Nauka Fortranu F90/95 przez przykłady dla początkujących.
Skrypt internetowy (wersja z 31 III 2008)

Krzysztof Rościszewski i Romuald Wit

Instytut Fizyki Uniwersytetu Jagiellońskiego.

Bardzo proszę Czytelnika aby w przypadku znalezienia, błędów i/lub nieścisłości przekazał odpowiednią informację na adres e.mailowy:

krzysztof.rosciszewski@uj.edu.pl

(Poprawki będą wprowadzane do nowych wersji skryptu).

Spis treści

I.C.2.Drugi prosty przypadek: zmienne oraz procedury w module, czyli tworzenie bloków z narzędziami (do obsługi naszych programów)

II.Z krótką wizytą na średnim poziomie zaawansowania
II.A.Nowe typy zmiennych
II.A.1.Definiowanie, deklarowanie i używanie typów "pochodnych", inaczej "złożonych" lub strukturalnych (derived type)
II.A.2.Wstępne informacje o zmiennych wskaźnikowych (czyli zmiennych pointerowych lub po prostu pointerach0 i o przydzieleniu danemu pointerowi (assigment) zmiennej (o atrubucie target) na którą pointer pokazuje.
II.A.3.Różnica pomiędzy zwykłą instrukcją podstawienia a instrukcją podstawienia w przypadku pointerów
II.A.4.Allocate i deallocate dla pointerów
II.A.4.1.Pointery jako składowe struktur (derived type)
II.A.4.2.Pointery jako składowe "derived type" – ciąg dalszy
II.A.5.1.Pointery jako argumenty subrutyn i funkcji
II.A.5.2.Pointery jako wartości funkcji

III.Zamiast zakończenia
III.A.1.Definiowanie swoich własnych operatorów

Wstęp

Niniejszy skrypt przeznaczony jest do szybkiej, praktycznej nauki Fortranu F90/95. Adresujemy go do osób "niecierpliwych", nieco bardziej zaawansowanych, które już wiedzą co to jest programowanie. A więc do tych, które już gdzieś miały styczność ze starym Fortranem F77 lub z językiem C, lub też do osób które znają Pascal ze szkoły średniej. Dlatego pojęcia takie jak kompilacja (compilation[1]), konsolidacja (linking), pojęcie plików (files) i katalogów (directories), nie będą wyjaśniane. Podobnie sprawy hardware'owe, związane z budową i działaniem komputera oraz jego podjednostek też przyjmujemy (w domyśle) jako dobrze znane Czytelnikowi.

Jedna z powszechnie przyjętych (i popularnych) metoda nauki jakiegoś języka programowania polega na zapoznaniu się z prostymi przykładami i od samego początku zakłada dużą aktywność Czytelnika (własnoręczne pisanie, uogólnianie i uruchamianie przykładowych programów). Powinny to być w miarę proste przykłady. Taki punkt widzenia został przyjęty w niniejszym opracowaniu. Drugie bardzo istotne zastrzeżenie, które robimy, dotyczy prezentowanych przykładów i stopnia komplikacji przytaczanych algorytmów. Ze względów dydaktycznych przedstawiamy proste przykłady i najbardziej prymitywne algorytmy; zbyt proste, aby miały jakiekolwiek profesjonalne lub półprofesjonalne zastosowania. Nasze przykłady mają jedynie ilustrować sens i wyjaśniać metody stosowania poszczególnych elementów języka F90/95. Decyzja niniejsza jest czysto pragmatyczna. Stopień komplikacji algorytmów stosowanych profesjonalnie jest bowiem taki, że wymaga od Czytelnika dużego skupienia i poświęcenia im zbyt dużej uwagi, którą na obecnym etapie kierujemy wyłącznie na poznanie podstawowych elementów nowego języka programowania. Z tego samego powodu zdecydowaliśmy się nie omawiać niektórych zagadnień oraz opuścić omawianie "rozwiązań alternatywnych". Prawie każda rzecz w F90/95 może być zapisana lub wykonana na kilka różnych sposobów i przy użyciu różnej składni. Dotyczy to zwłaszcza elementów języka, które są nieco przestarzałe, a utrzymano je jedynie w celu zachowania ciągłości w stosunku do starego Fortranu F77. W naszej opinii w pierwszym etapie należy skoncentrować się na sprawach podstawowych. Na rzeczy bardziej skomplikowane i/lub rzadziej używane kolej nadejdzie dopiero w drugim etapie nauki. Tak więc omawianie niektórych elementów składni języka, oraz profesjonalnych metod numerycznych i algorytmów w F90/95 odkładamy na później

Jednym z wniosków wynikających z takiego postawienia sprawy jest to, że skrypt raczej nie jest przeznaczony dla informatyków. Kierujemy go przede wszystkim do studentów studiów wyższych i półwyższych: inżynierskich, technicznych, pedagogicznych, do inżynierów, do chemików, fizyków, matematyków oraz maturzystów – entuzjastów komputera.

Literatura źródłowa

Istnieje wiele bardzo dobrych podręczników do nauki F90/95. Ich jedynymi wadami dla polskiego czytelnika jest to, że są na ogół niedostępne oraz drogie. Większość z nich jest napisana w języku angielskim (kilka po niemiecku). Każdy z nich pewne sprawy omawia nadzwyczaj jasno i kompetentnie, a inne pobieżnie (czasami wręcz "mętnie"). Tak więc w

[1] *W nawiasach podajemy odpowiednie terminy angielskie. Większość dostępnych na rynku, "lepszych" podręczników, to podręczniki w języku angielskim. Dalsze (zaawansowane) etapy nauki programowania F90/95, bez korzystania z takich podręczników, są bardzo utrudnione – w chwili obecnej prawie niemożliwe*.

czasie nauki trzeba zazwyczaj mieć dostęp do książek 2 – 3 różnych autorów. Naszym zdaniem najlepszy (i najłatwiejszy) podręcznik to:

800–stronicowy **Fortran 90 Programming**
autorzy: T. Ellis, I. Philips, T. Lahey
(Addison-Wesley Publishing Company, New York 1994)
Dodatkowa informacja: istnieje dobry opis do komercyjnego pakietu " Lahey Fortran 90", sprzedawanego przez Lahey Computer Systems, Inc. Również dysk CD (dosyć rozpowszechniony na polskim rynku) zawiera w sobie plik z niezłym manualem.

Dalsze podręczniki warte polecenia to:
400– stronicowy **Programmer's Guide to Fortran 90**
autorzy: W. Brainerd, C. Goldberg, J. Adams
(Springer, New York 1996)

350–stronicowy
Migrating to Fortran 90
autor: J. Kerrigan
(O'Reilly & Associates, Inc. Sebastopol, 1994)

350–stronicowy
Fortran 90/95 explained
Autorzy: M. Metcalf, J. Reid
(Oxford Science Publ.,Oxford ,1998)

Osoby zaawansowane, programujące na serio w F90/95 absolutnie powinny się zaopatrzyć w rewelacyjny podręcznik (wraz z gotową, półprofesjonalną biblioteką numeryczną na dysku CD):

Numerical Recipes in Fortran 90, autorów W.H. Press, S.A. Teuklsky, W.T. Vetterling and B.P. Flannery, Cambridge University Press Press,

Volume 2 of Fortran Numerical Recipes
autorzy: W. Press, S. Teukolsky, W. Vetterling, B. Flannery
(Cambridge University Press, Cambridge, New York, 1996)
Uwaga: do kompletu trzeba też kupić pierwszy tom
Numerical Recipes in Fortran
autorzy: W. Press, S. Teukolsky, W. Vetterling, B. Flannery
(Cambridge University Press, Cambridge, New York, 1992).
Zawiera on co prawda programy wyłącznie w F77, ale za to obszernie opisuje oraz dokładnie tłumaczy stosowane (w pierwszym i w drugim tomie) algorytmy.

Plik źródłowy

Plik źródłowy programu sporządzamy posługując się jakimkolwiek edytorem tekstu zapisującym tekst w zwykłym pliku tekstowym (ze znakami ASCII). W przybliżeniu oznacza to standardowe litery alfabetu angielskiego, bez stosowania w pliku symboli specjalnych, dotyczących sposobu formatowania tekstu. Tak więc dokument tekstowy napisany w WORD

nie może być plikiem źródłowym, chyba że pracując w WORD zapamiętamy plik jako zwykły tekst (jest to jedna z opcji WORD'a).

Tekst programu można pisać w formacie starego Fortranu F77 (fixed form). Wymaga to dostosowana się do wielu uciążliwych reguł. Np. w jednej linii tekstu możemy umieścić tylko jedną instrukcję lub deklarację. Instrukcje piszemy poczynając od 7–go miejsca w linii (7–a kolumna tekstu), aż do 72–giej kolumny itp. Jest to dosyć niewygodne. Preferowana, nowoczesna metoda, którą będziemy stosować w niniejszym opracowaniu, to "forma swobodna" (free form). Poniżej podamy zbiór prostych reguł związanych z wersją swobodną.

Tekst możemy umieszczać w dowolnym miejscu linii (aż do 132 kulumny). Pomiędzy instrukcjami lub deklaracjami można wstawiać dowolną liczbę odstępów. Odstępów używamy (na zasadach powszechnie stosowanych w ortografii) do oddzielania od siebie nazw zmiennych, słów zastrzeżonych (Fortran keywords), etykiet (labels), instrukcji (statements), oraz liczb i stałych. **Kompilator nie rozróżnia dużych i małych liter** z wyjątkiem liter wchodzących w skład napisów.

1. Znak **!** (exclamation mark) oznacza rozpoczęcie komentarza. Sam znak **!** oraz to, co po nim następuje, aż do końca linii, jest ignorowane przez kompilator. Oczywiste zastosowanie: wstawianie własnych uwag i komentarzy do tekstu programu. Wyjątek od tej reguły dotyczy znaku **!** występującego jako składnik napisu (uwaga: napisy stanowią jedną z podstawowych klas zmiennych występujących w F90/95)

2. Linia może zawierać kilka instrukcji lub deklaracji oddzielonych znakiem średnika **;** (semicollon) lub kilku średników

3. Jeżeli jakaś instrukcja lub deklaracja okazała się zbyt długa to można ją kontynuować w następnej linii tekstu pod warunkiem zaznaczenia znakiem **&** (ampersand) miejsca gdzie tekst się przerywa. Dozwolone jest maksymalnie 39 linii kontynuacji

4. Poprzednia reguła ma wyjątek. Jeżeli w jakieś linii nie udało nam się ukończyć **napisu** to nie tylko zaznaczamy znakiem **&** miejsce przerwania ale i w następnej linii dodajemy znak **&** na samym początku (ewentualne odstępy poprzedzające **&** są ignorowane)

5. Etykiety (statement labels), jeżeli tylko ich użycie jest konieczne (**co nie jest zbyt częste**), zapisywane są jako liczby całkowite w zakresie od 1 do 99999 przed odpowiednią instrukcją i oddzielone od niej przez jeden lub kilka odstępów

Podajemy jako przykład prosty program drukujący na ekranie monitora 4 linijki tekstu. W programie tym demonstrujemy podane powyżej reguły. Zwróćmy uwagę, że w tekście programu nie używamy w ogóle polskich liter (choć mogą one być używane w komentarzach):

```
program  drukuj_powitanie
                                  !!! to jest komentarz
                                  !!! program wydrukuje trzy razy slowo
                                  !!! witaj
write(*,*) ' witaj ! '            !!! wykrzyknik w napisie  traktowany jest
                                  !!! jako cześć napisu
write(*,*) ' witaj ! ';           !!! średnik  tutaj nie jest potrzeby, ale też
                                  !!! nie przeszkadza
```

```
                                  !!! instrukcja pisania drugiego napisu nie
                                  !!! zaczynała się od
                                  !!! początku linii, niemniej jednak drugi
                                  !!! napis będzie drukowany
                                  !!! na monitorze  od początku drugiej linii
!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
write(*,*) ‘ wit&
               &aj ! ‘            !!! ostatnia forma w działaniu jest
                                  !!! identyczna z dwoma
                                  !!! pierwszymi  --  ilustruje metodę
                                  !!! dzielenia napisu
!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
write(*,*)   &
‘witaj ! ‘                        !!! pokazujemy tutaj  zwykłą metodę
                                  !!! kontynuacji niezakończonej
stop                              !!! instrukcji;
                                  !!! stop przerywa natychmiast program
end program drukuj_powitanie
```

I. Podstawowe elementy i konstrukcje F90/95

I.A. Blok główny – program

I.A.1. Najprostszy program

Najprostszy program składa się z jednego bloku: jednostki rozpoczynającej się słowem kluczowym **program nazwa_programu,** a zakończonej słowem kluczowym **end program nazwa_programu.** W środku występują deklaracje zmiennych, a potem instrukcje.

Jako wyznawcy poglądu, że jeden dobry przykład jest lepszy od "tony" teorii rozpoczynamy naukę właśnie od konkretnego prostego przykładu. Przyjmijmy, że chcemy wyliczyć wartość funkcji dla danego *x*, danych wartości parametrów *n, a, c* podanych z klawiatury komputera oraz dla wartości *d*, którą chwilowo przyjmujemy za równą 5.0 (lecz w przyszłości, być może, ją zmienimy).

```
program licz_y;
implicit none                     !!! deklarujemy, że każda używana przez
                                  !!! nas zmienna MUSI
                                  !!! być zdeklarowana; brak deklaracji i
                                  !!! użycie takiej niezadklarowanej zmiennej
                                  !!! spowoduje błąd przy kompilacji
                                  !!! programu
real :: a, c, x                   !!! deklaracja  zmiennych rzeczywistych o
                                  !!! nazwach a, c, x
real :: y, d=5.0                  !!! jeszcze dwie zmienne;  za zmienną d
                                  !!! od razu podstawiamy określoną
```

```
                                        !!! wartość początkową (5.0)
                                        !!! wartość  d można później zmieniać,
                                        !!! więc nie jest to stała !!!
real, parameter :: pi=3.14159;          !!!  deklaracja  stałej – pi będzie od tej pory
                                        !!! synonimem liczby  3.14159
integer :: n                            !!! deklaracja  liczby całkowitej o nazwie  n
!!!!!!!!!!!!!!!!!
write(*,*) ‘ podaj wartosc a’;          !!! drukuje na ekranie napis “podaj wartość
                                        !!! a”;
read(*,*)  a                            !!! ta instrukcja powoduje, że wypisana z
                                        !!! klawiatury liczba
                                        !!! zakończona wciśnięciem klawisza enter
                                        !!! zostanie wczytana
                                        !!! do pamięci komputera i zmagazynowana
                                        !!! jako zmienna rzeczywista
                                        !!! o identyfikatorze   a

write(*,*) ‘ podaj n’;   read(*,*) n;   write(*,*) ‘ podaj c’, read(*,*) c
write(*,*) ‘ podaj zmienna  x’;  read(*,*) x
!!!!!!!!!!!!!!!!!
y= (a*x**n + c)/sin( pi*x)   - d
                                        !!! wyrażenie stojące po prawej stronie
                                        !!! zostanie wyliczone
                                        !!! a jego wartość podstawiona  pod
                                        !!! zmienną   y
                                        !!! dwie gwiazdki   **  oznaczają
                                        !!! podnoszenie do potęgi
                                        !!! jedna gwiazdka oznacza mnożenie,
                                        !!! / oznacza dzielenie;
                                        !!! teraz drukujemy wynik na ekranie
write(*,*) ‘ wyliczona warosc y(x)  = ‘,  y
                                        !!! ta instrukcja
                                        !!! drukuje napis oraz obliczoną wartość
                                        !!!  y(x)  w formacie domyślnym—czyli
                                        !!! najbardziej dogodnym dla komputera
stop
                                        !!! stop powoduje natychmiastowe
                                        !!!  zakończenie  programu
end program licz_y
```

UWAGI:

W podanym przykładzie drukowaliśmy na ekranie napis – został on zaznaczony jako łańcuch liter, znaków i cyfr pomiędzy apostrofami. W F90/95 istnieją tylko dwie formy apostrofów:

1. pojedyńczy, pisany u góry, taki sam z prawej strony łańcucha jak i z lewej;
2. podwójny (pojedynczy klawisz na klawiaturze) pisany również u góry, taki sam z prawej i z lewej strony łańcucha (napisu).

WORD, w którym pisany jest niniejszy skrypt, ma własne idee jak ustawiać cudzysłów otwierający i zamykający. Staraliśmy się poprawiać te "literackie formy" – ale na pewno nie udało się tego zrobić w każdym miejscu (w komentarzach nie jest to istotne).

PAMIĘTAJMY więc: w programie źródłowym mamy tylko jeden typ pojedynczego apostrofu i jeden typ podwójnego – ograniczenie to nie dotyczy oczywiście komentarzy gdzie po wykrzykniku możemy wstawić "cokolwiek".

Wróćmy do naszego pliku źródłowego. Załóżmy że plik źródłowy został zapamiętany w roboczym katalogu (work directory) pod nazwą licz.f90 (rozszerzenie .f90 lub .for są obowiązkowe). Kompilację i linking pod systemem UNIX (lub LINUX) wykonamy rozkazem

> **f95 –o licz licz.f90**

gdzie:

- **>** – to znak zachęty (prompt),
- **f95** – to wywołanie kompilatora (można też wołać f90, odrobinę starszą wersję kompilatora; różnice pomiędzy f90 i f95 "na pierwszym etapie nauki nie są widoczne"),
- **-o licz** – oznacza, że plik wynikowy (kod maszynowy, nadający się już bezpośrednio do wykonania) nazywa się licz,
- **licz.f90** – to nazwa pliku źródłowego

Jeżeli nie ma błędów to wykonamy program pisząc

>licz + enter
lub też
>licz & + enter

Rozkaz ten spowoduje załadowanie do pamięci i wykonanie programu. Znak **&** w systemie UNIX oznacza uruchomienie programu nie interakcyjnie, lecz w tle.

Jeżeli kompilujemy plik w systemie WINDOWS, to zazwyczaj mamy do dyspozycji odpowiednie okno do edycji i kompilacji programu, w którym pojawia się pełne menu dostępnych czynności zdefiniowanych odpowiednimi napisami objaśniającymi i/lub ikonkami.

