, i
100 continue
```

или лучше, используя `end do` терминации заявление,

```
integer i
do i=1,5
  print *, i
end do
```

Альтернативный возврат

Альтернативный возврат - это средство контроля потока выполнения при возврате из подпрограммы. Он часто используется как форма обработки ошибок:

```
real x
```

```
call sub(x, 1, *100, *200)
print*, "Success:", x
stop

100 print*, "Negative input value"
stop

200 print*, "Input value too large"
stop

end

subroutine sub(x, i, *, *)
  real, intent(out) :: x
  integer, intent(in) :: i
  if (i<0) return 1
  if (i>10) return 2
  x = i
end subroutine
```

Альтернативный возврат отмечен аргументами `*` в списке аргументов подпрограммы.

В `call` заявлении выше `*100` и `*200` см операторов , помеченных `100` и `200` соответственно.

В самой подпрограмме операторы `return` соответствующие альтернативному возврату, имеют число. Это число не является возвращаемым значением, но обозначает предоставленную метку, которой выполняется исполнение при возврате. В этом случае `return 1` пропускает выполнение в оператор с меткой `100` и `return 2` прохождение выполнения в оператор, помеченный знаком `200` . Унаследованный оператор `return` или завершение выполнения подпрограммы без оператора `return` , выполнение пассессии сразу после оператора вызова.

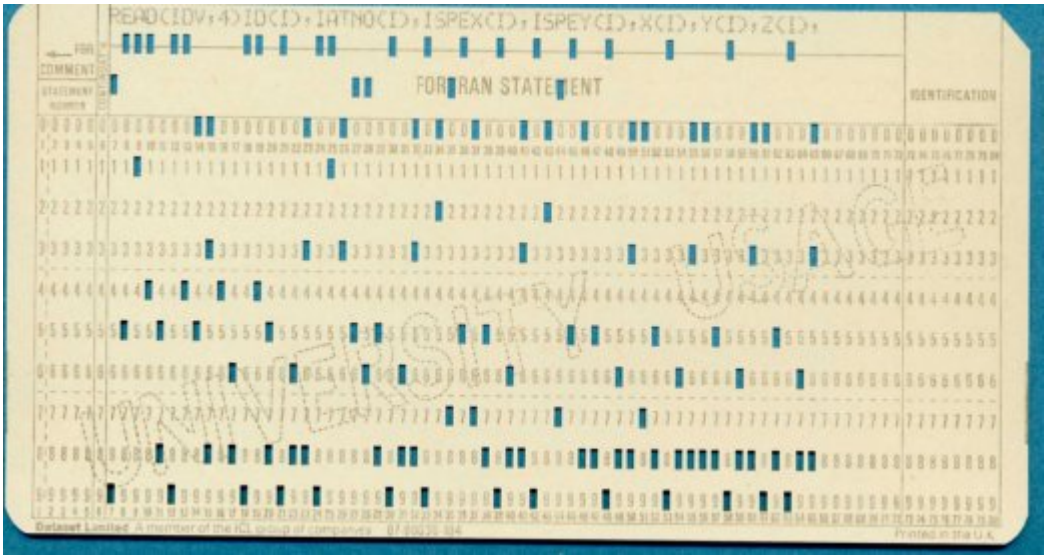
Синтаксис альтернативного возврата сильно отличается от других форм ассоциации аргументов, и средство вводит управление потоком вопреки современным вкусам. Более приятное управление потоком можно управлять с возвратом целочисленного кода состояния.

```
real x
integer status

call sub(x, 1, status)
select case (status)
case (0)
  print*, "Success:", x
case (1)
  print*, "Negative input value"
case (2)
  print*, "Input value too large"
end select

end

subroutine sub(x, i, status)
  real, intent(out) :: x
  integer, intent(in) :: i
```

