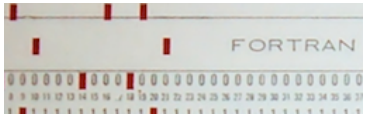
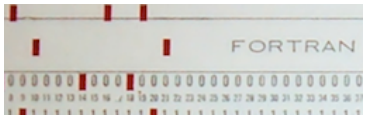


Programowanie współbieżne... (3a)

Andrzej Baran **2010/11**

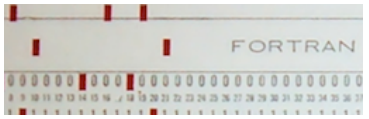
LINK: http://kft.umcs.lublin.pl/baran/prir/index.html

Fortran 95 – rozszerzenia



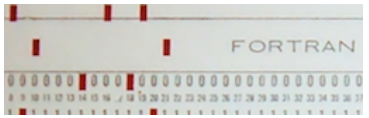
- M. Metcalf: Fortran 90/95/HPF Information File
  http://www.fortran.com/metcalf.htm
- M. Metcalf, Fortran 90 Tutorial
  http://wwwasdoc.web.cern.ch/wwwasdoc/WWW/f90/f90.html
- **M. Metcalf's Fortran 90 CNL Articles**
  http://wwwasdoc.web.cern.ch/wwwasdoc/f90.html
- Fortran Tutorials: http://www.fortran.com/
- Fortran 90/95 Explained, M. Metcalf and J. Reid, (Oxford, 1996)
- Fortran 95/2003 Explained), M. Metcalf, J. Reid, M. Cohen (Oxford University Press, 2004)

Treść



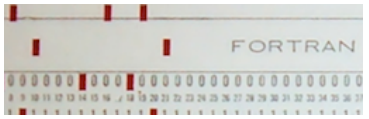
- Podprogramy
- Bloki *interface*
- Overloading – przeciążanie
- Rekurencja
- Pointer – wskaźnik, wskaz
- Association
- Dyrektywy *public* i *private*

Podprogramy



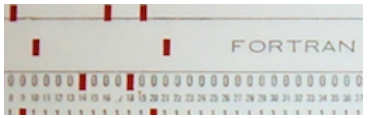

```
Subroutine nowa
   real x, y
   :

CONTAINS
   ! Podprogram=program w programie
   subroutine inna
      real y
      y=x+1_d
   end subroutine inna  ! wymagane!

End subroutine nowa
```

Bloki interface




```
MODULE arytmetyka_odcinkowa
   TYPE odcinek
     real x, y
   END TYPE odcinek
   INTERFACE OPERATOR(+)
     MODULE PROCEDURE dodaj_odcinki
   END INTERFACE
CONTAINS
   FUNCTION dodaj_odcinki(a,b)
      TYPE(odcinek), INTENT(IN) :: a, b
      TYPE(odcinek) dodaj odcinki
      dodaj_odcinki%x=a%x + b%x
      dodaj_odcinki%y=a%y + b%y
   end FUNCTION dodaj_odcinki  ! wymagane
END MODULE arytmetyka_odcinkowa
```

Bloki interface



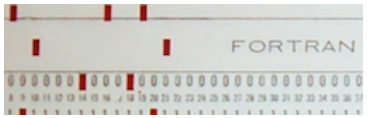
Wywołania podprogramów wewnętrznych lub modułowych odbywa się poprzez interfejsy widoczne (jawne) dla kompilatora.

Wywołania podprogramów zewnętrznych jest zazwyczaj niejawne (kompilator domyślnie przyjmuje sposób wywołania)

Podanie (w module lub w programie) nagłówka procedury, definicji argumentów oraz instrukcji END dla danego podprogramu określa interfejs – sposób jego wywoływania

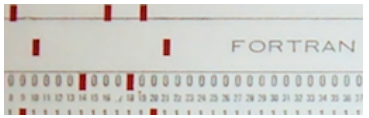
Interfejsy obowiązują w przypadku gdy podprogram posiada argumenty opcjonalne lub „z kluczem” (a=7), zmiennych typu POINTER i TARGET.

