Programowanie w języku Fortran

dr inż. Maciej Woźniak [1]

[1]Katedra Informatyki, Wydział Informatyki, Elektroniki i Telekomunikacji,
Akademia Górniczo-Hutnicza, Kraków, Polska

Boolean

Zmiennymi logicznymi w Fortran-ie są **logical**. Mogą przyjmować wartości **.TRUE.** i **.FALSE.**

```
logical :: x, y
x = .TRUE.
y = .FALSE.
```

Block

Blok zamknięty słowem kluczowym **block** pozwala ograniczyć zasięg zmiennej.

```
block
   integer :: x
   x = -18
   write(*,*) x
end block
! tu zmienna x jest niewidoczna
```

Associate

Blok zamknięty słowem kluczowym **associate** pozwala utworzyć zmienną w locie. Nowo utworzona zmienna jest migawką ustalonej wartości. Warto zwrócic uwagę, że zmiennej **z** nigdzie jawnie nie deklarujemy. Zmienna **z** jest też niemodyfikowalna.

```
integer :: x,y
x = 5
associate (z => x)
   write(*,*) z ! wypisze 5
   x = 10
   write(*,*) z ! wypisze 5
end associate

associate (z => x**y)
end associate
```

If

Instrukcje warunkowe zamknięte są w bloki ograniczone słowami kluczowymi **if, then, else, end if**

```
if (x .EQ. 5) then
   y = 10
else if (x .LE. 10) then
   y = -10
else
   y = -100
end if

if (x .EQ. 5) y = -100
```

Case

Instrukcje case zamknięte są w bloki ograniczone słowami kluczowymi **select case, case, end select**

```
select case (x)
  case (10)
    write(*,*) “10”
  case (15)
    write(*,*) “15”
  case default
    write(*,*) “unknown case”
end select
```

Exit i cycle

Exit wyrzuca nas za najbliższy **end**, chyba, że wypecyfikowano inaczej w kodzie programu. **Cycle** kończy daną iterację pętli.

```
do
  cycle
  exit
end do
```

```
block mblock
  do
    do
      exit mblock
    end do
  end do
end block mblock
```

Moduły

Moduły są odpowiednikiem paczek. Moduł może, ale nie musi dołączać inne moduły. Moduły są dołączane "kaskadowo" - jeśli załączony moduł załącza inne moduły, one również będą załączone.

```
module nazwa
  ! załadowanie innych modułów
  use mpi ! ten moduł będzie dołączony razem z modułem 'nazwa'
  implicit none
  ! deklaracja zmiennych i typów
  integer :: x
contains
  ! deklaracja funkcji i rutyn
  subroutine count(z)
    z = 10.d0
  end subroutine
end module nazwa
```

Moduł prywatny

Moduł może zawierać elementy prywatne i publiczne. W przypadku modułu prywatnego, wszystko co nie zostanie jawnie określone jako publiczne będzie prywatne.
Domyślnie moduły są publiczne. Moduł publiczny jest przeciwieństwem modułu prywatnego.

```
module prv
  implicit none
  private
  integer :: x, y, z
  integer, public :: i_public
  integer, public, parameter :: i_p_public = 100
contains
end module prv
```

Zmienne chronione

Oprócz zmiennych prywatnych i w pełni publicznych (do których ma się pełny dostęp z poza modułu) istnieją też zmienne chronione, które można z poza modułu tylko odczytać. Słowem kluczowym jest tutaj **protected**.

```
module prv
  implicit none
  private
  integer, public, protected :: i_public
contains
  subroutine modify_i_public(x)
  end subroutine
end module prv
```

Ograniczony import

Można importować tylko wybrane fragmenty modułów za pomocą **only**.

```
use prv, only : i_public
```

UWAGA - po słowie kluczowym **only** występuje tylko jeden dwukropek.
Importy można robić też per funkcja/subrutyna. Nie trzeba importować modułu do całego modułu

```
module fun
  implicit none
contains
  subroutine use_some(x)
    use prv, only : i_public
  end subroutine
end module fun
```