```
  integer, intent(out) :: status

  status = 0

  if (i<0) then
    status = 1
  else if (i>10)
    status = 2
  else
    x = i
  end if

end subroutine
```

Фиксированная исходная форма

Первоначально Fortran был разработан для формата фиксированного формата на основе 80-битной перфокарты:



Да. Это строка собственного кода автора.

Они были созданы на перфокартной машине, примерно так:

Изображения - это оригинальная фотография автора

Формат, как показано на иллюстрированной образце карты, имел первые пять столбцов, зарезервированных для ярлыков операторов. Первый столбец использовался для обозначения комментариев буквой **C.** Шестой столбец использовался для обозначения продолжения оператора (путем вставки любого символа, отличного от нуля «0»). Последние 8 столбцов были использованы для идентификации и секвенирования карты, что было очень полезно, если вы уронили свою колоду карт на пол! Кодировка символов для перфокарт имела только ограниченный набор символов и была только в верхнем регистре. В результате программы Fortran выглядели следующим образом:

```
       DIMENSION A(10)                                                  00000001
C THIS IS A COMMENT STATEMENT TO EXPLAIN THIS EXAMPLE PROGRAM           00000002
       WRITE (6,100)                                                    00000003
 100   FORMAT(169HTHIS IS A RATHER LONG STRING BEING OUTPUT WHICH GOES OVE00000004
      1R MORE THAN ONE LINE, AND USES THE STATEMENT CONTINUATION MARKER IN00000005
      2COLUMN 6, AND ALSO USES HOLLERITH STRING FORMAT)                  00000006
       STOP                                                             00000007
       END                                                              00000008
```

Косвенный символ также игнорировался везде, за исключением внутренней константы символа *Холлерита* (как показано выше). Это означало, что пробелы могут возникать внутри зарезервированных слов и констант или полностью пропущены. Это имело побочный эффект некоторых довольно вводящих в заблуждение заявлений, таких как:

```
       DO 1 I = 1.0
```

является присвоением переменной `DO1I` тогда как:

```
       DO1I = 1,0
```

фактически является циклом `DO` для переменной `I`

Современный Fortran теперь не требует этой фиксированной формы ввода и разрешает свободную форму с использованием любых столбцов. Комментарии теперь обозначаются а `!` который также может быть добавлен к строке оператора. Пробелы теперь не разрешены нигде и должны использоваться как разделители, как и на большинстве других языков. Вышеупомянутая программа может быть написана в современном Fortran как:

```
! This is a comment statement to explain this example program
Print *,"THIS IS A RATHER LONG STRING BEING OUTPUT WHICH no longer GOES OVER MORE THAN ONE
LINE, AND does not need to USE THE STATEMENT CONTINUATION MARKER IN COLUMN 6, or the HOLLERITH
STRING FORMAT"
```

Хотя продолжение старого стиля больше не используется, приведенный выше пример иллюстрирует, что очень длинные утверждения все равно будут возникать. Современный Фортран использует символ `&` в конце и в начале продолжения. Например, мы могли бы написать выше в более читаемой форме:

```
! This is a comment statement to explain this example program
Print *,"THIS IS A RATHER LONG STRING BEING OUTPUT WHICH still &
         &GOES OVER MORE THAN ONE LINE, AND does need to USE THE STATEMENT &
         &CONTINUATION notation"
```

Общие блоки

В ранних формах Fortran единственным механизмом создания глобального хранилища переменных, видимым из подпрограмм и функций, является использование механизма блока `COMMON` . Это разрешает последовательности переменных быть именами и совместно использоваться.

В дополнение к названным общим блокам может также быть пустой (неназванный) общий блок.

Можно было бы объявить пустой общий блок

```
common i, j
```

тогда как именованные блок- `variables` могут быть объявлены как

```
common /variables/ i, j
```

В качестве полного примера мы могли бы представить хранилище кучи, которое используется подпрограммами, которые могут добавлять и удалять значения:

```
      PROGRAM STACKING
      COMMON /HEAP/ ICOUNT, ISTACK(1023)
      ICOUNT = 0
      READ *, IVAL
      CALL PUSH(IVAL)
      CALL POP(IVAL)
      END

      SUBROUTINE PUSH(IVAL)
      COMMON /HEAP/ ICOUNT, ISTACK(1023)
      ICOUNT = ICOUNT + 1
      ISTACK(ICOUNT) = IVAL
      RETURN
      END