Interface: przykład




```
REAL FUNCTION minimum(a, b, func)
! Zwraca minimum funkcji func(x) na odcinku a, b
    real, intent(in) :: a, b
   INTERFACE
    REAL FUNCTION func(x)
       REAL, INTENT(IN) :: x
    END FUNTION func
   END INTERFACE
   REAL f, x
   :
   f = func(x)   ! Wywołanie funkcji użytkownika
   :
END FUNCTION minimum
```

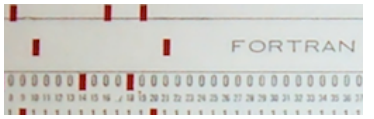
Overloading - przeciążanie




```
INTERFACE gamma
  FUNCTION sgamma(x)  !niska precyzja
      REAL (SELECTED_REAL_KIND(6)) :: dgamma, x
  END FUNCTION sgamma
  FUNCTION dgamma(x)  !wysoka precyzja
      REAL (SELECTED_REAL_KIND(12)) :: dgamma, x
  END FUNCTION sgamma
END INTERFACE
```

Funkcja `gamma` jest `przeciążona`, tzn. że można jej użyć na dwa sposoby, z dwoma różnymi typami argumentu x (niska i wysoka precyzja obliczeń), używając jednak tylko nazwy `gamma` – nazwy rodzajowej (*generic*). Nie musimy pamiętać nazw `sgamma`, `dgamma` itd. Kompilator, na podstawie interfejsu, sam rozstrzyga, której funkcji ma używać w danym przypadku.

Rekurencja niejawna

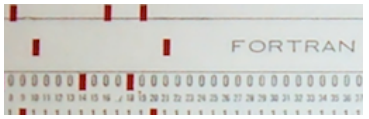



```
RECURSIVE FUNCTION integrate(f, granice)
! Całka f(x) w granicach granice
   REAL integrate
   INTERFACE
      FUNCTION f(x)
         REAL f, x
      END FUNCTION f
   END INTERFACE
   REAL, DIMENSION(2), INTENT(IN) :: granice
END FUNCTION integrate
```

Całkę po prostokącie z funkcji f(x,y) liczymy następująco:

```
FUNCTION fy(y)
  USE MODFUNC    ! Moduł MODFUNC zawiera funkcję f
  REAL fy, y
  yval = y; fy = integrate(f, granice_x)
END
```

Rekurencja jawna

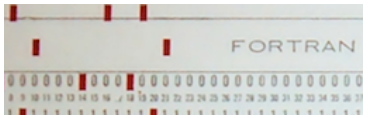



```
RECURSIVE FUNCTION factorial(n) RESULT(res)
! Silnia całkowitej liczby n
   INTEGER n, res
   IF (n == 0 .OR. N == 1) THEN
      res = 1
   ELSE
      res = n*factorial(n-1)
   END IF
END
```

Jest to przykład zastosowania rekurencji jawnej.

W przypadku przykładu z poprzedniej strony (procedura `integrate`) mieliśmy rekurencję pośrednią albo ukrytą.

Pointer - wskaz



Pointery wskazuja zmienne i struktury, obiekty.

```
REAL, POINTER :: zmienna
```

Pamięć na obiekt jest rezerwowana w momencie alokacji. Obiektem może być pointer.

```
ALLOCATE (zmienna)
```

Przykład listy łączonej:

```
TYPE element
    REAL wartosc
    INTEGER indeks
    TYPE(element), POINTER :: nastepny
END TYPE element
```

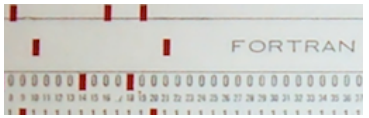
Początek listy

```
   TYPE(element), POINTER :: lista
```

Po alokacji, adresy pierwszych dwóch elementów listy są następujące

```
   lista%wartosc       lista%nastepny%wartosc
   lista%indeks        lista%nastepny%indeks
   lista%nastepny      lista%nastepny%nastepny
```

Association - związania



Pointer może być w stanie nieokreślonym (początkowym), związanym (z obiektem; associated) lub niezwiązanym:

```
DEALLOCATE (obiekt1, obiekt2)   ! Po zwolnieniu pamięci
NULLIFY zmienna                 ! Po „wyzerowaniu”
```

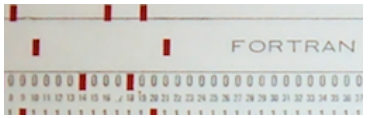
Z pomocą wewnętrznej funkcji `ASSOCIATED` można zbadać stan pointera:

```
IF (ASSOCIATED(pointer)) THEN
   :
```

lub stan związania pointera i określonej tarczy (target, pointer do obiektu):

```
IF (ASSOCIATED(pointer, target)) THEN
   :
```

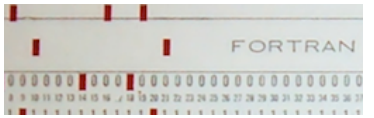
Związania



Poniższy fragment programu drukuje status związania wskazów.