Podsumujmy reguły, które można wywnioskować (w sposób oczywisty) z naszego przykładu:

1. Nasz program składa się (jak na razie) tylko z jednego bloku tekstu,
2. Program rozpoczynamy słowem kluczowym (zastrzeżonym) **program,** po czym następuje **nazwa_programu** – dowolny ciąg znaków składający się z liter, cyfr, oraz ewentualnie znaku podkreślenia, przy czym jako pierwsza musi wystąpić litera. Maksymalna liczba znaków w nazwie rozpoznawana jeszcze przez kompilator wynosi 31. Dalsze litery (jeśli ktoś ich jednak użył) będą ignorowane – tak jakby to był komentarz,
3. Program kończymy pisząc **end program nazwa_programu**. Dozwolona jest skrócona wersja: **end program** lecz odradzamy jej stosowanie – będziemy stosować prawie zawsze pełne formy, gdyż ułatwia to ogromnie szukanie i identyfikację błędów,

4. Instrukcja **implict none** występuje natychmiast po **program …** i oznacza bezwzględny nakaz deklarowania używanych zmiennych. Możliwe jest opuszczenie tej instrukcji – uważamy to jednak za niewskazane (zwłaszcza dla początkujących programistów),
5. Następnie następują deklaracje zmiennych, w dowolnej kolejności:
   - **real :: lista nazw** oddzielonych przecinkami, lub przecinkami + odstępami; każda nazwa stanowi unikalny identyfikator – ciąg liter i cyfr (oraz ewentualnie znaku podkreślenia) o długości od 1 do 31 znaków. Pierwsza musi występować litera
   - real :: – rozpoczyna deklarację listy zmiennych liczbowych, zmiennoprzecinkowych.
   - integer :: – rozpoczyna deklarację listy zmiennych całkowitych
   - real, parametr :: pi=3.14159 – to deklaracja stałej o nazwie pi o wartości 3.14159,
   - Słowo kluczowe **parameter** występujące po **real** (i oddzielone przecinkiem) jest pierwszym przykładem tzw. **atrybutu** występującego w jakiejś deklaracji. Z tym pojęciem jeszcze się spotkamy wiele razy,
6. Deklarowane zmienne (w dowolnym miejscu listy) można od razu zainicjować, tzn. przypisać im wartość początkową np.
   - real :: z, r12, r33= 3.5, w_nowe=5.5, w_stare;
   - real :: wyznacznik_nr2=0.0, x1prim, z1prim, z1bis; powyższy przykład demonstruje dodatkowo, jakie nazwy zmiennych można wybierać;
7. Po deklaracjach następują instrukcje (w naszym przypadku jedynie instrukcje pisania na ekranie monitora), instrukcje wczytywania liczb z klawiatury oraz prosta instrukcja arytmetyczna z podstawieniem wyniku pod zmienną y. Znaczenie 2-ch gwiazdek występujących w **read(*,*) …** oraz **write(*,*) …** jest następujące: pierwsza gwiazda oznacza użycie przez komputer **domyślnego** urządzenia zewnętrznego (zwykle ekranu monitora dla **write** oraz klawiatury dla **read**). Druga gwiazdka oznacza **domyślne** formatowanie drukowanych zmiennych (oraz liczb); maszyna użyje tylu znaków i w takiej formie, "jak jest jej wygodnie". W praktyce najczęściej zrezygnujemy z tej możliwości instruując kompilator jak ma zapisywać drukowane liczby,
8. Program kończymy instrukcją bezwarunkowego zatrzymania **stop** a koniec pliku źródłowego identyfikowany jest przez **end program nazwa_programu.**

Spróbujemy teraz nieco rozszerzyć zakres przed chwilą przyswojonych sobie wiadomości.

I.A.2. Zmienne i stałe podstawowe (intrinsic data types)

F90/95 (podobnie jak i stara wersja F77) zawiera i może wykonywać działania na 6 podstawowych typach danych. Są to:

1. liczby rzeczywiste (**real**),
2. całkowite (**integer**),
3. liczby zespolone (**complex**),
4. stałe logiczne (**logical**),
5. stałe znakowe (**character**),
6. liczby rzeczywiste podwójnej precyzji (**double precision**) – zdefiniowane analogicznie jak w starym Fortranie F77.

Nie rekomendujemy starego stylu, czyli używania **double precision** . Zamiast niego rekomendujemy używanie podwójnej precyzji w wersji **real(kind=…)** opisanej w rozdziale A5. Powodem tego jest fakt, że niektóre funkcje standardowe F90/95 (elemental functions) nie działają, lub nie działają dobrze jeśli ich argumentem jest zmienna double precision. Ponadto standard języka nie określa jaka jest dokładnie liczba miejsc po przecinku dla double precision; dla różnych maszyn może być to różna liczba, choć na ogół możemy spodziewać się co najmniej 10-11 cyfr znaczących, a przeciętnie 14-15 cyfr znaczących.

Tytułem zapowiedzi nadmieńmy, że oprócz **intrinsic data types,** F90/95 oferuje nam możliwość definiowania własnych typów danych – złożonych, lub inaczej pochodnych (**derived data types**), które odpowiadają znanym **"**rekordom" z PASCALA, czy "strukturom" z języka C. Ponadto możemy deklarować tablice (macierze), elementami których mogą być podane powyżej typy danych (zarówno **intrinsic** jak i **derived**) oraz deklarować wskaźniki (**pointer**) do dowolnych typów danych. Ale o tym dużo później …

I.A.3. Liczby oraz zmienne rzeczywiste

Obecnie napiszemy jedynie fragment programu w którym będą występować liczby (i zmienne) rzeczywiste oraz najprostsze instrukcje na nich wykonywane, uzupełnione o najczęściej stosowane funkcje standardowe (**intrinsic functions**). Zauważymy, że zmienne typu **real** w komputerze określone są w sposób domyślny; najczęściej jako liczby pojedynczej precyzji, tzn. z dokładnością nie wyższą niż do 7–8 miejsc po przecinku (np gdy liczba **real** zbudowana jest z 4 bytów). F90/95 dopuszcza deklarowanie dowolnej precyzji chociaż jak do tej pory producenci oprogramowania dostarczają kompilatorów zawierających standardowo pojedynczą i podwójną precyzję (15–16 miejsc po przecinku) a co najwyżej poczwórną precyzję (dla F90/95 w niektórych typach superkomputerów). Niemniej jednak możliwość teoretyczna istnieje….

Przykład:

```
program  licz_real; implicit none;
integer, parameter ::  DP= selected_real_kind(14)
real(kind=DP) ::  x_8byte, y, z;                !!! liczby podwójnej precyzji  mające
                                                !!! dokładność przynajmniej 14 miejsc
                                                !!! po przecinku lub więcej
                                                !!! Uwaga! Deklaracja stałej DP musi
                                                !!! wytąpić PRZED deklaracją
                                                !!! real(kind=DP)

real :: x, x10,  x0, xx, tr1, tr2_nowe ,a,  b, c, d, e,  f
real(kind=DP), parameter ::   zero=0.0d0, &
                      nul=0.0                   !!! ta linia jest kontynuacją  poprzedniej
real(kind=DP), parameter :: jeden=1.0_DP
integer,parametr ::  N=100, M=200;
!!!!!!!!!!!!!!                                  !!! Tutaj na chwilę przerywamy !
```

W podanym przykładzie liczby **real(kind=DP)** nazywane są liczbami z parametryzacją, **kind** jest słowem kluczowym, a DP jest stałą całkowitą. Producent oprogramowania ma pełną swobodę decyzji, jakie wartości parametru **kind** odpowiadają

jakiej dokładności. Najczęściej są to małe liczby całkowite, np. 3 lub 4. Programista nie wiedziałby więc jaką wartość DP podstawić gdyby nie to, że istnieje funkcja standardowa **select_real_kind(lmpp)** o wartościach całkowitych, która udziela takiej informacji. Lmpp – to liczba całkowita oznaczająca liczbę miejsc po przecinku jakiej żądamy (uwaga: polski przecinek dziesiętny w Fortranie piszemy jako kropkę dziesiętną).

Wynik podstawienia **DP=select_real_kind(14)** dostarcza takiej wartości DP, że liczby deklarowane jako **real(kind=DP)** mają na pewno dokładność 14 miejsc po przecinku lub lepszą. Jeżeli jednak żądamy nierealnej dokładności, np. 40 miejsc po przecinku, możemy być prawie pewni, że spowoduje to błąd kompilacji.

Funkcja **selected_real_kind** może być wywołana w inny, alternatywny sposób. Podamy go, choć praktyczna wartość tej informacji nie jest duża. Instrukcja

DP=selected_real_kind(P=14, R=30)

oznacza znalezienie takiej parametryzacji DP liczb typu real, która odpowiada 14 miejscom po przecinku (lub lepiej) oraz temu, że liczby mogą się zmieniać w zakresie od 10^{-30} do 10^{+30}. Litery **P, R** są słowami kluczowymi (keywords) kodującymi informację, która liczba odpowiada któremu wymaganiu (czyli odpowiednio: liczba cyfr po kropce dziesiętnej i zakres zmienności liczb). Deklarowanie naszym starym sposobem z jedną zmienną **DP=selected_real_kind(14)** jest w zupełności wystarczające, ponieważ obecne standardy zakresu zmienności liczb na komputerze są i tak więcej niż satysfakcjonujące…

Zwróćmy teraz uwagę na deklarację stałej

real(kind=DP), parameter :: jeden=1.0_DP,

gdzie zmienna jeden zostaje zainicjowana liczbą **1.0_DP**. Jak można się domyślić, zakończenie liczby przez **_DP** informuje kompilator, że ma do czynienia z liczbą jeden w dokładności 14 miejsc po przecinku. Oczywiście, deklaracja **real(kind=DP), parameter :: jeden=1.0;** jest też legalna. Tyle tylko, że zmienna **jeden** zostaje zainicjowana jako stała 1.0 o pojedynczej precyzji (mającej najwyżej 7 miejsc po przecinku). Powoduje to niewielki błąd zaokrąglenia (w stosunku do pierwszego przypadku), który w większość sytuacji może być zignorowany.

Uwaga! *Istnieje forma skrótowa (stosowania której jednakże nie zalecamy), w której zmiast pisać* ***real(kind=DP) ….*** *można napisać* ***real(DP) …..***

Przejdziemy teraz do działań na liczbach **real**. Zastosujemy zmienne przed chwilą zadeklarowane.

```
!!!!!!!!!!!!!!!!!!!!!!!!!! !!!      !!! kontynuacja programu   licz_real
x_8byte =  1.0e-12                  !!! podstawiamy za x_8byte  liczbę 10 do
                                    !!! minus 12 potęgi;
                                    !!! liczba ta ma pojedynczą precyzję;  przed
                                    !!! podstawieniem jest automatycznie
                                    !!! zamieniana na liczbę podwójnej
                                    !!! prezycji; niestety, mieszanie
                                    !!! różnej precyzji powoduje, że  miejsca
                                    !!! od  7 do 15 po
                                    !!! przecinku  zmiennej  x_8byte są już
                                    !!! niedokładne
!!!!!!!!!!!!!!!!!!!!!!!!!!!!   Uwaga!  Zapis   e-12  oznacza 10 do potęgi minus 12
!!!!!!!!!!!!!!!!!!!!!!!!!!!!   zapis    e8   oznacza    10  do potęgi 8;
!!!!!!!!!!!!!!!!!!!!!!!!!!!!   jest to tzw.  notacja inżynierska
```

```
x_8byte = 1.0e-12_DP                    !!! teraz  x_8byte  ma już pełną
                                        !!! dokładność 14 miejsc
                                        !!! dziesiętnych

y =   1 +  x_8byte                      !!! całkowita liczba jeden automatycznie
                                        !!! jest zamieniana na
                                        !!! podwójną precyzję tuż przed dodaniem
                                        !!! do x_8byte
                                        !!! niestety, mamy tutaj do czynienia znów
                                        !!! z "zaokrągleniem"

z=   y**2.5_DP*x_8byte + 1.5_DP/(x_8byte**2 + 1.0_DP)

                                        !!! Podobnie jak w innych językach
                                        !!! programowania priorytet działań jest
                                        !!!  następujący:
                                        !!! pierwsze wykonujemy potęgowania,
                                        !!! potem mnożenia i dzielenia, na końcu
                                        !!! dodawania  i odejmowania.
                                        !!!  Oczywiście, zawsze najpierw
                                        !!! wykonujemy działania w nawiasach
z= 1.0_DP;    z=z+1.0_DP;
write(*,*) z                            !!! wydrukowana zostanie liczba  2  w
                                        !!! formacie  maszynowym
                                        !!!  teraz stosowanie funkcji standardowych
x0= 3.14159;     x0=sin(x0);
z  =  3.14159_DP;   z = sin(z)
!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!    !!!  przerywamy  program  licz_real
```

Ostatnie linie demonstrują użycie funkcji standardowej sinus. Funkcje standardowe są dosyć "inteligentne" w F90/95. Rozpoznają automatycznie typ zmiennej (i jego parametryzację **kind**) oraz dostarczają wyniku tego samego typu (i precyzji). Poniżej podamy kilka najczęściej stosowanych funkcji standardowych:

```
!!!!!!!!!!!!!   kontynuujemy program licz_real
x= 1.0                                  !!! pojedyncza precyzja dla jedynki, także
                                        !!! dla x
a= exp(x)                               !!! funkcja wykładnicza
b=log(x)                                !!! funkcja logarytm (naturalny)
c= abs(x)                               !!! wartość bezwzględna,  czyli  moduł

!!!!!!!!!!!!!!!!!!!!!!!!!
a=  acos(x)                             !!! arcus cosinus  (argument musi być
                                        !!! pomiędzy  1 a –1)
b= asin(x)                              !!! arcus sinus
c= tan(x)                               !!! tangens
!!!!!!!!!!!!!!!!!!!!!!!!
a=  atan(x)                             !!! arcus tangens
b = cosh(x)                             !!! cosinus hiperboliczny
```

```
c= fraction(x)                          !!! podaje część dziesiętną liczby  (część
                                        !!! całkowita jest odrzucana)
!!!!!!!!!!!!!!!!!!!!!!
a= sqrt(x)                              !!! pierwiastek kwadratowy
                                        !!! przerywamy przykład
stop
end program licz_real
```

I.A.4. Liczby całkowite i zmienne typu integer

Zasady (z nielicznymi wyjątkami) są takie same, jak już poznane zasady z rozdziału I.A.3. Przytaczamy analogiczny do "licz_real" fragment programu :

```
!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
program  licz_integer;   implicit none;
integer,parameter ::  IDP= selected_int_kind(15)
                                        !!! selected_int_kind  pozwala znaleźć
                                        !!! całkowitą liczbę parametryzującą
                                        !!! daną dokładność  deklarowanych liczb
                                        !!! "integer"
integer(kind=IDP) ::  j, k, n0;
                                        !!! liczby  (najprawpododobniej 64 bitowe)
                                        !!! mające przynajmniej do 15  cyfr
                                        !!! znaczących, a więc  z
                                        !!! przedziału –10**15 do +10**15;
                                        !!! jeżeli 64 bitowe liczby integer nie są
                                        !!! przez producenta kompilatora
                                        !!! zdefiniowane wtedy za IDP zostanie
                                        !!! podstawiona  liczba – 1
                                        !!! sygnalizując w ten sposób
                                        !!! niepowodzenie naszej deklaracji;
                                        !!! Uwaga!  Deklaracja stałej IDP musi
                                        !!! nastąpić PRZED
                                        !!! deklaracją  integer(kind=IDP) ….

integer :: maximum, minimum
integer ::  licznik, r, s, t, jaki_kind   !!! na 32-bitowym komputerze standardowy
                                        !!! typ integer (bez parametryzacji kind)
                                        !!! ma na ogół  4 bajty,  a więc zakres liczb
                                        !!! całkowitych jest  od  około    -2* 10**9
                                        !!! do około  +2*10**9

integer(kind=IDP), parameter ::   izero=0_IDP
                                        !!! izero jest stałą o wartości zero
                                        !!! zero  podwójnej precyzji   zaznaczamy
                                        !!! przez przyrostek   " _IDP"
real ::   x, y, z
                                        !!! teraz  instrukcje:
```

```
j = 10_IDP; n0= -100_IDP;   k=  j**j/(n0-  j) +  j +7
                                                    !!! ostatnia liczba  7  zostanie
                                                    !!! skonwertowana z pojedynczej precyzji
                                                    !!! do podwójnej
 r =  10;  s = 3;  t = r/s ;   write(*,*) '  t = ', t
                                                    !!! Uwaga!  dzielenie liczb całkowitych  w
                                                    !!! Fortranie jest wyjątkowe, ponieważ
                                                    !!!  zawsze dostarcza liczby całkowitej;
                                                    !!!  reszta z dzielenia (to co po kropce
                                                    !!! dziesiętnej) jest
                                                    !!! automatycznie odrzucana;  tak więc
                                                    !!! wydrukowana
                                                    !!!  wartość NIE  będzie równa
                                                    !!! 3.33333333,   ale  BĘDZIE RÓWNA   3

                                                    !!! teraz niektóre najczęściej stosowane
                                                    !!! funkcje standardowe o argumentach
                                                    !!! typu integer
x =   sin(k)
                                                    !!! sin daje wartości real; liczba k ulegnie
                                                    !!! konwersji do real przed
                                                    !!! wyliczeniem  sinusa

r  = abs( 10+s)                                     !!! wartość bezwzględna (moduł)
                                                    !!! funkcja  abs od argumentu integer  daje
                                                    !!! wartość integer
                                                    !!! jest to kolejny przykład na to, że
                                                    !!! większość
                                                    !!! funkcji standardowych w Fortranie
                                                    !!! automatycznie rozpoznaje typ zmiennej
                                                    !!! i dostosowywuje się  do niego

 j = int(r, kind=IDP)                               !!! funkcja int dokonuje konwersji liczby r
                                                    !!! (pojedyncza   precyzja)  do  typu
                                                    !!! integer (kind=IDP)

 r = int(1.1)                                       !!! funkcja int  odcina część dziesiętną, za r
                                                    !!! jest podstawiona
                                                    !!! wartość  całkowita   równa  1
                                                    !!! (Uwaga! Brak drugiego argumentu, a
                                                    !!! więc  "kind" standardowy !)

  j = int(1.1, IDP)                                 !!! to samo, tylko wartość 1  wychodząca
                                                    !!! z  int  skonwertowana
                                                    !!! jest do  typu   integer(kind=IDP);
                                                    !!! oczywiście
                                                    !!! znów mamy do czynienia z błędem
                                                    !!! zaokrąglenia
                                                    !!! jeśli zapomnimy, jaki   KIND został
                                                    !!! przyporządkowany zmiennej  j
```

```
                                        !!! możemy to sprawdzić stosując funkcję
                                        !!! standardową  o nazwie “kind(...)”
jaki_kind= kind(j)

                                        !!! funkcja kind ma wartość “integer”;
                                        !!! można ją stosować zarówno
                                        !!! do badania  KIND  w liczbach typu
                                        !!! integer jak ROWNIEŻ
                                        !!! do badania KIND  w liczbach typu real

maximum = max(r,s)                      !!! pod wartość zmiennej maximum zostaje
                                        !!! podstawione większa z liczb r, s
minimum= min(r,x)                       !!! to samo dla minimum
!!!!!!!!!!!!!!!!!!!!!!!!!!!
                                        !!! funkcje “max”  oraz “min”   mogą mieć
                                        !!! więcej argumentów;
                                        !!! argumenty mogą być zarówno typu
                                        !!! “integer” jak i  “real” -
                                        !!! patrz przykład poniżej:

maximum =   max(r, s, 10+r, x*y*z)
!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
s =    15;   r = modulo(r, 10);    write(*, *)  ‘ modulo(15, 10) =’, r
                                        !!! bardzo ważna  funkcja  modulo (może
                                        !!! operować także na zmiennych typu
                                        !!! “real”);
                                        !!! wynikiem jej działania jest  RESZTA
                                        !!! pozostała po dzieleniu całkowitym
                                        !!! w naszym przykładzie  zostanie
                                        !!! wydrukowana   wartość r   równa   5
stop; end program licz_integer
```

Na tym przerywamy przykłady z funkcjami standardowymi. Pokazaliśmy tylko te funkcje, które są używane bardzo często. Biblioteka funkcji i procedur F90/95 jest zbyt obszerna (około 160 pozycji), aby pokusić się o omawianie większej liczby przykładów.