      SUBROUTINE POP(IVAL)
      COMMON /HEAP/ ICOUNT, ISTACK(1023)
      IVAL = ISTACK(ICOUNT)
      ICOUNT = ICOUNT - 1
      RETURN
      END
```

Общие утверждения могут использоваться для неявного объявления типа переменной и указания атрибута `dimension` . Такое поведение часто является достаточным источником

путаницы. Кроме того, подразумеваемая ассоциация хранения и требования для повторных определений между программными единицами используют использование обычных блоков, подверженных ошибкам.

Наконец, общие блоки очень ограничены в объектах, которые они содержат. Например, массив в общем блоке должен иметь явный размер; выделяемые объекты могут не возникнуть; производные типы не должны иметь инициализацию по умолчанию.

В современном Fortran это совместное использование переменных может быть связано с использованием модулей . Вышеприведенный пример может быть записан как:

```
module heap
  implicit none
  ! In Fortran 2008 all module variables are implicitly saved
  integer, save :: count = 0
  integer, save :: stack(1023)
end module heap

program stacking
  implicit none
  integer val
  read *, val
  call push(val)
  call pop(val)

contains
  subroutine push(val)
    use heap, only : count, stack
    integer val
    count = count + 1
    stack(count) = val
  end subroutine push

  subroutine pop(val)
    use heap, only : count, stack
    integer val
    val = stack(count)
    count = count - 1
  end subroutine pop
end program stacking
```

Именованные и пустые общие блоки имеют несколько другое поведение. Отметить:

- объекты в названных общих блоках могут быть определены изначально; объекты в пробе не должны быть
- объекты в пустых общих блоках ведут себя так, как будто общий блок имеет атрибут `save` ; объекты в именованных общих блоках без атрибута `save` могут стать неопределенными, когда блок не входит в область активного программного блока

Эта последняя точка может быть противопоставлена поведению модульных переменных в современном коде. Все переменные модуля в Fortran 2008 неявно сохраняются и не становятся неопределенными, когда модуль выходит из области видимости. До того, как

переменные модуля Fortran 2008, как и переменные в именованных общих блоках, также станут неопределенными, когда модуль выходит из сферы действия.

Назначенный GOTO

Назначенный GOTO использует целочисленную переменную, которой присваивается метка оператора с помощью оператора ASSIGN.

```
100 CONTINUE

...

ASSIGN 100 TO ILABEL

...


GOTO ILABEL
```

Назначенный GOTO устаревает в Fortran 90 и удаляется в Fortran 95 и более поздних версиях. Этого можно избежать в современном коде, используя процедуры, внутренние процедуры, указатели на процедуры и другие функции.

Вычисление GOTO

Вычисленный GOTO позволяет разветвлять программу в соответствии со значением целочисленного выражения.

```
GOTO (label_1, label_2,... label_n) scalar-integer-expression
```

Если `scalar-integer-expression` `label_1` `scalar-integer-expression` равно 1, программа продолжается на метке оператора `label_1` , если она равна 2, она переходит к `label_2` и т. Д. Если он меньше `1` или больше, чем `n` программа продолжит следующую строку.

Пример:

```
ivar = 2

...

GOTO (10, 20, 30, 40) ivar
```

перейдем к метке оператора 20.

Эта форма `goto` устарела в Fortran 95 и позже, будучи замененной конструкцией `select case` .

Назначенные спецификаторы формата

До Fortran 95 можно было использовать назначенные форматы для ввода или вывода. Рассматривать

```
integer i, fmt
read *, i

assign 100 to fmt
if (i<100000) assign 200 to fmt

print fmt, i

100 format ("This is a big number", I10)
200 format ("This is a small number", I6)

end
```

`assign` оператор присваивает метку заявление к целочисленной переменной. Эта целочисленная переменная позже будет использоваться в качестве спецификатора формата в заявлении `print` .

Такое назначение спецификатора формата было удалено в Fortran 95. Вместо этого более современный код может использовать некоторую другую форму управления потоком выполнения