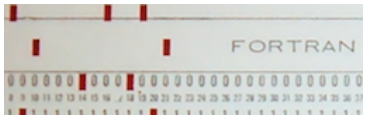
```
REAL, POINTER :: p, q          ! Związek nieokreślony
REAL, TARGET :: t = 3.0
p => t                         ! p wskazuje na t
q => t                         ! q również wskazuje t
PRINT *, ”Po p => t, ASSOCIATED(p) = “, ASSOCIATED(p) ! .true.
PRINT *, "ASSOCIATED(p, q) = ", ASSOCIATED(p, q)      ! .true.
NULLIFY(p)
PRINT *, ”Po NULLIFY(p), ASSOCIATED(p) = ", ASSOCIATED(p) !.false.
PRINT *, "ASSOCIATED(p, q) = ”, ASSOCIATTED(p, q)       ! .false.
...
p => t                         ! p wskazuje t
NULLIFY(p, q)
```

Przykłady użycia „pointer”




```
REAL, TARGET  :: b(10,10), c(10,10), r(10), s(10, z(10)
REAL, POINTER :: a(:,:), x(:), y(:)
INTEGER mult
  :
DO mult = 1, 2
   IF (mult == 1) THEN
      y => r                ! Nie ma kopiowania danych
      a => c
      x => z
   ELSE
      y => s                ! Nie ma kopiowania danych
      a => b
      x => r
   END IF
   y = MATMUL(a, x)         ! Zwyczajne obliczenia
END DO
```

Pointer = wynik funkcji



Jeśli funkcja zwraca “duży” obiekt wówczas wygodnie jest operować zmienną `POINTER`.

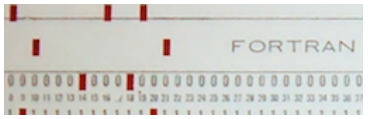
```
USE data_handler
     REAL x(100)
     REAL, POINTER :: y(:)
     :
     y => compact(x)
```

gdzie moduł `data_handler` zawiera funkcję, która usuwa duplikaty z tablicy:

```
  FUNCTION compact(x)
  ! Funkcja usuwa powtarzające się elementy tablicy x
        REAL, POINTER :: compact(:)
        REAL x(:)
        INTEGER n
        :               ! Znajdź liczbę n różnych wartości
        ALLOCATE(compact(n))
        :               ! Skopiuj elementy różne do compact
     END FUNCTION compact
```

Wyniku można użyć w wyrażeniach, pamiętając o związaniu go ze zdefiniowanym wskaźnikiem.

Pointer jako alias



Załóżmy, że pracujemy tylko z częścią tablicy

```
REAL, TARGET :: table(100, 100)
```

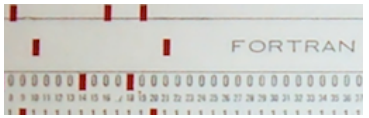
w zakresie o ustalonych wskaźnikach `table(m:n, p:q)`

Odwołania można zorganizować następująco:

```
REAL, DIMENSION(:,:), POINTER window
:
window => table(m:n,p:q)
```

Wskaźniki w window są `1:n-m+1, 1:q-p+1.`

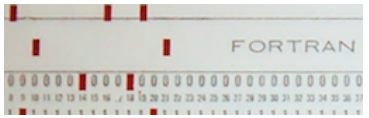
Private i public



Atrybuty `private` i `public` w definicjach modułów ograniczają zakres zmiennych I zakres dostępu do operatorów oraz zmieniają ogólne reguły dostępu

```
REAL, PUBLIC :: x, y, z       ! opcja
INTEGER, PRIVATE :: u, v, w

PUBLIC :: x, y, z, OPERATOR(.add.)
PRIVATE :: u, v, w, ASSIGNMENT(=), OPERATOR(*)

PRIVATE             ! Ustawia opcję dla całego modułu
PUBLIC :: tylko_to
```

Private...



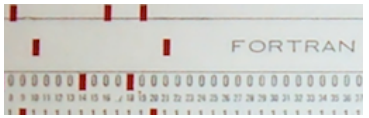
W definicjach typów :

- wszystko `PUBLIC`
- typ `PUBLIC składowe PRIVATE`
- wszystko `PRIVATE`

Przykład.

```
MODULE mine
   PRIVATE
   TYPE, PUBLIC :: list
      REAL x, y
      TYPE(list), POINTER :: next
   END TYPE list
   TYPE(list) :: tree
   :
END MODULE mine
```

Zadania



Funkcje wewnętrzne tablicowe. Praktyka.

Moduły.

Zadanie.
Napisz program, który będzie dodawać, mnożyć, …itd liczby zespolone zapisywane w dowolnej postaci: *x+iy* lub *a*exp(if)*.

Zadanie.
Obliczenia w poczwórnej precyzji.
Naszkicuj schemat programu obliczeń "*quadruple precission*". Napisz jego fragment.

Problemy...?