Uwaga końcowa: dla typów **real, complex, integer, logical, character** zdefiniowana jest możliwość używania parametryzacji **KIND**, np. **complex(kind=DP)** lub **logical(kind=3)**. Naszym zdaniem dla **real** oraz **complex** określenie dokładności jest bardzo pożyteczne; dla **integer** umiarowanie pożyteczne (możliwość tworzenia tzw. długich liczb całkowitych), natomiast dla **logical** i **character** nie jest to zbyt potrzebne.

I.A.5. Stałe i zmienne logiczne (logical)

Typu logicznego nie omawialiśmy do tej pory. Jest bardzo prosty – mamy tylko dwie stałe: prawda i fałsz. W składni F90/95 zapisuje się je jako **.true.** oraz **.false.** Podany poniżej fragment programu zademonstruje jak deklarować zmienne i stałe logiczne i jakie na nich możemy wykonywać działania:

```
program   logical_demo;  implicit none;
real ::  a= 1.0,   b=10.0,   c=100.0                   !!! pomocnicze zmienne rzeczywiste
!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
logical,parametr ::    prawda=.true., falsz=.false.            !!!  deklaracja 2-ch stałych
logical ::  test1, war,   iflag , iflag1                       !!!   zmienne
logical :: test2, test3, test4, test5, test6, test7
logical :: war1,war2,   war1o2, war1lub2, war3
!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
war = .true.  ;                                    !!! za war podstawiliśmy  true
test1  =  a<c                                      !!! test1 = .true. bo  nierówność a<c jest
                                                   !!! prawdziwa;
test2  =  a <= c                                   !!! nierówność  a jest mniejszy lub równy
                                                   !!! c  (też prawdziwa)
test3  =  a == c                                   !!! test3 = .false. , bo  a nie jest równe c
test4  =  a>= c                                    !!! test4 = .false. ,  bo  a nie jest większe lub
                                                   !!! równe c
test5  =  a> c                                     !!! test5 = .false. ,   bo a nie jest większe niż c
test6  =  a /=c                                    !!! test6 = .true.  ,  bo   a nie równa się c
!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!

test7=   2.0*a**2 + b/c  >  0.0                    !!! bardziej skomplikowany przyklad,
                                                   !!! test7   = .true.
                                                   !!! dotychczas mieliśmy relacje
                                                   !!! arytmetyczne,  teraz będą logiczne:
war1= .true.; war2= .false.
war1o2 =   war1 .and.  war2                        !!!  koniunkcja logiczna;   war1o2 = .false.
war1lub2 = war1 .or. war2                          !!!  suma logiczna;    war1lub2 = .true.
war = .not.  war1                                  !!! zaprzeczenie logiczne,  war = .false.
iflag =   war1  .eqv. war2                         !!! .false.   war1 NIE jest ekwiwalentne
                                                   !!! (równoważne) war2
iflag1 = war1 .neqv. war2                          !!! neqv  -> nie równoważne
!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
!!!    bardziej  skomplikowany przykład:

war3 =   (.true.  .and.  ( war1  .or.  war2 ) )  .and.  ( a> b+c)       !!!  war3 = .false.
                                                   !!!  przerywamy program
stop; end program logical_demo
```

Uwaga!

Operatory relacji arytmetycznych mają słowne synonimy (historycznie pochodzące z F77). Jeżeli porównujemy dwie liczby a oraz b to operatory relacji zapisane w obu pierwszych kolumnach tabelki (patrz poniżej) są identyczne:

A < b	a .lt. b	a mniejsze niż b	a less than b
A <= b	a .le. b	a mniejsze lub równe b	a less than or equal b
A == b	a .eq. b	a równe b	a equal b
A >= b	a .ge. b	a większe lub równe b	a greater than or equal b
A > b	a .gt. b	a większe niż b	a greater than b

A /= b	a .ne. b	a nie równe b	a not equal b

Uwaga!
W złożonych wyrażeniach, co do których nie mamy pewności jak i w jakiej kolejności są wykonywane operacje, używamy albo nawiasów (rekomendujemy!), albo bierzemy pod uwagę następujące reguły:

*Wszystkie operatory arytmetyczne (plus, minus, dzielnie, mnożenie, potęgowanie) mają wyższy priorytet (są wykonywane w pierwszej kolejności) niż operatory logiczne (**and, or**, **not**,…) lub operatory relacji większy, mniejszy,….*

*Druga reguła: operatory relacji arytmetycznych mają wyższy priorytet niż operatory logiczne (czyli **.gt.** ma wyższy priorytet niż **.and.**)*

Trzecia reguła dotycząca piorytetu operatorów relacji logicznych pomiędzy sobą:
*najwyższy priorytet - **.not.***
*następnie - **.and.***
*- **.or.***
*najniższy priorytet - **.eqv. , .neqv.***

Zauważmy że dużo prostsze jest używanie nawiasów celem wskazania, w jakiej kolejności wykonujemy wyrażenia logiczne, niż pamiętanie powyższych reguł!

I.A.6. Stałe i zmienne zespolone (complex)

Liczby zespolone zapisujemy w F90/95 jako pary liczb rzeczywistych, czyli w postaci (a,b), gdzie a, b to para liczb rzeczywistych oznaczająca odpowiednio część rzeczywistą i część urojoną. Dlatego też wiele informacji które poznaliśmy dla stałych i zmiennych typu **real** stosuje się również dla typu **complex**. Jak poprzednio, użyjemy przykładowego programu:

```
program  complex_demo
implicit none;
real ::    re_z,  im_z, modul_z                    !!!  pomocnicze zmienne real
integer, parameter ::  DP = selected_real_kind(14);   !!! pomonicza stała typu integer
real, parameter :: zero=0.0                             !!! pomocnicza stała typu real
complex, parameter :: c_zero= (zero, zero)              !!! stała typu complex
complex ::  r,  s, z1,  z2,   c_zero=(0.0, 0.0)
                                                        !!! zmienne o standardowej (zazwyczaj
                                                        !!! pojedynczej) precyzji
complex(kind=DP) ::   w,  u,    jeden =(1.0_DP,   0.0_DP)

                                                        !!! jak widać powyżej  w, u, jeden,  sa
                                                        !!! zmiennymi o
                                                        !!! "podwójnej" precyzji
                                                        !!! tzn. maja przynajmniej po 14 cyfr
                                                        !!! znaczących;
                                                        !!! jeden  jest już podstawione
                                                        !!! w  =  (1.0,  1.0_DP)
```

```
                                        !!! część rzeczywista w  (1.0) ulegnie
                                        !!! automatycznej konwersji na liczbę
                                        !!! wyższej precyzji (uwaga: błąd
                                        !!! zaokraglenia)
z2 =   z1*z1 -  jeden/( z1 +  (1.0E-5,  1.0) )

                                        !!! kolejne przykłady automatycznej
                                        !!! konwersji  typów

z1 =   3 *(1.0, 1.0)  +   30            !!! co w czasie wykonania podlega
                                        !!! automatycznej
                                        !!! konwersji do   (3.0, 3.0) + (30.0 , 0.0 )
                                        !!! innymi słowami: liczby rzeczywiste oraz
                                        !!! integer
                                        !!! będą potraktowane jako
                                        !!! liczby zespolone o  zerowej części
                                        !!! urojonej

r = ( 11.0, 22.0) ;
s= conjg( r)  ;    write(*,*)  ' s = ', s
                                        !!! funkcja standardowa conjg
                                        !!! pozwala otrzymać liczbę
                                        !!! zespoloną sprzężoną, czyli  s =(11, -22)
                                        !!! im_z  =   aimag(r)
                                        !!! im_z jest równe 22, czyli części urojonej
                                        !!! liczby r

re_z   =   real(r)
                                        !!! funkcja "real"  wycina część rzeczywistą
                                        !!! liczby zespolonej;
                                        !!! dlatego   też   re_z  = 11.0
                                        !!! UWAGA: argumentem funkcji "real"
                                        !!! może być
                                        !!! także zmienna lub wyrażenie
                                        !!! typu integer lub real,  wynik pozostaje
                                        !!! liczbą  rzeczywistą
                                        !!! Uwaga: możliwy jest (ale niekonieczny)
                                        !!! drugi  argument funkcji real
                                        !!! określający   "kind"

modul_z =  abs( (1.0,1.0))              !!! wynik, to pierwiastek kwadratowy z 2,
                                        !!! czyli moduł z liczby zespolonej (1, 1)
                                        !!! UWAGA:   funkcja abs jest określona
                                        !!! również dla argumentów
                                        !!! rzeczywistych i całkowitych  (patrz
                                        !!! rozdzial  A.5)

u =  (1,   1.0_DP);

u = sin(u) + cos(u) + exp(7.0_DP * u) + log(u)  + sqrt(u)
```

```
                                        !!! standardowe funkcje matematyczne
                                        !!! (poznane
                                        !!! wcześniej dla argumentów real)
                                        !!!  są rozszerzone na dziedzinę liczb
                                        !!! zespolonych;
                                        !!!  rozpoznawanie argumentu (wraz z
                                        !!! dokładnością,  tzn. rozpoznawanie
                                        !!! parametryzacji KIND)  jest
                                        !!! automatyczne;
                                        !!! wynik jest tego samego typu co argument
                                        !!! ostatnia własność nie dotyczy  typu
                                        !!! double precision, o czym już
                                        !!! wspomnieliśmy w rozdziale I.A.4.

                                        !!! Uwaga!  Inne funkcje  standardowe np.
                                        !!! typu
                                        !!! arcus (acos, asin, atan)
                                        !!! oraz funkcje hiperboliczne  (cosh,
                                        !!! sinh,tanh) oraz   funkcja  tangens (tan)
                                        !!! nie mają rozszerzenia na dziedzinę liczb
                                        !!! zespolonych;
                                        !!! rozpoznają one jednak  parametryzacje
                                        !!! "KIND"
                                        !!! swojego rzeczywistego argumentu

stop; end program complex_demo
```

Obecnie przejdziemy do stałych i zmiennych znakowych.

I.A.7. Stałe i zmienne znakowe, czyli napisy (character)

Zmienne znakowe stanowią bardzo ważny (i użyteczny) typ danych w Fortranie. Ich deklarację poprzedzamy słowem **character(len= liczba_calkowita) ::**
len jest skrótem od length – oznacza długość deklarowanej stałej (lub zmiennej) znakowej; tak więc **len= liczba_całkowita** oznacza że deklarowane zmienne będą miały po "**liczba_całkowita"** znaków. Istnieje forma skrótowa gdzie dozwolone jest opuszczenie słowa kluczowego **len** a więc możemy pisać np. **character(8)** zamiast character(len=8). Jeszcze inna dozwolona forma (pochodząca ze starego F77) to **character*8** - nie rekomendujemy jej.

Dla zmiennych znakowych określone są tylko dwie elementarne operacje: zlepianie napisów (w jeden wspólny napis) oraz podstawianie. Biblioteka funkcji standardowych F90/95 zawiera jednak kolekcję bardzo przydatnych funkcji działających na napisach i przekształcających napisy (inny termin czyli synonim do "napisu" to "łańcuchy" znakowe; programiści najczęściej używają jednak slangu i mówią o "stringach").