```
integer i
read *, i

if (i<100000) then
  print 100, i
else
  print 200, i
end if

100 format ("This is a big number", I10)
200 format ("This is a small number", I6)

end
```

или символьная переменная может использоваться как спецификатор формата

```
character(29), target :: big_fmt='("This is a big number", I10)'
character(30), target :: small_fmt='("This is a small number", I6)'
character(:), pointer :: fmt

integer i
read *, i

fmt=>big_fmt
if (i<100000) fmt=>small_fmt

print fmt, i

end
```

Функции выписки

Рассмотрите программу

```
implicit none
integer f, i
f(i)=i

print *, f(1)
end
```

Здесь `f` - функция оператора. Он имеет целочисленный тип результата, принимающий один целочисленный аргумент фиктивного аргумента. [1]

Такая функция оператора существует в пределах области, в которой она определена. В частности, он имеет доступ к переменным и именованным константам, доступным в этой области.

Тем не менее, функции оператора подвержены многим ограничениям и потенциально запутывают (смотря на случайный взгляд, как оператор присваивания элемента массива). Важными ограничениями являются:

- результат функции и фиктивные аргументы должны быть скалярными
- фиктивные аргументы находятся в той же области, что и функция
- функции оператора не имеют локальных переменных
- функции оператора не могут передаваться как фактические аргументы

Основные преимущества функций оператора повторяются внутренними функциями

```
implicit none

print *, f(1)

contains

  integer function f(i)
    integer i
    f = i
  end function

end
```

Внутренние функции не подпадают под ограничения, упомянутые выше, хотя, возможно, стоит отметить, что внутренняя подпрограмма может не содержать дополнительной внутренней подпрограммы (но она может содержать функцию оператора).

Внутренние функции имеют свою собственную область действия, но также имеют доступную ассоциацию хостов.

---

[1] В реальных старых примерах кода было бы необычно видеть, что фиктивные аргументы функции оператора неявно типизированы, даже если результат имеет явный тип.

Прочитайте Современные альтернативы историческим особенностям онлайн: https://riptutorial.com/ru/fortran/topic/2103/современные-альтернативы-историческим-особенностям

глава 13: Типы данных

Examples

Внутренние типы

Ниже перечислены типы данных, *присущие* Fortran:

```
integer
real
character
complex
logical
```

`integer` , `real` и `complex` - числовые типы.

`character` - это тип, используемый для хранения символьных строк.

`logical` используется для хранения двоичных значений `.true.` или `.false.` ,

Все числовые и логические внутренние типы параметризуются с использованием видов.

```
integer(kind=specific_kind)
```

или просто

```
integer(specific_kind)
```

где `specific_kind` - целое число с константой.

Символьные переменные, а также имеющие параметр вида, также имеют параметр длины:

```
character char
```

объявляет `char` как переменную символа length-1 типа по умолчанию, тогда как

```
character(len=len) name
```

объявляет `name` символьной переменной по умолчанию и длиной `len` . Можно также указать вид

```
character(len=len, kind=specific_kind) name
character(kind=specific_kind) char
```

объявляет `name` символом доброго `kind` и длины `len` . `char` является длиной 1 символ доброго

`kind` .

Альтернативно, устаревшая форма для объявления символа

```
character*len  name
```

может быть замечен в более старом коде, объявляя, что `name` имеет длину `len` и тип символа по умолчанию.

Объявление переменной внутреннего типа может иметь форму выше, но также может использовать форму `type(...)` :

```
integer i
real x
double precision y
```

эквивалентно (но очень предпочтительно)

```
type(integer) i
type(real) x
type(double precision) y
```

Производные типы данных

Определите новый тип, `mytype` :

```
type :: mytype
  integer :: int
  real    :: float
end type mytype
```

Объявить переменную типа *mytype* :

```
type(mytype) :: foo
```

К компонентам производного типа можно обращаться с помощью оператора `%` [1] :

```
foo%int = 4
foo%float = 3.142
```

Функция Fortran 2003 (еще не реализована всеми компиляторами) позволяет определять параметризованные типы данных:

```
type, public :: matrix(rows, cols, k)
  integer, len :: rows, cols
  integer, kind :: k = kind(0.0)
  real(kind = k), dimension(rows, cols) :: values
```

```
end type matrix
```

`matrix` производного типа имеет три параметра типа, которые перечислены в круглых скобках, следующих за именем типа (это `rows` , `cols` и `k`). В объявлении каждого параметра типа необходимо указать, являются ли они типами типа (`kind`) или length (`len`).

Параметры типа типа, такие как внутренние типы, должны быть постоянными выражениями, тогда как параметры типа длины, такие как длина внутренней символьной переменной, могут изменяться во время выполнения.

Обратите внимание, что параметр `k` имеет значение по умолчанию, поэтому оно может быть предоставлено или опущено при объявлении переменной типа `matrix` следующим образом