Typy złożone

Typy (struktury) tworzy się za pomocą słowa kluczowego **type**

```
module prv
  implicit none
  type, public :: unsecure
    character (1000) :: user
    character (1000) :: password
  end type unsecure

  type, public :: secure
    integer :: user_hash
    integer :: password_hash
  end type secure
end module prv
```

Tworzenie zmiennych złożonych

Proste utworzenie a następnie wypełnienie typu. Do pól dobieramy się przez **%**

```
type (unsecure) :: un
un%user = "macwozni"
un%password = "ComicSans_to_dobra_czcionka"
```

Wypełnienie w momencie deklaracji zmiennej

```
type (secure (user_hash=10, password_hash=100)) :: se
type (unsecure (user="user", password="mypass")) :: unse1
type (unsecure ("user", "mypass")) :: unse2
```

Ostatni przypadek wymaga zachowania kolejności pól wynikającej z definicji struktury.

Czytanie parametrów z linii poleceń

Do zmiennych podanych z linii poleceń można dobrać się w dowolnym miejscu kodu. Funkcja **command_argument_count** zwraca ilość argumentów. Subroutyna **get_command_argument(number, value, length, status)** odczytuje dany argument do buffora **value**

```
integer:: args_count,first_argument,length,status
character(100):: value
args_count = command_argument_count()
call get_command_argument(1,value,length,status)
read(value,*) first_argument
```

STDERR

Fortran 2003 wprowadził standardowe wyjście na błąd - **ERROR_UNIT**. Definicja znajduje się w module **ISO_FORTRAN_ENV**.

```
use ISO_FORTRAN_ENV, ONLY: ERROR_UNIT
write (ERROR_UNIT, *) "We have problem"
```

Interface

W Fortran-ie można definiować interfejsy do routine i funkcji. Zawierają one tylko definicję bez wykonywalnego bloku. Mogą być stosowane przy łączeniu Fortran-a z C

```
interface
function f(x) result(y)
  real :: x,y
end function f
end interface
```

Procedury generyczne

Rozważmy dwie subroutiny swap (dla integer i real)

```
subroutine swap_real(x,y)
  real :: x,y, tmp
  tmp = x
  x = y
  y = tmp
end subroutine


subroutine swap_int(x,y)
  integer :: x,y, tmp
  tmp = x
  x = y
  y = tmp
end subroutine
```

Procedury generyczne

Możemy sprawić, żeby obie powyższe subroutiny były widoczne pod wspólnym **swap**

```
public :: swap
private :: swap_int, swap_real
interface swap
   procedure swap_int, swap_real
end interface
```

Procedury elementowe

Procedury elementowe przyjmują skalar (nie tablicę) jako argumenty. Zawołane na tablicy będą działać element po elemencie.

```
elemental subroutine swap_int(x,y)
  integer :: x,y, tmp
  tmp = x
  x = y
  y = tmp
end subroutine
```

Można je również łączyć w procedury generyczne.

Własne opertory

W Fortran-ie można definiować własne operatory oraz rozszerzać już istniejące. Np. + po rozszerzeniu może działać również na wartościach logicznych.

```
public :: operator(+)
private :: l_p_l
interface operator(+)
  procedure l_p_l
end interface

function l_p_l(x,y) result (sum)
  logical :: x,y,sum
  sum = x .OR. y
end function
```

Własne opertory

Nazwy własnych operatorów zaczynają się i kończą kropką, np. **.BLOW.**

```
interface operator(.BLOW.)
   procedure blow
end interface
```

Podmoduły

Podmoduły pozwalają na podział modułów na mniejsze kawałki. Ułatwiają też rekompilację po zmanie kawałka modułu. W przypadku gdy dwa moduły potrzebują siebie nawzajem (co jest niedozwolone - patrz analogiczne "circular dependency" w C++) sprawę rozwiązują podmoduły. Podmoduły mogą być kaskadowe (mogą mieć swoje podmoduły).

module (line)
end module line

submodule (line) sub_line
end submodule

Optymalnie umieszczać (pod)moduły w osobnych plikach.

Na następnych wykładach

Na najbliższych wykładach

- wypisywanie i jego formatowanie
- klasy
- testy jednostkowe
- programowanie funkcyjne
- **(pół)-automatyczna równoległość**