Przejdźmy do przykładu:

```
program  character_demo;  implicit none;
integer ::   j = 5,   k,  l , dlugosc                !!! pomocnicze zmienne całkowite
```

```
!!!!!!!!!!
character(len=80), parameter ::  napis =’ START’
                                                    !!! “napis” jest synonimem stałej 80
                                                    !!! znakowej,
                                                    !!! której pierwsze 5 liter
                                                    !!! to: S, T, A, R, T,   pozostałe
                                                    !!! (niepodstawione przez nas)
                                                    !!! znaki od pozycji
                                                    !!! nr  6 do pozycji  80   automatycznie
                                                    !!! wypełnione są znakiem odstępu
!!!!!!!!  teraz zmienne znakowe:

character(len= 1) :: litera1, litera2, znak, cyfra=’5’
character(len=4)  :: ext=’.out’
character(len= 2*j+1) ::  jedenascie          !!! Uwaga!  JUŻ WCZESNIEJ
                                                    !!!  w czasie deklarowania zmiennych
                                                    !!!  integer  za  j podstawiona była  liczba 5

character(len=22) :: nazwisko, imie,  nazwisko_imie,  tytul = ’zaczynamy !!’

                                                    !!! uwaga   znaki  !!   kończące słowo
                                                    !!! “zaczynamy”  NIE  SĄ TRAKTOWANE
                                                    !!! jako komentarz tylko  jako litery w
                                                    !!! napisie

character(len = 16) ::  test
!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
nazwisko = ’ Nowak’                          !!! na oznaczenie napisów używamy tylko
                                                    !!! ’....’
                                                    !!! lub  ewentualnie podwójnego apostrofu
                                                    !!! ”...”
imie  = ’ HIPOLIT’
                                                    !!! za nazwisko podstawiliśmy napis
                                                    !!! składający się z 5 liter
                                                    !!! brakujące  17 znaków
                                                    !!! (razem ma być 22)
                                                    !!! wypełnione są odstępami

nazwisko_imie = ’ HIPOLIT ’ // ’ Nowak’
                                                    !!! operator // powoduje zlepianie napisów
                                                    !!! (ang: concatenation);
                                                    !!! jeśli długości zlepianych napisów są zbyt
                                                    !!! długie to zabraknie odpowiedniej liczby
                                                    !!! liter (i nastąpi obcięcie);
                                                    !!! uwaga:  //  zapisujemy jako dwa znaki
                                                    !!! dzielenia (bez odstępu)

                                                    !!! obecnie otwieramy  plik (na dysku) do
                                                    !!! którego będziemy drukowali nasze
                                                    !!! napisy
```

```
open(unit=66, file='napisy'//cyfra//ext)
                                        !!! otwieramy plik o nazwie "napisy5.out"
                                        !!! file jest słowem  kluczowym (keyword)
                                        !!! identyfikującym argument nr 2
                                        !!! instrukcji   open -  w tym wypadku
                                        !!! nazwę pliku
                                        !!! w programie plik identyfikowany będzie
                                        !!! jako  jednostka  (unit) lub urządzenie
                                        !!! nr 66
                                        !!! dozwolone jest używanie   dowolnych
                                        !!! numerów  od 10 do 99,
                                        !!! przyjmuje się, że numery 5, 6
                                        !!! standardowo (głównie ze względów
                                        !!! historycznych) oznaczają
                                        !!! klawiaturę (naprawdę – standardowy
                                        !!! input), i ekran (naprawdę -  standardowy
                                        !!! output); liczby   5,  6 można
                                        !!! zastępować znakiem
                                        !!! gwiazdki   oznaczającej jednostkę
                                        !!! domyślną  (default unit)
                                        !!! Uwaga! 'napisy'//cyfra//ext    nie jest
                                        !!! zmienną, ale  stałą znakową zlepioną z
                                        !!! trzech  mniejszych napisow; zauważmy,
                                        !!! że ta nazwa nie zawiera znaków odstępu
                                        !!! (w środowisku WINDOWS znaki
                                        !!! odstępu nie są dozwolone jako składniki
                                        !!! nazw  plików; w środowisku UNIX –
                                        !!! tak; ale jest to praktyka automatycznie
                                        !!! prowokująca poważne błędy ze
                                        !!! strony programisty.  I dlatego
                                        !!! zdecydowanie ODRADZAMY  !!!!!

                                        !!! teraz rozszerzamy możliwości
                                        !!! instrukcji  pisania,  czyli write;
                                        !!! ogólna postać  to  write(argument1,
                                        !!! argument2, …..)
                                        !!! lista_stalych_i_zmiennych
                                        !!! Zauważmy, że do tej pory
                                        !!! argumentami  write  były dwie gwiazdki,
                                        !!! co interpretowalismy jako domyślne
                                        !!! wartości argumentów
write(66, fmt='(a80)')  nazwisko_imie

                                        !!! do jednostki nr 66 (czyli do naszego
                                        !!! pliku o nazwie napisy5.out)
                                        !!! wydrukowana jest zmienna
                                        !!! "nazwisko_imie";
                                        !!! za słowo kluczowe   fmt  (format)  jest
                                        !!! podstawione   a80  co
```

```
                                        !!! interpretowane jest jako  polecenie
                                        !!! drukowania  80 znaków w linii
                                        !!! (alfanumerycznych -- więc stąd mamy
                                        !!! kod  -- “a”)

                                        !!! ponieważ zmienna nazwisko_imie
                                        !!! zawiera tylko 22 znaki,
                                        !!! możliwość użycia  (drukowania)
                                        !!! pozostałych (80-22)  znaków  nie zostanie
                                        !!!  zrealizowana

jedenascie = ’123456789AB’;

                                        !!! zmienne znakowe  mają ponumerowane
                                        !!! swoje  pojedyncze znaki składowe
                                        !!! standardowo możemy używać  części
                                        !!! napisu (bez dodatkowego deklarowania)

write(66,*) ’ drukowanie czesci zmiennej jedenascie od znaku nr 3 do znaku nr 8’
write(66,fmt=’(a6)’)  jedenascie(3:8)          !!! wydrukowane będzie   345678
!!!!!!!!!!!

write(66, *)  ‘ a teraz drukowanie polowy zmiennej jedenascie od znaku nr 5 do konca’

write(66, fmt=’(a)’)  jedenascie(5:)          !!! wydrukowane  zostanie  56789AB
                                        !!! inaczej  jedenaście (5:11) ;
                                        !!! ale cyfra  11 może być opuszczona;
                                        !!! (5:) oznacza więc, że ostatnia cyfra jest
                                        !!! domyślna;
                                        !!! innymi słowy bierzemy znaki od 5  w
                                        !!! górę tyle, ile się tylko da  (czyli do końca)

                                        !!! uwaga   na format  typu A
                                        !!! (alfanumeryczny)
                                        !!! bez podania liczby
                                        !!! (jak np a6 w poprzednich linijkach)
                                        !!! podając samo    a   informujemy
                                        !!! kompilator, że  należy
                                        !!! wydrukować tyle  liter, ile potrzeba
                                        !!! (komputer sam sobie wyliczy ile trzeba)


nazwisko = ’Nowak’;  imie= ’HIPOLIT’
nazwisko_imie = nazwisko//imie
write(66, fmt=’(a22)’)   nazwisko_imie
                                        !!!  tutaj spotka nas niemiła niespodzianka
                                        !!! mianowicie wszystkie zmienne
                                        !!! uprzednio zadeklarowano jako
                                        !!! 22 znakowe;
                                        !!! oznacza to, że za zmienną nazwisko
```

```
                                        !!! podstawione jest Nowak +  dużo
                                        !!! odstępów;  ponieważ
                                        !!! zlepiamy dwa napisy po 22 znaki a wynik
                                        !!! ma mieć też tylko 22 znaki
                                        !!! brakuje więc miejsca -   druga część
                                        !!! zostanie wogóle pominięta
                                        !!! (po zlepieniu).  Aby temu zapobiec
                                        !!! powinniśmy wyciąć
                                        !!! niepotrzebne odstępy ze zmiennych
                                        !!! nazwisko oraz imie

nazwisko_imie = trim(nazwisko)//’  ’//trim(imie)

                                        !!! funkcja standardowa   trim(..) ; jej
                                        !!! rezultatem jest stała znakowa z
                                        !!! usuniętymi zbędnymi odstępami (po
                                        !!! prawej) ze swojego argumentu,
                                        !!! jedyny istotny odstęp,  którego
                                        !!! potrzebujemy to ten pomiędzy
                                        !!! nazwiskiem i imieniem
                                        !!! dolepiliśmy wiec  go “ręcznie”   jako
                                        !!! napis  ‘  ‘
                                        !!! (składający sie tylko z jednego  odstępu)

!!!  przykłady innych funkcji standardowych działających na napisach:
!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
test = adjustr(nazwisko)                !!! funkcja adjustr  przesuwa grupę
                                        !!! znaczących znaków
                                        !!! w  zmiennej “nazwisko” do prawej;
                                        !!! puste miejsca
                                        !!! powstałe z lewej  wypełniane są
                                        !!! odstępami
test = adjustl(nazwisko)                !!! podobne przesuniecie ale do lewej
!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
dlugosc= len(nazwisko)                  !!! wylicza długość  (ile znaków, łącznie z
                                        !!! odstępami)
                                        !!! jest w zmiennej znakowej   nazwisko;
k = len_trim(nazwisko)                  !!! to samo, ale ewentualne odstępy po
                                        !!! końcu zmiennej
                                        !!! znakowej (z prawej) nie są zliczane
!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
l = index(‘Ala ma kota’, ‘kot’)         !!! jeżeli  sekwencja znakowa ‘kot’
                                        !!! zawiera się w sekwencji znakowej ‘Ala
                                        !!! ……’
                                        !!!  to  index podaje  pozycję  sekwencji
                                        !!! ‘kot’
                                        !!! w naszym przypadku  l = 8
                                        !!! gdybyśmy zamiast ‘kot’ podali  ‘koot’
                                        !!!  wtedy  l= 0   ( gdyż ‘koot’
                                        !!! nie zawiera się w  ‘Ala ….’)
```

```
                                    !!! teraz przerywamy nasz program tuż po
close(66)                           !!! instrukcji zamknięcia pliku
stop;  end program character_demo
```

Obecnie omówimy jeszcze jedną ważną własność stałych i zmiennych znakowych. Otóż każdy pojedynczy znak alfanumeryczny (i znaki specjalne także) mają przypisane odpowiednie numery porządkowe (częściowo zależne jest to od procesora). W standardzie F90/95 przyjmuje się, że pierwsze 26 dużych liter alfabetu (angielskiego) ma uporządkowanie A > B > C > D > …; przed nimi idzie znak odstępu (puste miejsce). Litery małe następują po dużych literach: a > b > c > d > ……Cyfry (jako znaki, a nie jako liczby) są uporządkowane w sposób następujący 0 > 1 >2 > 3 >4 >5 >6 >7 >8 >9 . Cała grupa cyfr występuje albo przed literami, albo po literach, albo w środku pomiędzy małymi a dużymi literami (standard F90/95 pozwala tu na pewną dowolność). Dalsze znaki są uporządkowane już w/g reguł ustalonych przez konkretnego producenta kompilatora. Niezależnie od reguł F90/95 pojedyncze znaki mają własne numery w/g standardu ASCII.

Jeżeli **I** oznacza liczbę całkowitą (oraz **I** jest mniejsze od 128) to funkcja standardowa **achar(I)** zawraca nam znak ASCII o numerze **I**. Dla **I**>128 wynik zależy od konkretnej implementacji kompilatora F90/95, typu maszyny oraz systemu operacyjnego. Na odwrót: jeżeli **Z** jest znakiem w/g standardu ASCII, to **iachar(Z)** dostarcza numeru porządkowego znaku **Z** w/g standardu ASCII.
Podobnie funkcja **ichar(Z)** dostarcza numeru porządkowego jaki został przypisany **Z** w konkretnym kompilatorze, na konkretnej maszynie i systemie operacyjnym.
Tak więc każdy napis można ostatecznie jednoznacznie zamienić na liczbę całkowitą dodatnią (zlepiony ciąg numerów dla kolejnych pojedynczych znaków). Dwie takie liczby dla 2-ch napisów (**napis1, napis2)** można porównać z sobą co odpowiada **alfabetycznemu uporządkowaniu tych** 2-ch napisów. Porządkujemy alfabetycznie w taki sposób że **znak1 > znak2** (czyli znak1 idzie przed znakiem2) jeśli numery porządkowe określone dla znak1 i znak2 spełniają odwrotną relację (I1<I2).

Jeśli porządkujemy dwa napisy w/g standardu ASCII to mamy do dyspozycji logiczne funkcje: **lge, lgt, lle, llt.**
lge(napis1,napis2) daje wartość **.true.** jeżeli **napis1** jest większy lub równy **napis2**
lgt(napis1,napis2) daje wartość **.true.** jeżeli **napis1** jest większy niż **napis2**
lle(napis1,napis2) daje wartość **.true.** jeżeli **napis1** jest mniejszy lub równy **napis2**
llt(napis1,napis2) daje wartość **.true.** jeżeli **napis1** jest miejszy niż **napis2**.

Przykładowo:
lgt(‘Aleksandra’,’Ola’) jest **.true.**
gdyż już dla pierwszej pary liter zachodzi A> O;
lgt(‘ala’,’Ala’) jest **.false.**
gdyż a<A.
Zamiast funkcji standardowych **lgt, lge, llt, lle** możemy używać operatorów **> , >=, <, <=** lecz jedynie w wypadku, gdy kompilator używa reguł uporządkowania w/g standardu ASCII; daje to identyczne rezultaty. Podsumowywując: omówiliśmy narzędzia, które mogą służyć do alfabetycznego sortowania kolumn wyrazów.

I.A.8. Instrukcja pisania (write)

Instrukcje czytania i pisania są czymś absolutnie podstawowym dla każdego języka programowania. Zanim coś wyliczymy musimy zaplanować w jaki sposób będziemy w stanie uzyskać wgląd do naszych rezultatów. Służy do tego głównie instrukcja wyprowadzania danych (pisania) przez komputer. Podobnie nie mniej istotne jest wczytywanie danych startowych do programów.

Zacznijmy od ogólnej postaci instrukcji **write**

write(lista_informacji_kontrolnych) lista_tego_co_piszemy

Lista informacji kontrolnych (oddzielonych przecinkami) może zawierać wiele elementów. Dla większości zastosowań wystarczy poznać tylko te najbardziej typowe.
Podajmy więc przykład:

```
program test_pisanie;  implicit none;
character(len= 40) ::  info= ' plik z danymi = test_pis.dat  '
integer :: j = 1,  k= 222;  real :: f, x = 1.0, y= 0.12345e-15, z= 1000.25, w
real, parameter  :: pi=3.14159
!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
w = k + z*y;    f=   0.123e8;

open(unit=66, file='test_pis.dat')                !!! otwieramy plik (tekstowy), do którego
                                                  !!! będziemy wpisywać  napisy
write(unit=66, *) '  to bedzie pierwszy napis '
write(66, *)    '  to bedzie pierwszy napis '     !!!  skrótowa forma pierwszej instrukcji
!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
write(unit=66, fmt='(''  pi= '', f9.5,  ''  y = '' , e16.5, ''   k*k = '', i9) ')   pi, y, k*k
                                                  !!! w 2-ch następnych liniach będzie to
                                                  !!! samo:
write(66, fmt=4444)   pi, y,  k*k
4444  FORMAT(''  pi= '', f9.5,  ''  y = '' , e16.5, ''   k*k = '', i9)
!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
close(66)                                         !!! instrukcja zamknięcia  pliku
                                                  !!! "test_pis.dat"

                                                  !!! numer  jednostki  -  66  może być od tej
                                                  !!! pory przydzielony
                                                  !!! do zupełnie innego pliku (jeśli tylko go
                                                  !!! otworzymy)
                                                  !!! Równie dobrze możemy używać innego
                                                  !!! numeru z zakresu 10-99
                                                  !!!  -  co rekomendujemy (celem uniknięcia
                                                  !!! pomyłek)
                                                  !!! dla lepszej kontroli wykonywania
                                                  !!! programu   te  same informacje
                                                  !!! drukujemy JESZCZE RAZ  na ekranie
write(unit=*,*) '  to bedzie pierwszy napis '
write(*,fmt=4444)   pi, y,  k*k
                                                  !!!  użycie gwiazdki (jako pierwszego
                                                  !!! argumentu) oznacza jednostkę domyślną
                                                  !!! tzn. drukowanie  na ekranie monitora;
```

!!! przerywamy program

stop; end program test_pisanie

Uwagi dotyczące instrukcji **close** i **open**.

Najczęściej używana komenda do otwarcia pliku to np. open(unit=66, file='nazwa'), a do zamknięcia pliku to po prostu close(66) lub close(unit=66).

Inne (z wielu) możliwości:

open(unit=66, file='nazwa', status='keep')	to samo co open(66, file='nazwa')
open(unit=66, file='nazwa', status='replace')	stary plik o tej samej nazwie zostanie wymazany i powstanie nowy
close(unit=66, status='delete')	zamknij plik i następnie go usuń

Wróćmy teraz do instrukcji **write.** W każdym z 4-ch podanych (powyżej) przypadków lista kontrolna **write** składa się z 2-ch elementów. Pierwszy, to słowo kluczowe **unit** zainicjowane liczbą całkowitą 66 - tym samym numerem kodowym który został nadany jako identyfikator pliku 'test_pis.dat'. Jeżeli ten element listy kontrolnej piszemy jako **pierwszy,** wtedy słowo kluczowe **unit** może zostać pominięte (a efekt będzie taki sam). Podstawienie gwiazdki **unit = *,** lub po prostu samej gwiazdki ***** oznacza urządzenie domyślne (czyli najczęściej ekran monitora). Taki sam efekt (ze względów historycznych) ma podstawienie **unit=6**, gdyż cyfra 6 już w najstarszych wersjach Fortranu oznaczała standardowy (czyli w naszym przypadku domyślny) plik wyjściowy (output).

Drugie słowo kluczowe **fmt** (skrót od **FORMAT**) też może być opuszczone (jeżeli jest dokładnie na drugim miejscu; nie rekomendujemy jednak tej praktyki). Za format podstawiamy stałą znakową (napis) zawierającą sekwencyjną listę zakodowanych (za pomocą skrótów języka angielskiego) informacji o tym jak należy formatować (redagować zapis) poszczególnych elementów **z_listy_tego_co_piszemy**. W skład napisu wchodzą ponadto zamykające go nawiasy okrągłe. W formacie mogą pojawiać się wewnętrzne napisy (ujęte w podwójne apostrofy) , będą one przedrukowywane w takiej samej formie, w jakiej zostały ujęte (w formacie).

Wyjaśnijmy znacznie poszczególnych elementów formatu po kolei:

('' pi= '', f9.5, '' y = '' , e16.5, '' k*k = '', i8)
oznacza, że jako pierwszy mamy przedrukować (bez żadnych zmian) do pliku napis ' pi= ', potem instrukcja write oczekuje, że wydrukujemy liczbę rzeczywistą w formacie **f9.5**, następnie (w tej samej linii) przedrukowujemy napis ' y = ', potem jakąś liczbę rzeczywistą w formacie **e16.5** następnie (ciągle ta sama linia) napis ' k*k= ' w końcu jakąś liczbę całkowitą (tzn. trzecią , kolejną liczbę **z_listy_tego_co_piszemy).**

f9.5 - f to skrót od ang. floating point (liczba zmiennoprzecinkowa) czyli liczba będzie pisana z kropką dziesiętną; **9** oznacza, że na napisanie liczby mamy do dyspozycji 9 miejsc, **.5** oznacza ze po kropce dziesiętnej mamy napisać 5 cyfr znaczących. Dalsze cyfry znaczące są obcinane (z zaokrągleniem). Jeśli 9 miejsc nie wystarczy na napisanie liczby (gdyż jest ona większa) to **write** spowoduje wydrukowanie **9 gwiazdek** dla zaznaczenia, że

zadeklarowany format nie jest adekwatny do napisania danej liczby. Jeśli 9 miejsc to za dużo – zostaną dołożone odstępy (przed drukowaną liczbą).
e16.5 - e oznacza stosowanie notacji inżynierskiej (czyli za liczbą wystąpi np. e11, co oznacza 10 do potęgi 11); **16 –** oznacza że na napisanie całej liczby (łącznie ze znakami, z **e** , z kropką dziesiętną) mamy dokładnie 16 znaków; **.5** jak poprzednio oznacza 5 miejsc po kropce dziesiętnej

I (I - integer) oznacza pisanie liczby całkowitej, na napisanie której przeznaczone jest dokładnie **8** miejsc (pól)

Poszczególne elementy formatu oddzielamy przecinkiem.

Często zdarza się, że format jest bardzo długi - musimy wtedy rozciągnąć pisanie instrucji **write** na dwie linijki względnie zastosować starą formę Fortranu F77 tzn **write(66, fmt=4444) …** lub jeszcze bardziej skrótowo **write(66, 4444) …** gdzie **4444** oznacza tzw. etykietę (**label)** . Etykieta jest jedno÷pięcio cyfrową liczbą całkowitą (z wyjątkiem zera) . W starym Fortranie F77 etykietami można było zaznaczać poszczególne (pojedyncze) instrukcje; w nowym F90/95 etykiety stosujemy raczej rzadko (na ogół nie ma takiej potrzeby). W naszym przykładzie etykieta **4444** (oddzielona przynajmniej jednym odstępem) stoi przed **FORMAT(…).** Instrukcja **write(66, fmt=4444)…**oznacza więc: zamiast brakującego właśnie formatu użyć tego formatu, który zaznaczony został etykietą **4444**. (W programie mogą być setki różnych etykiet oznaczających różne formaty). **4444 Format(…)** może być umieszczony w dowolnym miejscu programu (po zakończeniu deklaracji zmiennych, a przed **end program …).**

Inny przykład:
write(unit=66, fmt='(a40,10x, 2I5, 2(f9.5, e16.6/)) ') info, j, k, x,y, x2,y**2**

Nowe elementy formatu:

10x – informacja, że należy wydrukować 10 odstępów,

/ informacja że należy kontynuować pisanie od początku następnej linii
2(…) - oznacza 2 razy powtórz (w formacie) to samo co jest w wewnętrznym nawiasie (…) .
Tak więc wydrukujemy do pliku oznaczonego jako jednostka 66 wartość zmiennej znakowej info (deklarujemy na to 40 miejsc, format A40), potem 10X oznacza 10 odstępów, następnie dwie liczby integer (po pięć miejsc; 2I5 jest identyczne z I5,I5). Potem x zostanie wydrukowane w formacie f9.5 (9 miejsc łącznie, 5 po kropce dziesiętnej), po nim y w formacie e16.6 (16 miejsc razem, 6 po kropce dziesiętnej, notacja inżynierska) potem nastąpi przejście na początek następnej linii gdzie wydrukowane zostaną wartości x**2 oraz y**2 (w formatach f9.5, e16.6) i znów nastąpi przejście do następnej linii. Tak więc jeśli następną instrukcją jest **write(66, *)** ' ----------' to znaki podkreślenia zostaną wydrukowane po pustej linii.
Uwaga! Każda nowa instrukcja **write** rozpoczyna pisanie od nowej linii (podobnie każda nowa instrukcja **read** czyta dane od nowego wiersza). Tę regułę można ominąć wprowadzając nowy argument do listy kontrolnej **write.** A więc kolejny nowy element spotykamy w:

write(unit=66, fmt=4444, advance='no') x ; 4444 Format(f9.5)

Trzeci argument **write**, oznaczony słowem kluczowym advance (nie można go opuścić) i zainicjowany słowem 'no' oznacza, że zakończenie write NIE SPOWODUJE

automatycznego przejścia do następnej linii; następna instrukcja write będzie więć pisać od tego miejsca (w tej samej linii), gdzie się poprzednia instrukcja **write(…..advance='no')** … zakończyła. Możliwość użycia opcji **advance** (nieznana w starym F77) daje wiele natychmiastowych korzyści -- nawet dla początkującego programisty.

