

## Podprogramy - wprowadzenie

Jedną z zasad praktyki dobrego programowania jest modularność kodu, oznaczająca, że różne części programu są w miarę możliwości wyodrębnione i niezależne od siebie. Fortran ma budowę modularną i może składać się z większej liczby segmentów. Jeden segment, segment programu głównego, jest wyróżniony i musi w programie wystąpić; oprócz niego mogą pojawić się segmenty podprogramów. Bez względu na wykorzystywane przez program podprogramy, realizacja programu zawsze rozpoczyna się od programu głównego.

O podprogramie można myśleć jako o osobnym programie, który może zostać uaktywniony (wywołany) w odpowiednim momencie. Jest niezależny od programu głównego i nic nie wie o zmiennych przez ten program wykorzystywanych. I odwrotnie, program główny nic nie wie o zmiennych wykorzystywanych przez podprogram. Pomimo tej niezależności i autonomii podprogram jest jednostką niesamodzielną i może być uaktywniony tylko przez segment nadrzędny.

Podprogramy umożliwiają wydzielenie określonego ciągu instrukcji i nadanie im postaci podprogramu. Z formalnego punktu widzenia można dosyć dowolnie dzielić program na mniejsze fragmenty podprogramów. Najczęściej uzasadnieniem i motywacją wykorzystania podprogramów jest zwiększenie czytelności programu w wyniku jego rozbicia na mniejsze podzadania opisywane za pomocą kolejnych segmentów lub sytuacja, w której określony ciąg czynności powinien być wielokrotnie realizowany w toku wykonania programu, i co więcej, w odniesieniu do różnych wartości.

Innym ważnym aspektem jest wykorzystanie gotowych podprogramów dostępnych w postaci bibliotek.

Fortran wyróżnia dwa rodzaje podprogramów: procedury (ang. *subroutine*) oraz funkcje (ang. *function*).

W zależności od ich umiejscowienia w strukturze programu w Fortranie 90/95 wyróżnia się trzy kategorie podprogramów:

- podprogramy wewnętrzne,
- podprogramy w modułach,
- podprogramy zewnętrzne.

W porównaniu z Fortranem77, w którym istniał tylko odpowiednik podprogramów zewnętrznych, stanowi to istotne rozszerzenie.

- Podprogram jest podprogramem wewnętrznym jeśli znajduje się pomiędzy instrukcją `CONTAINS` a instrukcją `END` (`END PROGRAM`, `END MODULE` lub `END` podprogramu zewnętrznego).
- Podprogram wewnętrzny jest podprogramem należącym do modułu jeśli znajduje się pomiędzy instrukcją `CONTAINS` oraz instrukcją `END MODULE`.
- Podprogram jest podprogramem zewnętrznym jeśli nie znajduje się wewnątrz segmentu `PROGRAM`, `MODULE` lub innego podprogramu. Podprogram zewnętrzny może znajdować się w oddzielnym pliku lub w tym samym pliku co inne segmenty po instrukcji `END` (`END PROGRAM`, `END MODULE` lub `END` należący do innego podprogramu).

Przynależność do kategorii pociąga za sobą określone konsekwencje. natomiast bez względu na nie, podstawowe zasady opracowywania podprogramów są takie same.

Rozpocniemy, trochę historycznie, od *podprogramów zewnętrznych* (ang. *external*).

Trzy podstawowe zagadnienia, jakie zostaną teraz omówione to:

- W jaki sposób nadać ciągowi instrukcji 'ramy' podprogramu?
- W jaki sposób wskazać, że podprogram ma zostać wykonany?
- W jaki sposób następuje przekazywanie informacji między segmentami?

Waga ostatniego zagadnienia jest uzasadniona faktem, że każdy segment programu stanowi niezależną całość. W szczególności podczas realizowania operacji należących do jednego

segmentu nie są dostępne żadne informacje na temat tego, co zdarzyło się w innych segmentach. W konsekwencji zmienna, której nadano określoną wartość w jednym segmencie nie jest 'widziana' w innych segmentach. To radykalne stwierdzenie zostanie później osłabione.

**Przykład:**

Napisać podprogramy typu *SUBROUTINE* oraz typu *FUNCTION*, realizujące operację dodania do siebie dwóch wartości rzeczywistych.

Czynność dodawania wartości dwóch (przykładowych) zmiennych *a* i *b* jest realizowana przez instrukcję:

```
c = a + b
```

Aby tej instrukcji nadać formę segmentu, należy ją odpowiednio 'obudować', dodając nagłówek, deklaracje oraz zakończenie w postaci instrukcji *END*.