```
type (matrix (55, 65, kind=double)) :: b, c ! default parameter provided
type (matrix (rows=40, cols=50)     :: m    ! default parameter omitted
```

Имя производного типа не может быть `doubleprecision` или идентичным любому внутреннему типу.

---

1. Многие люди задаются вопросом, почему Fortran использует `%` как оператор доступа к компонентам, а не более общий `.` , Это потому, что `.` уже принимается синтаксисом оператора, т `.not.` е `.not.` , `.and.` , `.my_own_operator.` ,

Точность чисел с плавающей запятой

Числа с плавающей точкой типа `real` не могут иметь никакого реального значения. Они могут представлять действительные числа до определенного количества десятичных цифр.

FORTRAN 77 гарантирует два типа с плавающей точкой, а более поздние стандарты гарантируют по меньшей мере два реальных типа. Реальные переменные могут быть объявлены как

```
real x
double precision y
```

`x` здесь является реальным по умолчанию, и `y` является реальным видом с большей десятичной точностью, чем `x` . В Fortran 2008 десятичная точность `y` составляет не менее 10, а ее десятичный показатель - не менее 37.

```
 real, parameter             :: single = 1.12345678901234567890
 double precision, parameter :: double = 1.123456789012345678890d0

 print *, single
```

```
 print *, double
```

печать

```
   1.12345684
   1.1234567890123457
```

в обычных компиляторах, использующих конфигурацию по умолчанию.

Обратите внимание на `d0` в константе двойной точности. Для обозначения двойной точности используется реальный литерал, содержащий `d` вместо `e` для обозначения экспоненты.

```
! Default single precision constant
1.23e45
! Double precision constant
1.23d45
```

Fortran 90 ввел параметризованные `real` типы, используя виды. Тип реального типа - это целое число с константой или константой:

```
real(kind=real_kind) :: x
```

или просто

```
real(real_kind) :: x
```

Этот оператор объявляет, что `x` имеет тип `real` с определенной точностью в зависимости от значения `real_kind` .

Литералы с плавающей запятой могут быть объявлены в определенном виде с использованием суффикса

```
1.23456e78_real_kind
```

Точное значение `real_kind` не стандартизировано и отличается от компилятора компилятором. Чтобы узнать тип какой-либо реальной переменной или константы, можно использовать функцию `kind()` :

```
print *, kind(1.0), kind(1.d0)
```

обычно печатает

```
4 8
```

или же

```
1 2
```

в зависимости от компилятора.

Добрые номера можно задать несколькими способами:

1. Одиночная (по умолчанию) и двойная точность:

```
integer, parameter :: single_kind = kind(1.)
integer, parameter :: double_kind = kind(1.d0)
```

2. Используя внутреннюю функцию `selected_real_kind([p, r])` укажите требуемую десятичную точность. Возвращаемый вид имеет точность не менее `p` цифр и позволяет экспоненту не менее `r` .

```
integer, parameter :: single_kind = selected_real_kind( p=6, r=37 )
integer, parameter :: double_kind = selected_real_kind( p=15, r=200 )
```

3. Начиная с Fortran 2003, предварительно определенные константы доступны через встроенный модуль `ISO_C_Binding` чтобы гарантировать, что реальные виды взаимодействуют с типами `float` , `double` или `long_double` сопроводительного компилятора C:

```
use ISO_C_Binding

integer, parameter :: single_kind = c_float
integer, parameter :: double_kind = c_double
integer, parameter :: long_kind = c_long_double
```

4. Начиная с Fortran 2008, предварительно определенные константы доступны через встроенный модуль `ISO_Fortran_env` . Эти константы предоставляют реальные виды с определенным размером хранилища в битах

```
use ISO_Fortran_env

integer, parameter :: single_kind = real32
integer, parameter :: double_kind = real64
integer, parameter :: quadruple_kind = real128
```

Если определенный тип недоступен в компиляторе, значение, возвращаемое `selected_real_kind()` или значением целочисленной константы, равно `-1` .

Предполагаемые и отложенные параметры типа длины

Переменные типа символа или производного типа с параметром длины могут иметь параметр длины, *предполагаемый* или *отложенный* . Символьная переменная `name`