Uwaga! *Opcja advance nie może być stosowana z opcją fmt=* (czyli z formatem domyślnym).*

Jeszcze inne bardzo użyteczne dla programisty narzędzie, to możliwość używania write z opcją **append:**
write(unit=66, fmt=4444, position='append') x
Ta instrukcja będzie normalną instrukcją pisania, ale do pliku już uprzednio istniejącego. Nowe dane zostaną po prostu dołączone po końcu starego pliku. Jeśli plik był pusty nie będzie błędu i '**append**' zostanie po prostu zignorowane.
Przykłady innych opisów w formacie:

2L7 - dwie stałe logiczne (komputer wypisuje **T** lub **F** jako skróty od true lub false); na każdą przeznaczone jest 7 miejsc; litera T lub F dosunięta jest do prawej (z lewej mamy odstępy).
T9 - znak tabulacji; następne dane będą pisane w obecnym wierszu od 9 kolumny
2F16.8 - dwie liczby rzeczywiste (dla każdej - 8 cyfr po przecinku; 16 znaków razem), stanowi to też dobry format dla liczby zespolonej (complex).

E8.2E1 - liczba rzeczywista w formacie inżynierskim; razem 8 znaków; 2 cyfry po kropce dziesiętnej; jednocyfrowa potęga (jeśli mielibyśmy wydrukować liczbę 1.234 to zostanie ona wydrukowana jako **0.12E+1** z dodatkiem jednego odstępu z lewej).

'' to jest napis '' - łańcuch znaków umieszczony w formacie traktowany jest jako stała znakowa którą należy przedrukować dosłownie (ta sama ilość i takich samych znaków).

Na obecnym etapie nauki opuszczamy rzadziej używane w formacie opisy dotyczące wyprowadzenia liczb w układzie binarnym, oktalnym, w hexagonalnym, itp.

Dodatkowe informacje:

Istnieje forma instrukcji **write** z opuszczonym pierwszym argumentem (tylko pierwszym). Jest to **print.** Użycie **print** implikuje, że piszemy na standardowym wyjściu (czyli jest to to samo co write(*,…) ….lub write(6,….)). Instrukcja **print** jest używana dosyć często.

I.A.9. Instrukcja czytania (read)

Instrukcja czytania ma zasadniczo taką samą listę kontrolą jak **write**. Standardowy plik input (wejściowy) może być zdefinowany jako **unit=***, jako sama gwiazdka (tylko na pierwszej pozycji listy kontrolnej), wreszcie (ze względów historycznych) jako **unit=5** (lub samo 5). Podobnie jak **print** stanowi skróconą wersję **write(*,…),** tak **read** bez argumentów, względnie z tylko samym formatem, stanowi skróconą wersję **read(unit=*, ….)**. W piewszym etapie nauki należy jednak unikać posługiwania się tymi skróconymi formami.

Zazwyczaj unikamy stosowania ściśle określonego formatu w **read**. Dlaczego? Konkretne formaty stosujemy najczęściej (bo musimy) tylko do wczytywania ciągu znaków, lub stałych logicznych, a także przy danych składających się z przeplatanych (w jednej linii) danych numerycznych i znakowych. *Jednak dla wczytywania "czystych" danych liczbowych (real, integer, complex, także dowolnej ich mieszaniny) posługujemy się najczęściej formatem domyślnym, czyli gwiazdką*. Komputer sam rozpozna format tych wczytywanych liczb. Zwróćmy uwagę, że odwrotnie było z pisaniem w formacie domyślnym (też gwiazdka). Pisanie w fomacie domyślnym nie jest wygodne, bo komputer zazwyczaj decyduje się nam wydrukować liczbę dokładnie, co oznacza ogromną liczbę cyfr np. 1.0 może zostać wydrukowany jako 0.99999999E+000 . Taka ilość informacji na ogół nie jest nam potrzebna i dlatego używamy formatów wskazujących, ile naprawdę cyfr znaczących i w jaki sposób należy wydrukować.

Stosowanie (przy wczytywaniu) konkretnego formatu do liczb **real,** np. żądanie wczytanie stałej x : **read(unit=*, fmt='(f10.4)') x** oznaczałoby, że wprowadzając **x** z klawiatury musimy zapisać go absolutnie dokładnie w/g następującego przepisu: "liczba musi się kończyć w momencie wprowadzania 10-go znaku z klawiatury; kropka dziesiętna musi być 6-tym znakiem; jeśli liczba ma zbyt mało cyfr, to z lewej musimy dodać odpowiednią liczbę odstępów (aby razem było 10 znaków)". Jest to niewygodne i niepraktyczne jeśli wiemy, że to samo możemy osiągnąć posługując się **read(unit=*, *) x** , przy czym x wprowadzamy z klawiatury jak nam się podoba (bez liczenia co jest w którym miejscu).

Teraz omówimy możliwość uzupełnienia instrukcji read przez jeszcze jeden dodatkowy argument swobodnego wyboru - "opcjonalny" (optional) mianowicie: **iostat= zmienna_integer** (argument ten może pojawić się także w instrukcji **write** -- ale na ogół nie nie jest używany). Jeśli tego argumentu nie ma to błędne użycie instrukcji np.
read(*, fmt='(f10.4)') x i podanie z klawiatury paru liter (zamiast liczby) spowoduje błąd wykonania (egzekucji) programu i przerwanie wykonywania programu. Jeśli **iostat=zmienna_integer** jest jednak obecny to program będzie dalej wykonywany (mimo wystąpienia błędu) ale:

- wykonywanie samej instrukcji read zostanie przerwane;
- celem zasygnalizowania wystąpienia błędu za **zmienna_integer** zostanie podstawiona jakaś niezerowa wartość;
- jeżeli błędu nie było i instrukcja **read** stała wykonana prawidłowo za **zmienna_integer** podstawiana jest wartość zero (co sygnalizuje powodzenie wykonania **read**). ,

Jako **zmienna_integer** zwyczajowo używa się "sugestywnych" nazw zmiennych zadeklarowanych np. jako: **integer :: error, blad, ierrror** itp.

Podamy teraz prosty przykład. Na pliku tekstowym (przyjmijmy, że jest to zbiór 80 znakowych linii zwykłego tekstu) o nazwie "zakupy.dat" mamy dwie kolumny danych: pierwsza kolumna to liczby (kwoty zakupów); druga kolumna to komentarz - napisy określające typ zakupu. Nad danymi mamy "tytuł": linię z tekstem zawierającym informacje (jaki to plik, itp). Dla przykładu, przyjmijmy że zakupy.dat wygląda następująco:

PLIK zakupy.dat (dane o zakupach części zamiennych)
100.0 kable
1267 uklady scalone

35.67 rozne materialy
100.0 katalog
itd

Problem jest następujący: nie wiemy, jaka jest długość pliku (ile pozycji), gdyż każda z osób kupujących w każdej chwili może dokonać nowego zakupu i wprowadzić (za pomocą edytora tekstu) nową linię na końcu "zakupy.dat". To, czego chcemy, to wyliczenie sumarycznych kosztów. Pomoże nam w tym maleńki program posługujący się nową instrukcją **DO** …. **END DO** którą omówimy później bardzo dokładnie, a teraz skorzystamy z niej (bez wyjaśnień) wyłącznie dla lepszej ilustracji "jak posługujemy się **read**"

```
program  sumuj;  implicit none;
integer ::  error;  real ::   suma =0.0,   zakup
character(len=80) :: tytul
open(55,file='zakupy.dat');                !!! otwieramy plik z danymi
read(unit=55,fmt='(a80)') tytul
                                           !!! do zmiennej tytul wczytujemy
                                           !!! komentarze z
                                           !!! pierwszej linii pliku
!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
DO                                         !!! instrukcja  oznaczająca: powtarzaj  w
                                           !!! nieskończoność
                                           !!! to co jest pomiędzy DO   oraz   END DO

 read(55,*,iostat=error)    zakup
                                           !!! wczytanie w swobodnym formacie
                                           !!! jednej liczby;
                                           !!! odstępy  po liczbie sygnalizują jej koniec;
                                           !!! reszta
                                           !!! pozostająca w linii (czyli napisy –
                                           !!! komentarze)
                                           !!! jest wiec  ignorowana; w następnym
                                           !!! wykonaniu
                                           !!! cyklu  DO  będzie czytana nowa linia,
                                           !!! nowa liczba,
                                           !!! a napis (po liczbie) znów będzie
                                           !!! ignorowany.
                                           !!! Uwaga: jeśli w czasie czytania dane się
                                           !!! skończą   (koniec pliku),  to nastąpi
                                           !!!  błąd odczytu; w takim wypadku
                                           !!! przerywamy czytanie:
IF (error .ne. 0)   then
                                           !!! uwaga: instrukcje  if omawiamy dopiero
                                           !!! w rozdziale I.A.10

close(55)                                  !!! zamykamy plik
EXIT                                       !!! instrukcja  EXIT   powoduje
                                           !!! natychmiastowe  przerwanie pętli,
```

```
                                        !!! gdyż error nie równa sie 0, czyli nastapił
                                        !!! błąd odczytu, co
                                        !!!  (jak przypuszczamy) spowodowane jest
                                        !!! brakiem dalszych danych  (koniec pliku)
else
suma=suma+zakup                         !!! error = 0  wiec  nie ma błędu i dalej
                                        !!! sumujemy zakupy
end if

END DO
                                        !!!   teraz  drukujemy dane na ekranie

write(*,fmt='(a80)')   tytul
write(*,fmt='('' suma zakupow wynosi ='', f16.2)')   suma

                                        !!! uwaga: standardowych plików wejścia i
                                        !!! wyjścia nie zamykamy
                                        !!!  (instrukcja close);
                                        !!!  robi to za nas  automatycznie  system
                                        !!! operacyjny komputera
stop
end program  sumuj
```

I.A.10. Instrukcja warunkowa if

Instrukcja logiczna w najprostszym swoim wariancie wygląda następująco:

```
if  (wyrazenie_logiczne) then
instrukcja_lub_blok_instrukcji
end if
```

gdzie wyrazenie_logiczne jest wyrażeniem skalarnym (NIE MOŻE być wyrażeniem macierzowym - wbrew niektórym starszym podręcznikom...). Jeżeli w wyniku wyliczenia wyrażenia logicznego otrzymujemy wartość **.true.,** wtedy wszystkie instrukcje pomiędzy **then** a **end if** są wykonywane; w przeciwnym wypadku – nie są wykonywane.
Proste przykłady:

```
program  if_test
implicit none
real :: x, y;  logical :: logi, b1, b2, b3= .true.
x = 100.0;  b1=.true. ;  b2=.false.

if ( x > 0.0)  then
y = sqrt(x) ;  write(*,fmt='(e18.8)')  y        !!! te instrukcje będą wykonane, gdyż  x>0
end if
!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
if (b3)   then
logi = (x > y)  .eqv.  (b1 .or. b2)
                                        !!! zgodnie z regułami dla priorytetu działań
                                        !!! logicznych oraz relacji arytmetycznych
```

```
                                        !!! nawiasy nie są potrzebne ; logi = x>y
                                        !!! .eqv. b1  jest wiec OK.
                                        !!!  Rekomendujemy jednak używanie
                                        !!! nawiasów, bo  jest to tanie i proste
                                        !!!  rozwiązanie (nie pozostawiające miejsca
                                        !!! na  wątpliwości)
end if
!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!1

if (  b1 .and. b2)  then
                                        !!!  b1 .and. b2   daje  wartość  .false.  –
                                        !!! instrukcja
                                        !!! nie będzie wykonana
write(*,*) ’  ten napis nie zostanie napisany ’
end if
stop
end program if_test
```

Uproszczona forma tej instrukcji stosuje się (wyłącznie) do przypadku, gdy między **then** a **end if** znajduje się *dokładnie tylko jedna instrukcja*. Możemy wtedy opuścić **then** oraz **end if** pod warunkiem, że instrukcja nastąpi od razu po warunku logicznym (w tej samej linii). Drugi przykład z powyższego programu “if_test” można zapisać w taki właśnie sposób:
if (b1 .and. b2) write(*, *) ’ ten napis nie zostanie napisany ’. Ta forma skrótowa jest “zabytkiem historycznym” ze starych wersji Fortranu (nawet starszych niż F77), ale jest tak bardzo popularna, że można ją spotkać w prawie każdym programie.

Bardziej rozbudowana forma instrukcji **if** pozwala na wykonanie pewnego ciągu instrukcji nawet w przypadku, gdy warunek logiczny nie jest spełniony; zaznaczone to jest przez użycie słów **then, else** (ang: then => wtedy, w takim przypadku; else => lub, w przeciwnym wypadku):

```
if ( wyrazenie_logiczne) then
instrukcja_lub_blok_instrukcji
else
inna_instrukcja_lub_bok_instrukcji
end if
```

Przykładowo, zmienimy poprzednio dyskutowany przykład:

```
…..
if (b1 .and. b2) then
write(*,*) ’ ten napis pojawi sie tylko gdy  b1 .and. b2 == .true. ’
write(*,*)  ’ czyli gdy:  b1 == .true. oraz   b2 == .true. ’
else
write(*,*) ’ ten napis pojawi sie natomiast gdy  b1 .and. b2  nie rowna sie .true. ’
write(*,*)  ’ czyli gdy albo :  b1 == .false.  i/albo   b2 == .false. ’
end if
….
```

Uwaga! Wyrażenia (statements) **then, else, end if** muszą być ostatnie w linii (znów zabytek ze starych wersji Fortranu); a **else** oraz **end if** w ogóle muszą być jedynymi wyrażeniami w linii. Jeżeli koniecznie chcemy obejść to wymaganie powinniśmy użyć średników:

if (b1 .and. b2) then; b3= x .gt. y; else; b3= x .eq. y; end if

Istnieje jeden wyjątek od reguły: " else musi wystąpić w linii jako ostatni". Dotyczy on konstrukcji złożonych ("wielopiętrowych") instrukcji if: po else może bezpośrednio występować kolejna instrukcja warunkowa.

Bez podawania schematu "formy ogólnej" przejdźmy od razu do przykładów. Przyjmijmy, że sprawdzamy, czy jakaś liczba jest podzielna przez 4; a jeśli jest, to czy jest również podzielna przez 100. Ten wąski i dosyć "sztuczny" problem jest częścią innego większego problemu (kalendarz...) , lecz w tej chwili przyjmujmy, że testujemy poprawność tylko tego naszego "wąskiego" problemu. Zastosujemy w tym testowym programiku dzielenie całkowite (liczba_calkowita / liczba_calkowita = liczba_calkowita z odrzuceniem reszty) dla sprawdzania podzielności:

```
program blok1_if
implicit none;  integer :: rok
logical :: podzielny4, podzielny 100
write(*,*) ' podaj z klawiatury  wartosc   zmiennej   rok '
read(*,*)  rok                                  !!! używamy swobodnego formatu przy
                                                !!! wczytywaniu
if ( (rok/4)*4  .ne. rok)  then
                                                !!!  jeżeli dzielenie całkowite: rok/4 daje
                                                !!!  liczbę
                                                !!!  całkowitą bez reszty,
                                                !!!  to wartość (rok/4)*4 odtwarza wartość
                                                !!!  zmiennej rok
                 podzielny4 = .false.  ;   podzielny100 = .false.
else  if ( (rok/100)*100  .ne. rok )  then
                podzielny4 = .true. ;  podzielny100=.false.
else
               podzielny4 = .true. ;   podzielny100=.true.
end if
!!!!!!!!!!!!!!!!!!!!
write(*, 4)  podzielny4;  write(*,100) podzielny100;
   4 FORMAT('' podzielny4 ='', L4)
100  format('' podzielny100='', L4)
                                                !!! zadaliśmy formaty za pomocą etykiety  4
                                                !!! oraz  etykiety 100  (label)
stop
end program blok1_if
```

Po raz pierwszy (przy pisaniu if…) zastosowaliśmy tutaj redakcję tekstu z "wcięciami", co nie jest wcale jakimś sztywnym obowiązkiem, ale zdaniem większości programistów bardzo polepsza czytelność pisanych programów.

