Programowanie w języku Fortran

dr inż. Maciej Woźniak [1]

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

Komentarze i kontynuacja linijki

Komentarz zaczyna się od znaku **!** podczas, gdy chcąc kontynuować daną linijkę poniżej używamy na końcu kontynuowanej **&**

```
integer :: array1a(11)
! to jest komentarz
integer :: array1b(10) ! to również jest komentarzem
integer :: &
   array2(10,20) ! tu natomiast mieliśmy doczynienia z kontynuacją
```

Wstęp do tablic

Domyślnie tablica jest indeksowana od 1 do iSIZE włącznie. Tablice mogą mieć dowolą ilość wymiarów.

```
integer :: array1(iSIZE)
integer :: array2(iSIZE,iSIZE)
integer :: array3(iSIZEX,iSIZEY,iSIZEZ)
```

Indeksacja tablic

Tablice można indeksować inaczej. Poniżej dwa tożsame indeksacje tablicy.

```
integer :: array1(iSIZE)
integer :: array2(1:iSIZE)
```

Można również stosować inne sposoby indeksowania dla takich samych rozmiarów tablic.

```
integer :: array3(0:iSIZE-1)
integer :: array4(10:iSIZE+9)
integer :: array5(-1:iSIZE-2)
```

Slice tablicy

Dozwolone jest wyciąganie dowolnego typu podtablicy.

```
integer :: array1(100)
integer :: array2(10)
   array2 = array1(1:10)
   array2 = array1(91:100)
```

Slice tablicy

Dozwolone jest wyciąganie dowolnego typu slice (pod)tablicy.

```
integer :: array1(10,10)
integer :: array2(100,100)
integer :: array3(10)
   array3 = array1(:,1)
   array3 = array1(3,:)
   array3 = array2(91:100,4)
   array2(5:14,1) = array3
```

Operacje na tablicy

Dozwolone jest wykonywanie operacji na całych (pod)tablicach

```
integer :: array1(10,10)
integer :: array2(100,100)
integer :: array3(10)
   array1 = 0
   array2 = 2
   array1 = array1 - 5
   array1 = array2(1:10,21:30) * 5
   array3 = array1(:,1) - array2(1:10,1)
   write(*,*) array1
```

Operacje na tablicy

"Mnożenie całych tablic"

```
integer :: array1(10)
integer :: array2(10)
integer :: i
```

Poniższa pętla

```
do  i=1,10
   array1(i) = array1(i) * array2(i)
end do
```

Jest tożsama z

```
array1 = array1 * array2
```

Operacje na tablicy

"Porównywanie całych tablic"

```
integer :: array1(10)
integer :: array2(10)
integer :: i
```

Poniższa pętla

```
do  i=1,10
   write(*,*) array1(i) .eq. array2(i)
end do
```

Jest tożsama z

```
write(*,*) array1 .eq. array2
```

Wynikiem porównania dwóch tablic jest tablica wartości logicznych.

Tablice deklarowane “w locie” i reshape

Język pozwala na deklarowanie tablic “w locie” oraz zmianę typu tablicy na wielowymiarową za pomocą **reshape**. Pomocne może być polecenie **shape**.

```
integer :: array1(3)
integer :: array2(10)
integer :: array3(2,5)
  array1 = (/ 1 , 2 , 3 /)
  array1 = [ 1 , 2 , 3 ]
  array3 = reshape(array2, shape(array3))
  array3 = reshape(array2, (/2,5/))
  array3 = reshape(array2, [2,5])
```

Tablice w 'routinach'

Dozwolone jest dynamiczne podanie rozmiaru tworzonej na stosie tablicy

```
subroutine dynamic(isize)
implicit none
integer, intent(in) :: isize
integer :: array(isize)
end subroutine
```

Uwaga - stos ma ograniczony rozmiar. Teoretycznie można go zwiększyć, jednak w ograniczonym zakresie. Większe tablice należy alokować "ręcznie".

Alokowanie tablic

Tablice (i nie tylko) można "ręcznie" alokować za pomocą **allocate** oraz dealokować za pomocą **deallocate**.
Słowo kluczowe **allocated** sprawdza, czy tablica była zaalokowana.

```
integer, allocatable :: array1a(:)
integer, allocatable, dimension(:) :: array1b
integer, allocatable, dimension(:,:,:) :: array3
  allocate (array1a(iSIZE))
  allocate (array1b(iSIZE))
  allocate (array3(iSIZEX,iSIZEY,iSIZEZ))
  if (allocated(array1a)) deallocate(array1a)
  if (allocated(array1b)) deallocate(array1b)
  if (allocated(array3)) deallocate(array3)
```

Tablice **ZAWSZE** alokowane są jako jeden blok pamięci.

Sprawdzanie rozmiaru tablic

Rozmiar tablicy można sprawdzić w trakcie działania aplikacji za pomocą **size**.

```
integer, allocatable :: array1a(:)
integer :: array1b(10)
integer :: array2(10,20)
  allocate (array1a(11))
  write(*,*) size(array1a) ! 11
  write(*,*) size(array1b) ! 10
  write(*,*) size(array2) ! 200
  write(*,*) size(array2(:,:)) ! 200
  write(*,*) size(array2(1,:)) ! 20
  write(*,*) size(array2(:,1)) ! 10
```

Liczby zespolone

Język wspiera liczby zespolone oraz operacje na nich.

```
complex :: c1
complex (kind=4) :: c2
  c1 = 1.d0
  c2 = (-1.d0, -1.d0)
  c2 = (-1.d0, -1.d0) * (-1.d0, -1.d0)
  c2 = c2 + c1
```

Zawieranie się typów danych

Typy danych o mniejszej dokładności zawierają sie w typach o dokładności większej.

- **integer (kind=4)** $\in$ **integer (kind=8)**
- **integer** $\in$ **real**
- **real** $\in$ **complex**

Jawne rzutowanie

Typy danych można rzutować jawnie i niejawnie.

integer (kind=8) :: a
integer (kind=4) :: b
real (kind=4) :: c
complex (kind=4) :: d

a = b
b = a
b = **int**(a, **kind**=4)
c = **real**(d)
c = **aimag**(d)

Parametry

Język wspiera "parametry" czyli niemodyfikowalne typy danych o odgórnie znanych wartościach. Słowem kluczowym przy deklaracji jest **parameter**

```
integer (kind=8), parameter :: a = 10
integer (kind=8) :: b = 8
```

Wskazuje ono kompilatorowi, że może inaczej przechowywać w pamięci daną zmienną oraz nie pozwoli programiście jej zmodyfikować.

Na następnych wykładach

Na najbliższych wykładach

- jeszcze więcej o typach danych
- wypisywanie i jego formatowanie
- instrukcje warunkowe
- makra i kompilacja warunkowa
- parametry z linii poleceń
- moduły i podmoduły
- klasy i struktury