**Budowa podprogramu typu *SUBROUTINE*** jest następująca:

```
SUBROUTINE nazwa_procedury (arg1, arg2, ..., argn)
IMPLICIT NONE
deklaracje
instrukcje
podprogramy_wewnętrzne
END SUBROUTINE nazwa_podprogramu
```

Zastosujemy to do naszej instrukcji sumującej:

```
SUBROUTINE SUMA(...)
IMPLICIT NONE
REAL :: a,b,c
c=a+b
END SUBROUTINE SUMA
```

Nagłówek wymaga uzupełnienia. Należy dopisać argumenty (nazywane *parametrami formalnymi*) podprogramu - inaczej mówiąc, listę zmiennych których wartości muszą być znane aby dało się zrealizować czynności podprogramu oraz zmiennych, które przechowują wyniki otrzymane w wyniku działania podprogramu. Lista argumentów definiuje rodzaj pomostu umożliwiającego wymianę informacji pomiędzy rozłącznymi 'wysepkami' programu zbudowanego z różnych segmentów. W omawianym bardzo prostym przykładzie potrzebne są dwa parametry wejściowe (*a* i *b*) oraz jeden wynikowy (*c*). Po wpisaniu tych zmiennych do nagłówka podprogramu otrzymuje się poprawną procedurę:

```
SUBROUTINE SUMA (a, b, c)
IMPLICIT NONE
REAL :: a,b,c
c=a+b
END SUBROUTINE SUMA
```

Podobnie można skonstruować podprogram typu *FUNCTION*, realizujący to samo zadanie

**Budowa podprogramu typu *FUNCTION*** jest następująca:

```
typ FUNCTION nazwa_funkcji (arg1, arg2, ..., argn)
IMPLICIT NONE
deklaracje
instrukcje
podprogramy_wewnętrzne
END FUNCTION nazwa_funkcji
```

W odróżnieniu od podprogramu typu *SUBROUTINE*, w którym nazwa służy jedynie do identyfikacji, nazwa podprogramu typu *FUNCTION* oprócz roli identyfikacyjnej pełni też funkcję 'nośnika' wyniku - stąd konieczność przypisania funkcji **typu**.

Kolejną różnicę stanowi to, że w podprogramie typu *FUNCTION* musi wystąpić przynajmniej jedno podstawienie pod nazwę funkcji (co umożliwia 'wyprowadzenie' wyniku na zewnątrz).

```
REAL FUNCTION SUMA (A,B)
IMPLICIT NONE
REAL :: A, B
SUMA = A + B
END FUNCTION SUMA
```

Tym razem nie ma potrzeby wprowadzać dodatkowego parametru do przechowania wyniku (zmienna *c* w przykładzie `SUBROUTINE`), bowiem rolę tę pełni nazwa funkcji.

**Uwaga:** deklaracja typu funkcji może znajdować się albo w nagłówku, albo wśród deklaracji:

```
FUNCTION SUMA (A,B)
IMPLICIT NONE
REAL :: A, B, SUMA
SUMA = A + B
END FUNCTION SUMA
```

**Uwaga:** Oprócz podstawienia pod nazwę funkcji wyniku jej działania istnieje możliwość wskazania zmiennej, pod którą zostanie podstawiona wartość będąca wynikiem działania funkcji. Wskazanie odbywa się w nagłówku funkcji za pomocą słowa kluczowego `RESULT`:

```
FUNCTION SUMA (A,B) RESULT (S)
IMPLICIT NONE
REAL :: A, B, S
S = A + B
END FUNCTION SUMA
```

**Uwagi:**

- Różnice pomiędzy podprogramami typu `FUNCTION` i `SUBROUTINE`, poza różnicami formalnymi oraz innym sposobem wywołania, nie są zbyt wyraźne.
- Mogą wystąpić funkcje bez parametrów. Wtedy w wywołaniu umieszcza się nazwę podprogramu wraz z parą nawiasów ( )

**Wywołanie podprogramów** - Parametry formalne i aktualne

Wywołanie podprogramu typu `FUNCTION` odbywa się analogicznie do wywoływania na przykład standardowych funkcji arytmetycznych, czyli poprzez umieszczenie ich nazwy wraz z parametrami w wyrażeniu.

Na przykład:

```
y=SUMA(2., z)
```

Wywołanie podprogramu typu `SUBROUTINE` odbywa się poprzez instrukcję `CALL` nazwa\_procedury wraz z parametrami.

Na przykład:

```
CALL SUMA(2., z, s)
```

Parametry znajdujące się w wywołaniu nazywa się *parametrami aktualnymi*.