Podamy jeszcze dwa dodatkowe przykłady (już bez umieszczania ich wewnątrz małych, "kompletnych" programików):

```
……
if (rok  .eq. 2000 )  then
          write(*,*)  ’  to jest koniec wieku’
else if (rok> 1900 .and.  rok<2001)  then
          write(*,*)  ’  to jest XX wiek’
end if
….
```

Po wyrażeniu **else if** nastąpiła tutaj forma z **then, end if**, bez drugiego else.

```
….
if (rok > 1900 .and. rok <2001) then
       if (rok>1950) then
       write(*,*) ’ druga polowa 20 wieku’
       else
       write(*,*) ’ pierwsza polowa 20 wieku’
       end if
else if (rok>1800) then
      write(*,*) ’ to jest w wieku XIX’
else if (rok>1000 .and.  rok < 1101)  then
      write(*,*) ’ to jest w wieku XI’
else
    write(*,fmt= (’’  nie sprawdzamy w ktorym wieku jest rok=’’, i5)’)  rok
end if.
```

Takie wielopiętrowe instrukcje warunkowe są używane bardzo rzadko, gdyż prawie zawsze można je zastąpić bardziej elegancką instrukcją **case.** Przy występowaniu dużej liczby if’ów łatwo o pomyłkę: można np. zapomnieć o napisaniu jednego z zamykających end if itp. Radzimy, żeby stosować wtedy pisownię z odpowiednim wcinaniem tekstu. Druga możliwość unikania pomyłek, to stosowanie tzw. “nazwanych” konstrukcji **if**, czyli stosowanie *słownych etykiet*. Etykiety te powinny być literowo-cyfrowymi identyfikatorami (te same zasady co przy deklarowaniu zmiennych) i nie mają nic wspólnego z etykietami służącymi np. do identyfikacji instrukcji **FORMAT**. Wstawiamy je przed odpowiednie **if** i kończymy znakiem dwukropka. Ponadto nazwa etykiety musi zostać powtórzona bezpośrednio po **end if** (już bez dwukropka).

Przykładowo “wielopiętrowy” blok if z poprzedniego przykładu może wyglądać następująco, jeżeli użyjemy etykiet o nazwach poziom1, poziom2 :

```
poziom1: if (rok > 1900 .and. rok <2001) then
       poziom2: if (rok>1950) then
       write(*,*) ’ druga polowa 20 wieku’
       else
       write(*,*) ’ pierwsza polowa 20 wieku’
       end if  poziom2
else if (rok>1800) then
      write(*,*) ’ to jest w wieku XIX’
else if (rok>1000 .and.  rok < 1101)  then
      write(*,*) ’ to jest w wieku XI’
else
    write(*,fmt= (’’  nie sprawdzamy w ktorym wieku jest rok=’’, i5)’)  rok
end if poziom1;
```

Nie mamy obowiązku nazwania każdego bloku **if ... end if**; możemy to robić wybiórczo, np. w podanycm przykładzie można opuścić słowa poziom2: oraz poziom2; pozostawiając poziom1: oraz poziom1. (Można też zrobić na odwrót). Możliwość nazwania bloków if stanowi nieocenioną pomoc przy uruchamianiu programów – w istocie nie jest niczym innym jak dodatkowym (przejrzystym) mechanizmem sprawdzania poprawności naszego kodu źródłowego.

Zasygnalizujemy już teraz, że bloki **select case ... end select** oraz bloki **do.... end do** mogą być nazywane w identyczny sposób.

I.A.11. Instrukcja warunkowa case

Case (ang: przypadek, wypadek; the case in point: rozważany przypadek) jest instrukcją warunkową o znacznie większych możliwościach niż ma je instrukcja **if.** Zaczyna się od wyrażenia **select case(argument_case)** gdzie **argument_case** jest zmienną lub wyrażeniem o wartości:

- albo całkowitej (integer);
- albo znakowej (jeden lub ciąg znaków, czyli napis);
- albo logicznej.

(ostatniego przypadku nie omawiamy, gdyż faktycznie dubluje on przypadek instrukcji if i z tego względu na ogół nie jest często używany). Następnie po **select case(argument_case)** następuje lista (dowolnie długa); pojedynczy element tej listy to wyrażenie **case(konkretna_wartosc_argumentu_case)** (napisane w osobnej linii lub zakończone średnikiem) za którym następuje dowolna liczba instrukcji aż do wystąpienia kolejnego wyrażenia **case(konkretna_wartosc_argumentu_case).** Nie ma obowiązku wyczerpywania listy (tzn. wymieniania wszystkich konkretnych wartości jakie może przybrać **argument_case)**. Koniec listy sygnalizujemy wyrażeniem **end select,** lub **case default; instrukcje;… ; end select.** Zanim przejdziemy do przykładu zróbmy degresję słownikową: select - wybierz; default – słowo z punktu widzenia polskiego wieloznaczne; oznacza m.in. brak lub niepowodzenie; in default of .. oznacza " z braku….", judgement by default, oznacza wyrok (sądowy) zaoczny; w informatyce default oznacza ustalenie domyślne ("zaoczne"); w instrukcji **case default** stanowi odpowiednik **else** czyli wystąpienie default po liście **select,…** rozumiemy jako "dla przypadków pozostałych, dotychczas niewymienionych". Przejdźmy teraz do przykładu.

```
program   dobre_rady
implicite none;
integer ::  miesiac, dzien, ile;
write(*,*) ' jaki jest dzisiaj dzien miesiaca ??'
read(*,*)  dzien
!!!
write(*,*) '  który mamy miesiac ?? ';  read(*,*) miesiac

if (miesiac<1 .or. miesiac>12) then
write(*,*) '  blad:  liczba nie miesci się w przedziale 1-12 ; stop'; stop
end if
if (miesiac .eq. 2 .and. dzien>29) then
write(*,*) ' blad: nie ma takiego dnia; stop' ; stop
end if
```

```
!!!!!!!!!!!!!!!!!!!!!!!!!!
select case(dzien)
case(1)
write(*,*)  ' pierwszy dzien miesiaca;  odebrac pensje '
case(13);   write(*,*) ' pechowy dzien, zostac w domu !!!'
case(15)
write(*,*)  ' od jutra oszczedzac pieniadze !!!! '
end select
!!!!!!!!!!!!!!!!!!!!!!!!!!
stop
end program dobre_rady
```

Konkretna wartość argumentu występującego w **case** nie musi być pojedynczą wartością. Możemy również jako argumentu używać podzbiorów, lub sekwencyjnych podzbiorów, co kodowane jest wg następującego prostego schematu:

case(1:4)	-	od 1 do 4 (co jeden)
case(6,7,9)	-	6, 7 oraz 9
case(1:4,6,7,9)	-	od 1 do 4 (co jeden) oraz 6, 7, 9
case(:5)	-	argument mniejszy lub równy 5 (od minus "nieskonczonosci" do 5)
case(6:)	-	argument równy 6 lub większy

*Uwaga: w konkretnym bloku **select case** poszczególne przypadki **case** muszą być jednoznaczne, tzn. nie mogą się przekrywać, np. wystąpienie dwa razy case(1), case(1) nie jest prawidłowe !!!.*

Kolejny przykład (użyjemy nazwanego bloku **select case**, oraz opcji **case default**):

```
.....
rady_nr2:  select case(miesiac)
case(1,2,11,12);   write(*,*)  '  nie zapomnij szalika  !!!'
case(3:5);         write(*,*)  '  teraz jest wiosna, uwaga na katar !!!'
                   ile= miesiac –6 ;   write(*, 1234) ile
                   1234 format('' do lata jeszcze'', i3,''  miesiecy !)
case(6:9);         write(*,*)  '   jest cieplo; pora na sporty letnie'
case(10);          write(*,*)  '   juz jesien...'
case default                                  !!! pozostale przypadki tzn  miesiac < 1 lub
                                              !!! miesiac>12
write(*,*)  '  oszukujesz  !!!  nie ma takiego miesiaca '
end select  rady_nr2
....
```

Dla porządku podamy teraz przykład z użyciem napisu

```
......
character(len=2) :: dzien, miesiac;    character(len=1):: min='-'
character(len=4) :: rok;  character(len=10) :: data
.....
dzien =  '03';  miesiac = '05'; rok='2000';
.....
select case(dzien//min//miesiac//min//rok)     !!! jako argument case
```

```
                                        !!! mamy teraz
                                        !!! wyrażenie o wartości znakowej
case('03-05-2000'); write(*,*)  ' swieto narodowe !!!'
case('01-01-2001'); write(*,*)  '  mamy juz wiek XXI !!!'
end select
!
data = dzien//min//miesiac//min//rok
!
select case(data(1:3))
                                        !!! ten napis jest częścią oryginalnego 10
                                        !!! znakowego napisu  (tylko litery 1, 2, 3)
case('10','20','30');   write(*,*) '  dosyc okragla data...'
case('25': );  write(*,*) ' do konca miesiaca juz niewiele dni '
end select
....
```

Uwaga! Podobnie jak dla liczb całkowitych użycie konstrukcji **''25'':** oznacza napis '25' i napisy następne (w/g alfabetycznego porządku; porównaj -->rozdział o zmiennych znakowych A.7).

Na zakończenie podkreślmy jeszcze raz, że instrukcja **select case... end select** występuje nadzwyczaj często w różnych programach, że jej użycie jest bardzo proste i że naprawdę warto jej używać.

I.A.12 . Pętle, czyli instrukcja DO

Instrukcja powtarzania (kilka razy) bloku innych instrukcji nazywa się pętlą. W F90/95 najbardziej typowa pętla wygląda następująco:

```
integer :: j, wartosc_startowa=-4, wartosc_koncowa=6,  increment=3
!!!!!
  DO   j=  wartosc_startowa, wartosc_koncowa, increment
  blok_instrukcji

  END DO
```

gdzie **wartosc_startowa, wartosc_koncowa, increment** przyjmują wartości całkowite, są zmiennymi lub wyrażeniami o wartości **integer** (*ponadto są stałe w czasie wykonywania pętli; a więc między innymi są niezależne od j*). **Wartosc_koncowa** rozumiana jest jako wartość nieprzekraczalna; licznik może być równy wartości końcowej (za ostatnim razem) ale nie może jej przekroczyć**. Increment** nie może być zerowy, ale może być ujemny.

Kontrolna zmienna j (odgrywa rolę licznika) zmienia się w czasie wykonywania pętli. Przy pierwszym wykonaniu **bloku_instrukcji** jej wartość (w powyższym przykładzie) wynosi -4, przy drugim wykonaniu jest powiększona o 3 (czyli j=-1); przy trzecim znów o trzy (j=2); przy trzecim wykonaniu j=5. Czwartego razu już nie będzie gdyż wartość 5+3 jest większa od końcowej wartości, określonej jako 6. Jeżeli **wartość incrementu** wynosi dokładnie 1, to można go opuścić (wartość jeden przyjmowana jest wtedy domyślnie).

```
write(*,*)  ' drukujemy aktualna wartosc j '

DO  j= 3, 12;   write(*,fmt='(i3)') j;     END DO

                                          !!! wydrukowana zostanie  kolumna

                                          !!! liczb (jedna pod druga):  3,4,5,….,12
```

Uwaga!

```
DO   n=3, 0, 2
blok_instrukcji
END DO
```

W tym przykładzie **blok_instrukcji** nie będzie w ogóle wykonany ze względu na źle określoną wartość końcową. Jedynym efektem tej pętli będzie podstawienie za n wartości 3, co nastąpi przed wykonaniem (przez program) testu: czy w ogóle, oraz ile razy należy powtórzyć pętlę.

Prostym i intuicyjnie jasnym zastosowaniem pętli jest sumowanie. Przyjmijmy, że mamy wyliczyć sumę $S= a_1 + a_2 + a_3 + \ldots + a_{100,}$ gdzie j-ty wyraz sumy zadany jest formułą $a_j = 1/(1+2^j)$. Programik liczący wartość S może wyglądać następująco:

```
program sumuj
implicit none;
integer :: j;    integer, parametr :: DP=selected_real_kind(15);
real(kind=DP) :: s, a_j, potega
s=0.0_DP;   potega= 1;
   do j=1,100
   potega=potega*2.0_DP                  !!! rekurencyjne wyliczanie  wyrażenia
                                         !!! 2**j  co  jest
                                         !!!  obliczeniowo tańsze  od użycia:
                                         !!! potega=2**j
   s = s + 1.0_DP/(1.0_DP + potega)
   end do
write(*,fmt='( ''  suma='', E18.8E1)')   s
stop
end program sumuj
```

Nieco bardziej skomplikowany przykład pętli wyglądać może następująco:

```
integer :: i, k=5, suma=0
DO  i=  k+1, -k**2, -(2*k +4)             !!!  (a więc wartości: 6, -25, -14)
suma=suma + i
end do
```

Pętla będzie wykonana dla i= 6, -8 , -22 natomiast dla i=-36 już nie, gdyż przekroczyliśmy zakres ustalony przez **wartość_końcową** (czyli -k**2 ; więc -25).

Pętle, oczywiście, mogą być zagnieżdżone jedna w drugiej, mogą także być "nazwane" (za pomocą etykiet słownych – identycznie jak dla bloków **if**). Zilustrujemy to

wykonując sumę podwójną: $S = a_1 + a_2 + a_3 + \ldots + a_{100}$ gdzie ogólny j-ty wyraz jest dany przez inną sumę $a_j = j + (j+2) + (j+4) + \ldots + (j+10)$:

```
program suma2
implicit none;
integer:: j, n,  a_j;  integer,parameter :: prec=selected_real_kind(15)
real(kind=prec)  :: s
petla_j: do  j=1,100;

a_j =0
        petla_n : do  n= j, j+10, 2
        a_j = a_j + n
        end do petla_n
s=s+a_j
end do petla_j
write(*, fmt='(''suma ='',  E18.8)') s
stop
end program suma2
```

I.A.13. Awaryjne przerywanie wykonywania pętli - instrukcje EXIT oraz CYCLE

Rozkaz **EXIT** powoduje natychmiastowe przerwanie wykonywanej pętli – można to sobie wyobrazić jako skok tuż poza **END DO.** Rozkaz **CYCLE** powoduje przerwanie danej iteracji – można go sobie wyobrazić jako skok w miejsce tuż przed **END DO**. Po tym skoku licznik ulega inkrementacji i cykl jest nadal powtarzany. Bardzo często instrukcje **EXIT, CYCLE** stosowane są z trzecią (dotychczas jeszcze nie omówioną) formą pętli, która wygląda następująco:

```
DO
blok_instrukcji

END DO
```

Nie widzimy tutaj żadnego licznika, żadnych wartości początkowych, końcowych i "incrementu". Ta forma pętli, teoretycznie rzecz biorąc, wykonywana jest w nieskończoność aż do momentu, gdy w jakiś awaryjny sposób wymusimy jej przerwanie. Dobrym przykładem będzie obliczanie sumy szeregu nieskończonego: $S= a_1 + a_2 + a_3 + a_4 + \ldots.$ gdzie j-ty wyraz dany jest wzorem $a_j= 1/(2**j + 1)$. Umawiamy się, że sumę S przybliżymy przez skończoną sumę wyrazów większych niż epsilon=10^{-12} . W chwili, gdy kolejne wartości a_j spadną poniżej progu "epsilon" będziemy je ignorować. Osoby, które przeszły kurs analizy matematycznej będą w stanie stwierdzić, że nasza prymitywna reguła w zastosowaniu do niniejszego przypadku jest sensowna.

```
program nieskoczona_suma
integer :: j;  integer,parameter :: DP=selected_real_kind(15)
real(kind=DP) :: suma, a_j, epsilon
epsilon =1.0e-12_DP;  suma=0.0_DP
DO
      a_j = 1.0_DP/(2.0_DP**j + 1.0_DP)
```

```
      suma=suma+a_j
      if (a_j<epsilon) EXIT                    !!!  skrócona forma  if
END DO
write(*,*)  '  suma  w przyblizeniu = ', suma
stop
end do nieskonczona_suma
```

Inny przykład: wyliczyć sumę : S= 1+2+3+4+5+ ….. + 300. Sumujemy kolejne liczby (co jeden), ale dodatkowo zażądamy, by opuścić w tej sumie liczby: 55, 122 oraz 250. Użyjemy do tego instrukcji “cycle”:

```
program  cycle_test
integer ::  suma, n
suma=0;  n=0;
do;   n=n+1;                               !!! n posłuży nam jako licznik zmieniający
                                           !!! się od 1 do nieskończoności
     if (n>300) EXIT
     if (n .eq. 55 .or. n .eq. 122 .or. n .eq. 250)   CYCLE
     suma=suma+n
end do;
write(*,*) ' suma=',suma
 stop
end program cycle_test
```

Jeżeli pętle są zagnieżdżone w sobie, to **EXIT** lub **CYCLE** działa tylko w odniesieniu do najbliższej sobie (obejmującej je) pętli. Nie będziemy uczyli się ani rozważali tej reguły, gdyż istnieje prostszy sposób postępowania. W nazwanych (słownymi etykietami) pętlach możemy po **EXIT** podać nazwę etykiety. Wtedy skok będzie dotyczył pętli zaopatrzonej tą właśnie słowną etykietą, jak w przykładzie poniżej:

```
….
petla_po_n :   do  n=1,1000;
                    …….
                     do j=n, n**2 + n + 1
                     …….
                     if ( j > 500) EXIT  petla_po_n
                    end do
                ……..
end do petla_po_n                          !!! skok  (spowodowany EXIT) do tego
                                           !!! miejsca
….
```

Podsumowywując : **EXIT** stosować będziemy do awaryjnego przerywania pętli, **CYCLE** jest wygodny do uwzględniania wyjątków (np. wtedy, gdy pewne wartości zmiennej sterującej, czyli licznika, mają zostać opuszczone).

I.A.14. Macierze (array)

Macierz określamy jako jedno, dwu lub wielowymiarową tablicę liczb typu real, integer, complex. Elementy macierzy mogą też być stałymi znakowymi, lub logicznymi. Jeszcze inna możliwość, dla elementów będących tzw. typami pochodnymi (inaczej złożonymi) zostanie omówiona dużo później.

Konkretna wartość elementu macierzy określona jest przez wartości indeksów. Np. kwadratowa (dwuwymiarowa) macierz 10 x 10 liczb rzeczywistych o nazwie A ma elementy określane przez A_{ij} , indeksy i, j zmieniają się każdy od 1 do 10; indeks i numeruje wiersze tablicy, j numeruje kolumny. Inny przykład, to jednokolumnowa 3-elementowa macierz B o elementach $B_{j,}$ gdzie j=1,2,3 określa numer wiersza (kolumna jest tylko jedna). W F90/95 można deklarować macierze mające do 7-miu indeksów.