```
character(len=len) name
```

имеет длину `len` во время исполнения. И наоборот, спецификатор длины может быть

```
character(len=*) ...  ! Assumed length
```

или же

```
character(len=:) ...  ! Deferred length
```

Предполагаемые переменные длины принимают их длину от другого объекта.

В функции

```
function f(dummy_name)
  character(len=*) dummy_name
end function f
```

фиктивный аргумент `dummy_name` имеет длину фактического аргумента.

Именованная константа `const_name` в

```
character(len=*), parameter :: const_name = 'Name from which length is assumed'
```

имеет длину, заданную постоянным выражением в правой части.

Параметры отложенного типа длины могут меняться во время выполнения. Переменная с отложенной длиной должна иметь либо атрибут `allocatable` либо `pointer`

```
character(len=:), allocatable :: alloc_name
character(len=:), pointer :: ptr_name
```

Такая длина переменной может быть задана любым из следующих способов

```
allocate(character(len=5) :: alloc_name, ptr_name)
alloc_name = 'Name'         ! Using allocation on intrinsic assignment
ptr_name => another_name    ! For given target
```

Для производных типов с параметризацией длины синтаксис аналогичен

```
  type t(len)
    integer, len :: len
    integer i(len)
  end type t

  type(t(:)), allocatable :: t1
  type(t(5)) t2

  call sub(t2)
  allocate(type(t(5)) :: t1)

contains
```

```
  subroutine sub(t2)
    type(t(*)), intent(out) :: t2
  end subroutine sub

end
```

Литеральные константы

В модулях программы часто используются литеральные константы. Они охватывают очевидные случаи, такие как

```
print *, "Hello", 1, 1.0
```

За исключением одного случая, каждая константа литерала является скаляром, который имеет тип, параметры типа и значение, заданные синтаксисом.

Целочисленные литеральные константы имеют вид

```
1
-1
-1_1   ! For valid kind parameter 1
1_ik   ! For the named constant ik being a valid kind paramter
```

Реальные литеральные константы имеют вид

```
1.0    ! Default real
1e0    ! Default real using exponent format
1._1   ! Real with kind parameter 1 (if valid)
1.0_sp ! Real with kind paramter named constant sp
1d0    ! Double precision real using exponent format
1e0_dp ! Real with kind named constant dp using exponent format
```

Сложные литеральные константы имеют вид

```
(1, 1.)       ! Complex with integer and real components, literal constants
(real, imag)  ! Complex with named constants as components
```

Если действительные и мнимые компоненты являются целыми, комплексная константа литерала по умолчанию является сложной, а целочисленные компоненты преобразуются в реальную по умолчанию. Если один компонент вещественный, то параметром вида сложной константы литерала является действительный (и целочисленный компонент преобразуется в этот реальный вид). Если оба компонента являются реальными, сложная константа литерала имеет вид реального с наибольшей точностью.

Логические константы

```
.TRUE.     ! Default kind, with true value
.FALSE.    ! Default kind, with false value
```

```
.TRUE._1   ! Of kind 1 (if valid), with true value
.TRUE._lk  ! Of kind named constant lk (if valid), with true value
```

Значения символов буквально отличаются по понятию, поскольку спецификатор вида предшествует значению

```
"Hello"       ! Character value of default kind
'Hello'       ! Character value of default kind
ck_"Hello"    ! Character value of kind ck
"'Bye"        ! Default kind character with a '
'''Bye'       ! Default kind character with a '
""            ! A zero-length character of default kind
```

Как было предложено выше, символьные константы должны быть отделены апострофами или кавычками, а маркер начала и конца должен совпадать. Буквенные апострофы могут быть включены в рамки ограничителей кавычек или появляться в два раза. То же самое для кавычек.

Константы BOZ отличаются от приведенных выше, поскольку они указывают только значение: у них нет параметра типа или типа. Константа BOZ является битовой диаграммой и указывается как

```
B'00000'    ! A binary bit pattern
B"01010001" ! A binary bit pattern
O'012517'   ! An octal bit pattern
O"1267671"  ! An octal bit pattern
Z'0A4F'     ! A hexadecimal bit pattern
Z"FFFFFF"   ! A hexadecimal bit pattern
```

Литеральные константы BOZ ограничены там, где они могут появляться: как константы в операторах `data` и выбор внутренних процедур.

Доступ к символьным подстрокам

Для объекта символа