W chwili wywołania podprogramu:

- Wyznaczane są wartości tych parametrów aktualnych, które są wyrażeniami, a wyniki są umieszczane w zmiennych tymczasowych (dotyczy tylko tych parametrów, które pełnią rolę danych dla podprogramu);
- Jeśli parametr aktualny jest stałą, to jego wartość jest umieszczana w zmiennej tymczasowej (dotyczy tylko tych parametrów, które pełnią rolę danych dla podprogramu);
- Wszystkie parametry aktualne są przekazywane do odpowiadających im parametrów formalnych w wywoływanym podprogramie;

Wykonanie podprogramu odbywa się 'tak jakby' wszystkie parametry formalne zostały zastąpione parametrami aktualnymi. Parametry formalne są tylko atrapami służącymi do opisanie czynności. Prawdziwe obliczenia są wykonywane w odniesieniu do parametrów aktualnych. Powiązanie parametrów aktualnych i formalnych następuje 'po kolei', pierwszemu

parametrowi formalnemu odpowiada pierwszy w kolejności parametr aktualny, itd. Dlatego też wymaga się aby:

- Liczba parametrów formalnych aktualnych i formalnych była równa;
- Typy odpowiadających sobie parametrów aktualnych i formalnych muszą być takie same.

### Zgodność parametrów – jak uniknąć błędów

Język Fortran 90 jest wyposażony w narzędzia umożliwiające kontrolę zgodności parametrów.

#### Atrybut `INTENT`

Atrybut `INTENT` umożliwia określenie roli spełnianej przez poszczególne parametry formalne:

- `INTENT (IN)` - parametr jest przeznaczony do dostarczenia danych. Próba zmiany wartości parametru w obrębie podprogramu jest błędem.
- `INTENT (OUT)` - argument służy do wyprowadzenia wyniku.
- `INTENT (INOUT)` - argument może pełnić rolę zarówno danej jak i wyniku. Wartość parametru może ulegać zmianie w toku realizacji podprogramu. Jest to domyślna wartość.

#### Przykład:

```
SUBROUTINE SUMA (a, b, c)
IMPLICIT NONE
REAL, INTENT (IN) :: a,b
REAL, INTENT (OUT)::c
c=a+b
END SUBROUTINE SUMA
```

Kolejna procedura realizuje to samo zadanie, ale tym razem z wykorzystaniem tylko dwóch zmiennych. Zmienna `b` pełni teraz podwójną rolę zarówno parametru wejściowego jak i wyjściowego; stąd jej atrybut `INTENT (INOUT)`:

```
SUBROUTINE SUMA (a, b)
IMPLICIT NONE
REAL, INTENT (IN) :: a
REAL, INTENT (INOUT) :: b
b=a+b
END SUBROUTINE SUMA
```

#### Deklaracja `INTERFACE`

Dobrym zabezpieczeniem zgodności parametrów formalnych oraz parametrów aktualnych jest 'zadeklarowanie' podprogramu w segmencie go wywołującym. Służy do tego 'łącznik' `INTERFACE`, w którym podaje się podstawowe informacje o podprogramie i jego parametrach. Łącznik rozpoczyna się słowem kluczowym `INTERFACE`, zaś kończy frazą `END INTERFACE`. Jego treść składa się z nagłówka podprogramu, deklaracji parametrów oraz instrukcji `END SUBROUTINE nazwa_procedury` lub `END FUNCTION nazwa_funkcji`. Łącznik `INTERFACE` umieszcza się bezpośrednio po deklaracji `IMPLICIT NONE`, a przed pozostałymi deklaracjami. Użycie `INTERFACE` nie jest wymagane, ale bardzo zalecane.

#### Przykład:

```
PROGRAM SUMOWANIE
IMPLICIT NONE
INTERFACE
  SUBROUTINE SUMA(a,b,c)
    REAL, INTENT(IN) ::a, b
    REAL, INTENT(OUT)::c
  END SUBROUTINE SUMA
END INTERFACE
REAL:: x=2.,y=3.,z
CALL SUMA(x,y,z)
```

```
WRITE(*,*) `suma=', z
CALL SUMA(20., 10., x)
WRITE(*,*) `suma=', x
END PROGRAM SUMOWANIE
```

### Zmienne lokalne

Rozważmy podprogram typu FUNCTION obliczający pole trójkąta o zadanych długościach boków:

```
real function POLE_TROJ (a,b,c)
implicit none
real, intent(in) :: a, b, c
real :: s
logical:: test1, test2
test1 = a>0.0 .and. b>0.0 .and c>0.0
test2 = (a+b>c).and.(a+c>b).and.(b+c>a)
if (test1 .and. test2) then
    s = (a+b+c)/2
    POLE_TROJ=sqrt(s*(s-a)*(s-b)*(s-c))
else
    write (*,*) 'to nie jest trojkat'
    POLE_TROJ=-999
endif
end function POLE_TROJ
```

W podprogramie tym występują zmienne będące parametrami oraz zmienne pomocnicze (s, test1, test2) obowiązujące lokalnie wewnątrz tego podprogramu. Takie zmienne nazywa się *zmiennymi lokalnymi*, bowiem obszar ich działania obejmuje tylko jeden segment.