W F90/95 jest określonych 5 typów macierzy. Najprostszy to **explicite shape array,** co rozumiemy jako macierze, co do których podaliśmy liczbę indeksów i ich zakresy (od – do). Mamy spore wątpliwości, czy tłumaczenie tego terminu na język polski może być zarówno językowo eleganckie, jak i oddać niuanse znaczenia tego terminu. Będziemy posługiwać się więc terminem angielskim oraz pesudo polskim - macierz o "kształcie zadanym jawnie". Macierz (explicte shape array) deklarujemy dodając do deklaracji zmiennej atrybut **dimension** z zaznaczonym (w nawiasie) zakresem zmienności indeksów, przy czym zakres ten zaznaczamy przez dwie liczby całkowite oddzielone dwukropkiem; np. 2:10 oznacza zmienność indeksu od 2 do 10 (co jeden). Możliwe jest także użycie pojedynczej liczby, np. 10, co traktowane jest (domyślnie) jako 1:10. Do jednego konkretnego elementu macierzy odwołujemy się pisząc identyfikator macierzy z nawiasem, w którym podajemy konkretne (interesujące nas) wartości poszczególnych indeksów.

W przykładzie poniżej zadeklarujemy trzy kwadratowe macierze **logical** o nazwach k1, k2, k3 o wymiarach 10 x 10 ; macierz **logical** L_{ij} gdzie i, j = 11,12,…, 20; trzy wektory (macierze jednowymiarowe) **integer** w, u, v o zakresie zmienności indeksu od 7 do 9, pięć macierzy trójwskaźnikowych **real** R_{ijk}, T_{ijk}, Y_{ijk}, …. o zakresie zmienności indeksów i=1, 2, ..., 10; j= -30,..., 30; k= 0,...,10; dwie macierze **complex** Cij**,** Zij, gdzie i= 1,…,10; j= 1,…,10 oraz macierz jednowymiarową znakową (od 1 do 3) "napisy". Dla macierzy k1, k2 podstawimy .true. dla elementów diagonalnych oraz .false. dla elementów pozadiagonalnych, za składowe w, u, v wartości 1, 2, 3; macierz R wyzerujemy, macierz T oraz Y wypełnimy jedynkami, a za macierz Z podstawimy jednostki urojone dla pierwszego wiersza, a wartość jeden dla pozostałych wierszy. Na zakończenie wydrukujemy wartości macierzy do pliku tekstowego. W poniższym przykładzie postaramy się przekazać możliwie dużo informacji o deklarowaniu i posługiwaniu się macierzami.

```
program  macierze
implicit none;
integer, parameter :: St=8;   integer ::  r,s
logical, dimension(10,10) :: k1, k2              !!! zakres indeksów:  od  1 do 10
logical, dimension (11:20,11:20) : L             !!! zakres indeksów od 11 do 20
integer, dimension(7:9) :: w, u, v               !!! zakres indeksow:  od 7 do 9

real, dimension( 2:10, -30:30, 0:10)  ::  R, T, Y, TplusY, TmnozY

complex, dimension(1:10, 2+St)::  Z, C           !!! zakres zmienności indeksu może być
                                                 !!! wyliczony;  a więc 2+St  daje  10

character(len=10),dimension(1:3) :: napisy
```

```
do  r=1,10;  do s=1,10;
    if (r .eq. s) then
    k1(r,s) = .true.                    !!! za diagonalne (r=s) elementy k1
                                        !!! podstawiamy .true.
    else
    k1(r,s)=.false,                     !!!  za pozostale  (r .ne. s)  podstawiamy
                                        !!! .false.
    end if
end do; end do;

k2= k1                                  !!! macierze o  identycznym kształcie
                                        !!! możemy podstawiać jedna za drugą

L   =  k1                               !!! ta instrukcja też jest poprawna, gdyż
                                        !!! zakres zmiennosci każdego indeksu
                                        !!! obejmuje identyczną liczbę elementów;
                                        !!! mówimy, że kształty  L  oraz  k1 są
                                        !!!  takie same  mimo, że granice, pomiedzy
                                        !!! którymi zmieniają się indeksy, są
                                        !!!  różne  (dla k1 od 1 do 10  dla L  od 11 do
                                        !!! 20); dokladny sens instrukcji
                                        !!! podstawienia   L = k1
                                        !!! zostanie przedstawiony za chwilę poniżej
                                        !!! (za pomocą pętli do);
                                        !!!  wygląda to tak, jak gdyby  dla  maszyny
                                        !!! naturalny  zakres
                                        !!! zmienności indeksów zawsze  zaczynał
                                        !!! się  od   1 ;
                                        !!!  w takiej “interpretacji” maszyna  może
                                        !!! dodatkowo  obsługiwać drugi
                                        !!! równoważny system numeracji
                                        !!! (ten zadeklarowany przez nas); nasze
                                        !!! własne wartości indeksów są wtedy
                                        !!! po prostu przesunięte względem
                                        !!! ’’naturalnego indeksowania’’ maszyny

do r=11, 20;  do s=11, 20;
     L(r,s) =  k1(r-10, s-10)
end do; end do                          !!! identyczne  z   L = k1
k3 = k1 .and. k2                        !!! dozwolona instrukcja oznaczająca: dla
                                        !!! każdej pary indeksów
                                        !!! wykonaj to samo, tzn.   k3_ij = k1_ij
                                        !!! .and. k2_ij
                                        !!! uwaga : k3_ij   oznaczamy (w niejszym
                                        !!! komentarzu)
                                        !!! jako  element macierzy k3 o indeksach
                                        !!! i, j  (podobny sposób zapisu występuje  w
                                        !!! programie TeX)
!!!
do  r=7,9;   w(r)= r;  end do;
```

```
u = w
                                    !!! identyczny skutek osiagnąć można
                                    !!! stosując tzw. konstruktor macierzy,
                                    !!! czyli po prostu wymieniajac  w
                                    !!! odpowiednich nawiasach jej listę
                                    !!! elementów

v = (/7, 8, 9/)
!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
R  =  0.0                           !!! za macierz możemy podstawić liczbę;
                                    !!! rozumiane jest to następująco:
                                    !!! “za każdy element  R  podstaw tę samą
                                    !!! liczbę”
T = 1.0;  Y = 1.0;
                                    !!! macierze  T,  Y  wypełniamy jedynkami

TplusY =  T + Y                     !!! znów dozwolona instrukcja oznacza: dla
                                    !!! każdej trójki indeksów
                                    !!!  ijk   zrób to samo:
                                    !!!TplusY_ijk = T_ijk + Y_ijk

TmnozY = T*Y                        !!! nie jest to wcale iloczyn macierzy ale
                                    !!! instrukcja oznaczająca
                                    !!!   TmnozY_ijk = T_ijk*Y_ijk    (dla
                                    !!! każdej trójki  indeksów  i, j, k)
!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!

Z  =  (1.0, 0.0)                    !!! za wszystkie elementy  Z podstawiamy
                                    !!! liczbę zespoloną równą  1

do r=1, 10;   Z(1,r) =  (0.0,1.0);    end do;
                                    !!! pierwszy wiersz wypełniamy liczba
                                    !!! urojona   (0,1)
                                    !!! identyczny skutek możemy osiągnąć
                                    !!! stosując instrukcje

Z(1, :) = (0.0, 1.0)
                                    !!! Z(1, :)  rozumiana jest (podobnie jak
                                    !!! było to dla napisów)
                                    !!! jako macierz jednowymiarowa
                                    !!! powstała z macierzy kwadratowej Z
                                    !!!  i  obejmująca tylko elementy jej
                                    !!! pierwszego wiersza ;
                                    !!! dwukropek oznacza więc pełny zakres
                                    !!! zmienności  drugiego indeksu

C  =  (2.0, 0.0) * Z   +  (1.0,  0.0)
                                    !!!  interpretacja:    C_ij = 2*Z_ij  + (1,0)
                                    !!! dla każdej pary  i, j;
                                    !!!  tak wiec w F90/95 macierz można
```

```
                                        !!! mnożyć przez liczbę,  co oznacza
                                        !!! mnożenie każdego elementu macierzy
                                        !!! (przez tę liczbę)

napisy(1) = ' 1-wszy napis';            !!! tutaj średnik nie jest potrzebny; ale
                                        !!! też nie  przeszkadza
napisy(2) = ' 2-gi     napis';
napisy(3) = '  3-ci     napis';
                                        !!! to samo  można by było osiagnąć stosując
                                        !!! konstruktor macierzowy

napisy = (/'1-wszy napis', '2-gi   napis', '3-ci   napis'/)
                                        !!! teraz otwieramy plik do drukowania
open(88, file=' przyklady.txt')
write(88, *)  '  drukujemy przyklady z programu   macierze :'
write(88, *)  ' ------------------------------------------------------'
write(88, fmt='('' element  k1_23  macierzy  k1='', L8)')   k1(2,3)
write(88, *) ' elementy macierzy  k1 ; drukowane w/g kolejnosci kolumn'
write(88, *) ' czyli kolejno elementy pierwszej kolumny, drugiej kolumny, trzeciej…'
write(88, fmt='(10L7)')  k1             !!! drukujemy po 10  elementów  na wiersz
                                        !!! uwaga:  jeśli argumentem write jest
                                        !!! macierz  (sam symbol bez indeksów)
                                        !!! to wydrukowane zostaną wszystkie
                                        !!! elementy macierzy  w kolejności
                                        !!! naturalnej (w tym wypadku w/g
                                        !!! kolejności kolumn)
!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
write(88, *) ' teraz drukujemy elementy macierzy R  (w kolejnosci naturalnej)'
write(88, fmt='(5E16.7)')  R
                                        !!!
                                        !!! uwaga: kolejność naturalna oznacza w
                                        !!! tym przypadku   takie ułożenie
                                        !!! elementów   R_ijk, że   i   zmienia się
                                        !!! pierwsze, potem j, a na końcu k, a więc
                                        !!! mamy po kolei trojki    (1,-30,0), (2,-
                                        !!! 30,0), (3,-30,0) ,….(10,-30,0), (1,-29,0),
                                        !!! (2,-29,0),……(10,-29,0),…….
                                        !!!  w tym  miejscu przerywamy
                                        !!!
close(88);   stop;
end program macierze
```

I.A.15. Segmenty (części) macierzy

F90/95 jest bardzo rozbudowanym językiem jeżeli chodzi o działania na **całych** macierzach lub ich segmentach (częściach).

Do określania o jakie segmenty macierzy chodzi służy specjalna konwencja "trójek indeksów " (subcript triplets). Jeżeli w miejsce jakiegoś indeksu macierzy (poprzednio zadeklarowanej) zapiszemy trzy liczby oddzielone dwukropkami, np. 10:20:5, to

rozumiemy to następująco: dokonaj selecji indeksów, zacznij od wartości 10, zmieniaj indeks co 5 (trzecia liczba tzn. inkrement lub powiększenie) tak długo, aż nie przekroczy wartości 25 (drugi liczba); przykładowo

real, dimension(0: 100) :: A

…….

A(10:25:5) = 0.0

oznacza: weź segment macierzy składający się z elementów A_{10} , A_{15} , A_{20}, A_{25}.

Identyfikator A(10:25:5) oznacza więc jednowymiarową i czteroelementową macierz. Podstawienie A(10:25:5) = 0.0 oznacza, że elementy A(10), A(15), A(20), A(25) oryginalnej macierzy zostaną wyzerowane. Konwencja trójek ma liczne warianty i uproszczenia; ich znaczenie (w oparciu o powyższą deklarację macierzy A) wyjaśnia tabela poniżej:

A(5:50)	to samo co 5:50:1 (elementy od numeru 5 aż do 50, co jeden)
A(5:)	to samo co 5:100:1 (elementy od numeru 5 do "końca" , co jeden)
A(5::5)	to samo co 5:100:5 (elementy od 5 do "końca", co pięć)
A(:25)	to samo co 0:25:1 (elementy od "początku" do 25 , co jeden)
A(:25:5)	to samo co 0:25:5 (elementy od początku do 25, co pięć)
A(::5)	to samo co 0:100:5 (elementy od początku do końca, co pięć)
A(:)	to samo co 0:100:1 (elementy od początku do końca , co jeden czyli *wszystkie elementy A*

Konwencja trójek stosuje się, oczywiście, niezależnie dla każdego indeksu macierzy. Tak więc, jeżeli mamy deklarację **character(len=25), dimension(10,10) :: info** to info(:,:) jest identyczne z info (z całą macierzą); info(5,:) oznacza macierz jednowymiarową której elementami są elementy 5-go wiersza macierzy info, natomiast info(1:5,1:5) oznacza lewy górny kwadrant (ćwiartkę) macierzy info, itd.

Istnieje jeszcze jeden dosyć wyszukany sposób określania "segmentów" macierzy za pomocą tzw. wektorowego indeksu (vector subscript). Zamiast definicji znów posłużymy się przykładem

integer, dimension(2) :: index1; integer, dimension(6) :: wsk

real,dimension(4) :: W

….

w=(/1.0, 2.0, 3.0, 4.0/);

index1 = (/ 2, 4/); wsk=(/2, 2, 4, 4, 1, 3/)

…

write(*, fmt='(2F10.4)') W(index1)

write(*, fmt='(6F10.4)') W(wsk)

…

gdzie użyliśmy konstruktora macierzowego (/…./) celem podstawienia za elementy macierzy konkretnych liczb. Po podstawieniu "wektorowych indeksów" index1 oraz wsk, W(index1) będzie identyfikatorem macierzy (dwuelementowej) składającej się z W_2, W_4 natomiast W(wsk) będzie identyfikatorem 6-cio elementowej macierzy : W_2, W_2 , W_4, W_4 , W_1, W_3 (uwaga: każda z wartości macierzy wsk musi się mieścić w zakresie zmienności indeksu macierzy W; tzn. jedynie liczby 1,2,3,4 są dozwolone). Mieszanie trypletów z indeksami wektorowymi jest możliwe, co demonstrujemy poniżej:

```
integer, parameter :: zero=0
integer, dimension(5) :: wek_ind = (/6, 7, 8, 9, 10/)
                                   !!! macierz można inicjować już w czasie
                                   !!! deklaracji
integer, dimension(10, 10) ::  moje_dane
….
moje_dane(wek_ind, 6:10) = zero
moje_dane(wek_ind,1) =zero
```

W podanym przykładzie w pierwszym podstawieniu (assigment) wyzerowaliśmy prawą dolną ćwiartkę macierzy moje_dane. Oczywiście, taki sam efekt osiągnęlibyśmy stosując podstawienie: moje_dane(6:10,6:10) = zero. W drugim podstawieniu wyzerowaliśmy dolną połowę pierwszej kolumny macierzy moje_dane.

Zwracamy uwagę początkujących programistów na okoliczność, że możliwość posługiwania się segmentami macierzy (bez deklarowania nowych macierzy na zmagazynowanie wybranych elementów) jest bardo użyteczna. Powoduje to, między innymi, znaczne skrócenie naszego kodu źródłowego (np. w przypadku dużych programów operujących na macierzach: takich jak rozwiązywanie różnych problemów algebry liniowej, a w szczególności wyliczanie wyznaczników, odwracanie macierzy itp).

I.A.16. Konstruktory macierzy

Dla macierzy jednowymiarowych (z jednym indeksem) poznaliśmy już (w rozdziałach A.14 oraz A.15) konstruktor macierzy. Przypomnijmy więc, że:

```
integer, dimension(0:4) :: M
M=(/1, 2, 3, 4, 5/)
```

oznacza to samo co

M(0)=1 ; M(1)=2; M(2)=3; M(3)=4; M(4)=5;

Konstrukor macierzy może przybierać również inne (rzadziej używane) formy zawierające skrótowo zaznaczoną pętlę DO, co zapisuje się jako (/ (j, j=1, 5) /). Efekt podstawienia **M=(/1, 2, 3, 4, 5/)** jest więc identyczny, gdy użyjemy **M=(/ (j, j=1, 5) /).** Bardziej skomplikowany przykład konstruktora macierzy z ukrytą pętlą Do spodowuje podstawienie za kolejne elementy M kolejnych liczb nieparzystych : 1, 3, 5, 7, 9; zakodujemy to jako **M=(/ (2*j+1, j=0, 4) /).** Używanie mieszanej konwencji, czyli wykazu elementów, oraz pętli Do także jest możliwe. Przyjmijmy np., że mamy macierz 100-elementową, w której początkowe 10 elementów to liczby parzyste, ostatnie 3 elementy mają wartości 98, 99, 100 , a wszystkie pozostałe elementy są równe 333. Wtedy:

integer :: j, k
integer, dimension(100) :: przyklad
przyklad=(/ (2*j, j=1,10), (333, k=11, 97), 98, 99, 100/)

Trochę inaczej postępujemy gdy macierz jest wielowymiarowa. Dla przykładu rozważmy macierz dwuwymiarową. Postępowanie w takim wypadku jest następujące:

- tworzymy macierz jednowymiarową zawierającą elementy macierzy dwuwymiarowej poukładane kolumnami;
- do utworzonej macierzy jednowymiarowej stosujemy stardardową funkcję **reshape** która macierz jednowymiarową przekształci w dwuwymiarową.

Przyjmijmy więc, że chcemy utworzyć następującą macierz o wymiarach 5 x 2:

$$C=\begin{bmatrix} 0 & 5 \\ 1 & 6 \\ 2 & 7 \\ 3 & 8 \\ 4 & 9 \end{bmatrix}$$

Dokonamy tego w sposób następujący:

integer, parameter :: ilosc_wierszy=5, ilosc_kolumn=2
integer, dimension(5, 2) :: C
integer, dimension(0:4, 0:1) :: C2
integer, dimension(10) :: pomocnicza
integer, dimension(2) :: zakresy_indeksow
pomocnicza= (/0, 1, 2, 3, 4, 5, 6, 7, 8, 9/)
zakresy_indeksow = (/5, 2/)
C=reshape(pomocnicza, zakresy_indeksow)
……
lub po prostu
C=reshape((/0, 1, 2, 3, 4, 5, 6, 7, 8, 9/), (/5, 2/))
podobnie

C2 = reshape((/0, 1, 2, 3, 4, 5, 6, 7, 8, 9/), (/5, 2/))
jest identyczna z macierzą C (z wyjątkiem przesuniętej numeracji indeksów).