```
character(len=5), parameter :: greeting = "Hello"
```

подстрока может ссылаться на синтаксис

```
greeting(2:4)  ! "ell"
```

Для доступа к одной букве недостаточно писать

```
greeting(1)    ! This isn't the letter "H"
```

но

```
greeting(1:1)  ! This is "H"
```

Для массива символов

```
character(len=5), parameter :: greeting(2) = ["Hello", "Yo!  "]
```

у нас есть подстрочный доступ, как

```
greeting(1)(2:4)  ! "ell"
```

но мы не можем ссылаться на несмежные символы

```
greeting(:)(2:4)  ! The parent string here is an array
```

Мы можем получить доступ к подстрокам литеральных констант

```
"Hello"(2:4)
```

Часть символьной переменной также может быть определена с использованием подстроки в качестве переменной. Например

```
integer :: i=1
character :: filename = 'file000.txt'

filename(9:11) = 'dat'
write(filename(5:7), '(I3.3)') i
```

Доступ к сложным компонентам

Сложный объект

```
complex, parameter :: x = (1., 4.)
```

имеет действительную часть `1.` и сложную часть `4.` .. Мы можем получить доступ к этим отдельным компонентам как

```
real(x)  ! The real component
aimag(x) ! The complex component
x%re     ! The real component
y%im     ! The complex component
```

Форма `x%.` . является новой для Fortran 2008 и широко не поддерживается в компиляторах. Однако эту форму можно использовать для непосредственного задания отдельных компонентов комплексной переменной

```
complex y
```

```
y%re = 0.
y%im = 1.
```

Декларация и атрибуты

Всюду по темам и примерам здесь мы увидим много деклараций переменных, функций и т. Д.

Как и их имя, объекты данных могут иметь *атрибуты* . В этом разделе рассматриваются заявления, подобные

```
integer, parameter :: single_kind = kind(1.)
```

который дает объекту `single_kind` атрибут `parameter` (что делает его именованной константой).

Есть много других атрибутов, например

- `target`
- `pointer`
- `optional`
- `save`

Атрибуты могут быть указаны с так называемыми *инструкциями спецификации атрибута*

```
integer i    ! i is an integer (of default kind)...
pointer i    ! ... with the POINTER attribute...
optional i   ! ... and the OPTIONAL attribute
```

Однако обычно считается, что лучше избегать использования этих спецификаций спецификации атрибута. Для ясности атрибуты могут быть указаны как часть одной декларации

```
integer, pointer, optional :: i
```

Это также уменьшает соблазн использовать неявное типирование.

В большинстве случаев в этой документации по Фортрану это утверждение является предпочтительным.

Прочитайте Типы данных онлайн: https://riptutorial.com/ru/fortran/topic/939/типы-данных

глава 14: Явные и неявные интерфейсы

Examples

Внутренние / модульные подпрограммы и явные интерфейсы

Подпрограмма (которая определяет *процедуру*) может быть либо `subroutine` либо `function` ; он называется *внутренней подпрограммой,* если он вызывается или вызывается из той же самой `program` или *подпрограммы,* которая `contains` ее, следующим образом

```
program my_program

  ! declarations
  ! executable statements,
  ! among which an invocation to
  ! internal procedure(s),
  call my_sub(arg1,arg2,...)
  fx = my_fun(xx1,xx2,...)

contains

  subroutine my_sub(a1,a2,...)
    ! declarations
    ! executable statements
  end subroutine my_sub

  function my_fun(x1,x2,...) result(f)
    ! declarations
    ! executable statements
  end function my_fun

end program my_program
```

В этом случае компилятор будет знать все о любой внутренней процедуре, поскольку он относится к программному блоку в целом. В частности, он «увидит» `interface` процедуры, то есть

- является ли это `function` или `subroutine` ,
- которые являются именами и свойствами аргументов `a1` , `a2` , `x1` , `x2` , ...,
- которые являются свойствами *результата* `f` (в случае `function`).

Будучи известным интерфейсом, компилятор может проверить, `xx1` ли фактические аргументы (`arg1` , `arg2` , `xx1` , `xx2` , `fx` , ...) в соответствие с фиктивными аргументами (`a1` , `a2` , `x1` , `x2` , `f` , .. .).

В этом случае мы говорим, что интерфейс *явный* .

Подпрограмма называется *подпрограммой модуля,* когда она вызывается оператором в самом содержащем модуле,