**Uwaga:** Deklaracji zmiennych lokalnych nie umieszcza się w łączniku INTERFACE.

Dodatkowo zmienne lokalne charakteryzują się:

- są tworzone przy każdym wywołaniu podprogramu,
- są niszczone gdy podprogram kończy działanie,
- nie istnieją w pamięci pomiędzy wywołaniami,
- ich wartości nie są przechowywane pomiędzy kolejnymi wywołaniami danego podprogramu.

Ostatnią właściwość można zmienić nadając atrybut SAVE wybranym zmiennym. Atrybut SAVE sprawia, że wartości zmiennych lokalnych podprogramów są pamiętane pomiędzy kolejnymi wywołaniami co w szczególności umożliwia wyznaczenie liczby zrealizowanych wywołań podprogramu. Poniższy przykład ilustruje to zastosowanie oraz dodatkowo demonstruje, że zmiennym lokalnym, których wartości są inicjalizowane w wierszu deklaracji, atrybut SAVE jest nadawany domyślnie.

### Na przykład:

```
program test
implicit none
real ::a=1., b=2., c=3., d
interface
    subroutine ala (x,y,z)
        implicit none
        real, intent(in):: x,y
        real, intent (out)::z
    end subroutine ala
end interface
call ala(a, 2.,d)
write (*,10) d
call ala(b,c,d)
write (*,10) d
call ala(b, 2*b,d)
write (*,10) d
```

```
10 format ('suma=',f4.0)
end program test
subroutine ala (x,y,z)
implicit none
real, intent(in):: x,y
real, intent (out)::z
integer, save :: ile_wywolan=0
logical :: pierwszy=.true.
ile_wywolan = ile_wywolan+1
if (pierwszy) then
    pierwszy = .not.pierwszy !inaczej: pierwszy = .false.
    write (*,*) 'PIERWSZE wykonanie'
endif
write (*,'(a,i3)') 'wywołanie numer ',ile_wywolan
z = x+y
end subroutine ala
```

Wyniki:

**PIERWSZE wykonanie**

```
wywołanie numer  1
suma=  3.
wywołanie numer  2
suma=  5.
wywołanie numer  3
suma=  6.
```

### Podprogramy wewnętrzne

W odróżnieniu od podprogramów zewnętrznych podprogramy wewnętrzne mogą być wykorzystywane w obrębie segmentu (program główny lub moduł), w którym zostaną zdefiniowane. Są umieszczane na końcu programu głównego po wszystkich instrukcjach a przez instrukcją END. Poprzedzone są słowem kluczowym CONTAINS:

#### Przykład:

```
PROGRAM PROSTY
IMPLICIT NONE
REAL :: AA=2., BB=3., CC
CALL SUMA (AA,BB,CC)
WRITE (*,*) 'SUMA WYNOSI ',CC
CONTAINS
    SUBROUTINE SUMA (a, b, c)
    REAL, INTENT (IN) :: a,b
    REAL, INTENT (OUT)::c
    c=a+b
    END SUBROUTINE SUMA
END PROGRAM PROSTY
```

#### Uwagi:

- Podprogramy wewnętrzne mogą być wywoływane tylko z segmentu, w którym się znajdują.
- Mają automatycznie dostęp do wszystkich zmiennych, zadeklarowanych w segmencie głównym (dlatego też, między innymi, nie ma potrzeby umieszczać w nich deklaracji IMPLICIT NONE).
- Podprogram może zawierać odwołania do innych podprogramów należących do tego samego programu głównego, dołączonych modułów oraz podprogramów zewnętrznych.
- Podprogram wewnętrzny nie może zawierać w sobie innego podprogramu.

*Na temat podprogramów ciąg dalszy:*

- *rekurencja*
- *funkcje jako parametry*
- *tablice jako parametry*
  - *explicit-shape*
  - *assumed-size*
  - *assumed-shape*
- *tablice w roli zmiennych lokalnych*
- *argumenty opcjonalne (OPTIONAL, KEYWORD)*
- *Moduy (PUBLIC, PRIVATE, ONLY)*
- *'scoping' - zakres dziaania zmiennych*