```
module my_mod

  ! declarations

contains

  subroutine my_mod_sub(b1,b2,...)
    ! declarations
    ! executable statements
    r = my_mod_fun(b1,b2,...)
  end subroutine my_sub

  function my_mod_fun(y1,y2,...) result(g)
    ! declarations
    ! executable statements
  end function my_fun

end module my_mod
```

или инструкцией в другом программном модуле, `use` этот модуль,

```
program my_prog

  use my_mod

  call my_mod_sub(...)

end program my_prog
```

Как и в предыдущей ситуации, компилятор будет знать все о подпрограмме, и поэтому мы говорим, что интерфейс *явный* .

Внешние подпрограммы и неявные интерфейсы

Подпрограмма называется *внешней,* если она не содержится в основной программе, ни в модуле, ни в подпрограмме antoher. В частности, его можно определить с помощью языка программирования, отличного от Fortran.

Когда вызывается внешняя подпрограмма, компилятор не может получить доступ к его коду, поэтому вся информация, разрешенная компилятору, неявно содержится в вызывающем операторе вызывающей программы и в типе свойства аргументов acutal, а не фиктивных аргументов (чья декларация неизвестна компилятору). В этом случае мы говорим, что интерфейс *неявный* .

`external` оператор может использоваться для указания того, что имя процедуры относится к внешней процедуре,

```
external external_name_list
```

но даже при этом интерфейс остается неявным.

`interface`

блок может использоваться для указания интерфейса внешней процедуры,

```
interface
  interface_body
end interface
```

где `interface_body` обычно является точной копией заголовка процедуры, за которой следует объявление всех его аргументов и, если это функция, результата.

Например, для функции `WindSpeed`

```
real function WindSpeed(u, v)
  real, intent(in) :: u, v
  WindSpeed = sqrt(u*u + v*v)
end function WindSpeed
```

Вы можете написать следующий интерфейс

```
interface
  real function WindSpeed(u, v)
    real, intent(in) :: u, v
  end function WindSpeed
end interface
```

Прочитайте Явные и неявные интерфейсы онлайн: https://riptutorial.com/ru/fortran/topic/2882/явные-и-неявные-интерфейсы

кредиты

S. No	Главы	Contributors
1	Начало работы с Fortran	Alexander Vogt, Community, Enrico Maria De Angelis, Gilles, haraldkl, High Performance Mark, Ingve, innoSPG, milancurcic, packet0, RamenChef, Serenity, Vladimir F, Yossarian
2	C совместимость	Serenity, Yossarian
3	I / O	AL-P, Ed Smith, francescalus, Kyle Kanos, TTT
4	Внутренние процедуры	francescalus
5	Использование модулей	Alexander Vogt, Enrico Maria De Angelis, francescalus, Serenity, Vladimir F
6	Контроль выполнения	Enrico Maria De Angelis, francescalus, haraldkl, ptev, Serenity, syscreat, TTT, Vladimir F
7	Массивы	Enrico Maria De Angelis, francescalus, G.Clavier, Gilles, Serenity, TTT, Vladimir F, Yossarian
8	Объектно-ориентированное программирование	Enrico Maria De Angelis, francescalus, syscreat, Yossarian
9	Программные единицы и макет файла	agentp, francescalus, haraldkl, trblnc
10	Процедуры - Функции и подпрограммы	Alexander Vogt, Enrico Maria De Angelis, francescalus, haraldkl , Serenity, Vladimir F, Yossarian
11	Расширения исходного файла (. f, .f90, .f95, ...) и то, как они связаны с компилятором.	Arun
12	Современные альтернативы	Brian Tompsett - , d_1999, Enrico Maria De Angelis, francescalus, Serenity, TTT, Vladimir F, Yossarian

	историческим особенностям	
13	Типы данных	Alexander Vogt, Enrico Maria De Angelis, francescalus, Vladimir F, Yossarian
14	Явные и неявные интерфейсы	Enrico Maria De Angelis, Serenity, Vladimir F