W praktyce najczęściej jednak elementy macierzy wpisywać będziemy do osobnego pliku tekstowego, z którego później wczytamy je do programu (do odpowiedniej macierzy) za pomocą instrukcji **read.**

I. A.17 Funcje standardowe lbound, ubound, size oraz shape

Jak wspominaliśmy już wcześniej, w F90/95 można deklarować macierze mające do 7-iu indeksów. Wymiar macierzy, czyli liczba indeksów nazywa się inaczej rangą (ang.: rank). Np. macierz R_{ijkl} jest 4-o wymiarowa, bo ma 4 indeksy (rank=4). Liczba elementów przypadająca na ustalony indeks nazywy zakresem tego indeksu (ang: extent of a dimension) - jest ona równa górna_granica_indeksu – dolna_granica_indeksu+1. Rozmiarem macierzy (ang.: size) nazywamy całkowitą liczbę elementów macierzy. W końcu kształtem (ang.: shape) macierzy nazywamy jednowymiarową macierz, której kolejnymi elementami są zakresy kolejnych indeksów. W języku polskim użylibyśmy raczej słowa wymiary macierzy zamiast kształt. Ale pozostańmy w obszarze “slangu” angielsko-informatycznego; nie jest to

może eleganckie, ale wydaje się być bezpieczeniejsze (dla osoby właśnie się uczącej). Przykładowo określimy, jakie są ranga, zakresy, rozmiar, kształt (rank, extent, size, shape) dla

real, dimension(2:5, 1:10, 0:9) :: D

Łatwo sprawdzić że rank =3; extent's są 4,10,10 czyli shape zadany jest macierzą (/4,10,10/) ; size = 400. Dolna granica pierwszego indeksu wynosi 2, górna 5; dolna granica drugiego indeksu wynosi 1, górna 10; dla trzeciego odpowiednie liczby to 0, 9.

Wspomnieliśmy, że każda macierz posiada coś w rodzaju naturalnej numeracji, gdzie każdy indeks rozpoczyna się od wartości 1. Dlatego możemy wykonywać podstawienie następującego typu:

real, dimension(2:5, 1:10, 0:9) :: D
real, dimension(1:4, 1:10, 1:10) :: D_prim
…
D_prim = D

Widzimy, że macierze D oraz D_prim mają identyczny kształt, czyli że są zgodne ze sobą (po ang. określamy to jako *conformable arrays*, czyli że są *zgodne*). Pojęcie zgodności nieco rozszerzymy przyjmując, że np. z dowolną macierzą **real** zgodna jest również dowolna liczba **real** przy czym rozumiemy wtedy, że liczba została powielona w nową macierz o tym samym kształcie. Na macierzach zgodnych możemy wykonywać podstawianie jednej macierzy za drugą, dodawanie, oraz "mnożenie", traktowane zawsze jako blok identycznych operacji na pojedynczych elementach

Tak więc

Operacja	wynik operacji
A = (/1,2,3/)	A(1)=1, A(2)=2, A(3)=3
(/1,2,3/) + (/10,20,30/)	(/11,22,33/)
(/1,2,3/) * (/10,20,30/)	(/1*10, 2*20, 3*30/)
10 * (/1,2,3/)	(/10*1, 10*2, 10*3/)

gdzie A została zadeklarowana jako **integer, dimension(3) :: A**

Uwaga! Wbrew niektórym starszym podręcznikom instrukcje typu **if (A .eq. B) then,... gdzie A, B** to macierze o tym samym kształcie, ***NIE SĄ DOZWOLONE*.**

Jeszcze jeden przykład:

```
integer :: r, s
integer, dimension(10, 10) :: A, C
integer, dimension(0:9, 11:20) :: B
integer, dimension(20, 20) :: CC
B = 10;  A = 1
C =  B – 2* A;     CC(1:10, 1:10) = C
                                          !!! co jest identyczne z:
do r=1, 10; do s=1, 10;   A(r, s) = 1;  B(r-1, s+10) = 10;
end do; end do;
                                          !!!
do r=1, 10; do s=1, 10;
C(r, s) + B(r-1, s+10) – 2* A(r, s);  CC(r, s) = C(r, s)
end do; end do;
….
```

Fortran 90/95 posiada na swoim wyposażeniu funkcje, które dokonują skanowania macierzy podając jej główne charakterystyki. W głównym bloku programu funkcje te nie są zbyt użyteczne, ale w programach składających się z dziesiątków różnych bloków ich rola staje się ważna. A więc **size(macierz)** podaje liczbę elementów macierzy. Funkcję tę można stosować ponadto w drugi sposób **size(macierz, numer_indeksu)** - podaje ona wtedy liczbę (extent) elementów wzdłuż określonego wymiaru. Następnie funkcja **shape(macierz)** daje w wyniku jednowymiarową macierz charakteryzującą kształt macierzy. W końcu **lbound, ubound** (lower bound, upper bound, czyli dolna granica, górna granica) określają zakres zmienności określonego indeksu. Przykładowo:

```
integer ::  ind1_dolny, ind1_gorny,  ind2_dolny, ind2_gorny, rozm,  &
            zakres1, zakres2
integer, dimension(2) :: kszt
real, dimension( 0:4, 6) :: M = 1.0          !!! macierz M wypełniona jest jedynkami
…
                                             !!!  w wyniku otrzymamy:
rozm = size(M                                !!!  rozm = 5*6 = 30
zakres1 = size(M,1)                          !!!  zakres1 = 5
zakres2=size(M,2)                            !!!  zakres2 = 6
ind1_dolny =  lbound(M,1)                    !!!  ind1_dolny = 0
ind1_gorny = ubound(M,1)                     !!!  ind1_gorny = 4

ind2_dolny =  lbound(M,2)                    !!!  ind2_dolny = 1
ind2_gorny = ubound(M,2                      !!!  ind2_gorny = 6

kszt = shape(M                               !!!  kszt = (/5,6/)
……
```

Obecnie przejdziemy do omówienia innych ważnych funkcji standardowych.

I.A.18. Funkcje standardowe działające na macierzach

Funkcji standardowych, operujących na macierzach, jest zbyt wiele, żeby omówić je wszystkie. Poprzestaniemy więc jedynie na tych, które stosowane są najczęściej. Po pierwsze, macierze, których elementami są typy podstawowe (intrinsic types, a więc integer, real, complex) mogą być argumentami standardowych funkcji takich, jak logarytm, sinus, cosinus, wartość bezwzględna itp. Interpretowane jest to jako : wylicz sinus (cosinus,….) dla każdego elementu macierzy z osobna; z wyników utwórz macierz o takim samym kształcie. Dla przykładu:

```
integer ::  r, s
integer, parameter :: DP= selected_real_kind(15);
real(kind=DP), dimension(2:3,2:3) :: arg, wynik
arg =  1.0_DP
wynik = log(arg)
                                             !!! identyczny rezultat osiagnac można za
                                             !!! pomocą pętli:
do r=2,3; do s=2,3;  wynik(r,s) = log(arg(r,s)); end do; end do;
 …..
```

Podobnie zamiast funkcji **sin** moglibyśmy użyć: cos, tan, ich funkcji odwrotnych (arcusów) czyli **acos, asin, atan**, następnie funkcji hiperbolicznych **sinh, tanh**, funkcji

sqrt, log, abs, exp, funkcji **conjg** (funkcja sprzężenia zespolonego), funkcji **aimag** (wyciąga część rzeczywistą z liczb zespolonych) oraz funkcji **max, min**, wyciągające wartość maksymalną lub minimalną:

```
integer, dimension(2,2) ::  A,B,C,  D_maxi, D_mini;
A(1,1)= 1; A(1,2)=  2; A(2,1)=3; A(2,2)=  4;
B(1,1)= 1;  B(1,2)=-2; B(2,1)=3; B(2,2) = -4;
C = A + B                          !!! czyli elementy (ułożone wierszami) są
                                   !!! równe    2, 0;  6, 0

D_maxi = MAX(A,B);                 !!! elementy  D_maxi (ułożone wierszami)
                                   !!! są  równe    1, 2;  3, 4

D_maxi = MAX(A,B,C)                !!! elementy  D_maxi (ułożone wierszami)
                                   !!! są 2,2; 6, 4

D_mini = MIN(A, B, -2)             !!! elementy  D_mini (ułożone wierszami) są
                                   !!! -2,-2; -2, -4;
                                   !!! uwaga:  w ostatnim przykladzie liczba –2
                                   !!! jest zgodna z A, B ,  gdyż jest
                                   !!! interpretowana jako macierz o
                                   !!! elementach -2, -2; -2, -2 (ten sam kształt:
                                   !!! 2 x 2) ;
……
```

Podobnie podstawowe funkcje działające na argumentach znakowych (napisach -- “string’ach”) mogą jako argumenty mieć całe macierze znakowe.

Nadzwyczaj użyteczną klasą funkcji są te, które dokonują standardowych numerycznych operacji algebry macierzowej. Są to:

- **dot_product** - iloczyn skalarny 2-ch zgodnych (co do kształtu) macierzy jednowymiarowych czyli wektorów;
- **matmul** - prawdziwy iloczyn macierzowy albo 2-ch macierzy dwuwymiarowych - prostokątnych (o ile liczba kolumn pierwszej = liczbie wierszy drugiej), albo macierzy dwuwymiarowej i wektora, albo wektora i macierzy dwuwymiarowej (podobne zastrzeżenia co do odpowiedniej liczby wierszy, kolumn aby mnożenie macierzowe było określone);
- **transpose** - transpozycja macierzy dwuwymiarowych (zamiana wierszy na kolumny)

```
integer :: r, s, t
real :: suma, dlugosc2, dlugosc4, iloczyn_skal
!!!
real, dimension(2, 4) :: M2x4
real, dimension(4, 2) ::  M4x2
real, dimension(2, 2) ::  MxM
real, dimension(2) :: vector2 , wynik2
real, dimension(4) :: vector4, wynik4
real, dimension(4) ::   new_vec = (/ 1.0, 2.0, 3.0, 4.0/)
….
vector4 = 10.0;  vector2 = 4.0
                                   !!!
```

```
M2x4(1, :) = 1.0;                          !!! elementy I-go wiersza  = 1
M2x4(2, :) = 2.0;                          !!! elementy II-go wiersza = 2
M4x2 = transpose(M2x4)                     !!! elementy M4x2  wykaz w/g  wierszy:
                                           !!! 1, 2;  1, 2;  1, 2; 1, 2 ;
                                           !!!
iloczyn_skal = dot_product(vector4,new_vec) !!! iloczyn_skal =  100
                                           !!! to samo moglibyśmy wyliczyć stosujac
                                           !!! pętle, a więc:
suma=0.0
do r=1, 4;  suma=suma+ vetor4(r)*new_vec(r); end do
iloczyn_skal= suma;
                                           !!!
                                           !!! kwadraty długości wektorów liczymy też
                                           !!! za pomocą dot_product
dlugosc2 = dot_product(vector2,vector2)    !!! wynik =  32
dlugosc4 = dot_product(vector4,vector4)    !!! wynik =  400

                                           !!! teraz iloczyny:
 MxM=  matmul(M2x4, M4x2)
                                           !!! to samo w starym stylu (F77) uzyskuje
                                           !!! się stosując potrójną pętlę jak następuje
do r=1, 2; do s=1, 2;
      suma=0.0;
      do t=1, 4; suma=suma+ M2x4(r, t)*M4x2(t, s); end do;
     MxM(r, s) = suma
end do; end do;
                                           !!! elementy MxM ułożone wierszami są:
                                           !!! 4, 8;  4, 16;
                                           !!!  inne możliwości zastosowania  matmul
                                           !!! wyglądają następująco:
wynik2 = matmul(M2x4, vector4)             !!!  elementy  --> 40, 80
wynik4 = matmul(vector2,M2x4)              !!!  elementy  -- > 12, 12, 12, 12, 12
                                           !!!
                                           !!! nielegalne uzycie:  matmul( M2x4,
                                           !!! M2x4)   <-  ERROR !!!!
                                           !!! nielegalne uzycie:  matmul( M2x4,
                                           !!! vector2)  <-  ERROR
                                           !!! podobnie
                                           !!! matmul( vector4, M2x4)  <-  ERROR
…
```

I.A.19. Instrukcje i funkcje standardowe działające na macierzach – ciąg dalszy

Zanim omówimy pozostałe funkcje standardowe działające na macierzach musimy określić pojęcie maski logicznej. Nazwa “maska” jest tutaj czysto symboliczna. Wyobraźmy sobie, że mamy tablicę 8 x 8 składającą się z kwadratów (np. szachownicę) o jednym kolorze. Chcemy zamalować na inny kolor jedynie pewne wybrane pola. Sporządzamy więc maskę z tektury o wymiarach identycznych jak nasza tablica; wycinamy okienka w odpowiednich miejscach maski, przykładamy maskę do tablicy i skierowywujemy rozpylacz

z farbą na maskę. Wycięte okienka pozwolą na wybiórcze pokrycie farbą wybranych obszarów (analogiczna technika stosowana jest przy produkcji układów scalonych).

Jeżeli chcemy dokonać wybiórczej operacji tylko na niektórych elementach jakiejś macierzy możemy sporządzić maskę - logiczną tablicę tego samego kształtu (o tych samych wymiarach) co oryginalna macierz. Operacji na macierzy dokonujemy element po elemencie ale tylko wtedy, gdy odpowiednie elementy maski mają wartość .true.

Najprostszy przykład: w macierzy SYG(100) zgromadziliśmy kolejne (czasowe) wartości jakiegoś sygnału cyfrowego. Idealnie powinna być to sekwencja zer i jedynek. Ze względu na błędy transmisji, na szumy itp. dostajemy inny ciąg dodatnich liczb: np. 0.1, 1.1, 0.2, 0.9, ….. Przyjmujemy, że małe liczby to najprawdopodobniej zniekształcona wartość zero. Duże liczby to z kolei zniekształcona jedynka. Przyjmiemy więc (dosyć dowolnie), że oczyszczenie sygnału będzie wyglądać następująco. Progiem będzie 0.5; liczby mniejsze od 0.5 przekształcimy w zero; liczby większe od 0.5 przekształcimy na jedynki. Nasza maska, to D>0.5 . Pamiętamy, że liczby są zgodne z macierzami (w sensie kształtu macierzy). Innymi słowy D>0.5 oznacza macierz logiczną, której pierwszym elementem jest SYG(1)>0.5; drugim elementem jest SYG(2) >0.5; trzecim elementem jest SYG(3) > 0.5 itp. Wartości tych elementów, to .true. albo .false. w zależności od tego, czy nierówność jest spełniona czy nie. Gdybyśmy oczyszczenie naszego sygnału próbowali zrobić metodą tradycyjną (tak jak robiło się to w F77), to musielibyśmy użyć pętli DO:

```
real, dimension(100) :: SYG,  JEDYNKI =0.0
integer::  j, k
open(55, file='dane.syg');
                                        !!! w pliku  dane.syg przechowywane są
                                        !!! wartości sygnału – jedna liczba na jedną
                                        !!! linię;  razem 100 liczb i 100 linii

do j=1, 100;   read(55, *)  SYG(j); end do;

do k=1, 100
   if (SYG(k)  > 0.5 ) then
        SYG(k)=1.0
   else
       SYG(k) =0.0
  end if
end do;
```

Na modłę F90/95 identycznie tego samego (co robiła powyższa pętla po k) możemy dokonać używając krótkiej i zwięzłej instrukcji **where**:

```
where (SYG >0.5)
SYG =1.0
elsewhere
SYG = 0.0
end where
```

Instrukcja **where** występuje w trzech wariantach (podobnie jak instrukcja **if**). Jeden wariant właśnie poznaliśmy. Drugi wariant wygląda następująco:

```
JEDYNKI = 0.0
```

where (SYG > 0.5)
JEDYNKI=1.0
end where

Trzeci wariant to po prostu jeszcze bardziej skrócony zapis drugiego wariantu:

where (SYG>0.5) JEDYNKI=1.0

*Uwaga! W instrukcji **where** maska logiczna musi być zgodna z macierzą (conformable with) którą podstawiamy. Dozwolone jest tylko jedno podstawienie pod symbol macierzy po **where** i jedno po **elsewere** (żadnych indeksów; żadnych dodatkowych instrukcji). Ranga macierzy może być oczywiście dowolna.*

Obecnie po instrukcji **where** poznamy bardzo wygodne funkcje **SUM** oraz **PRODUCT.** Naszym przykładem będzie teraz macierz integer 31 x 100 o nazwie WYDATKI. W macierzy tej gromadzić będziemy zapisy o naszych wydatkach z budżetu domowego. Jeżeli WYDATKI(3,8) = 500 to umawiamy się, że oznacza to 8-y z kolei wydatek (w wysokości 500 zł) dokonany 3-go dnia obecnego miesiąca. Jeżeli wydatków było tego dnia tylko 8, to pozostałe wartości macierzy są zero, czyli WYDATKI(3,9:100) = 0. Teraz sformułujemy nasz problemik:

- ile pieniędzy wydaliśmy 3-go dnia miesiąca?
- ile pieniędzy wydaliśmy w ciągu całego miesiąca?
- ile pieniędzy wydaliśmy w ciągu całego miesiąca ale tylko w formie drobnych wydatków, tzn. kwot nie przewyższających 20 zł?

integer, dimension(31,100) :: WYDATKI
integer :: mala_suma, cala_suma, suma_3
!!! wczytujemy z pliku wartości wydatków
mala_suma = SUM(WYDATKI, WYDATKI<20)
cala_suma = SUM(WYDATKI)
suma_3 = sum(WYDATKI(3, :))

Jak widać, **SUM** może mieć dwa argumenty (faktycznie może mieć ich nawet trzy, ale te przypadki wykraczają poza zakres niniejszego skryptu). Pierwszy obowiązkowy argument, to jakaś macierz (o dowolnym kształcie). Gdy w **SUM** jest tylko ten jeden argument, wtedy **SUM** wylicza sumę **wszystkich** elementów macierzy. Tak więc **SUM(WYDATKI)** wylicza sumę elementów całej macierzy a **SUM(WYDATKI(3, :)** sumę segmentu macierzy WYDATKI (segment to cały 3-ci wiersz macierzy WYDATKI). Jeżeli jednak mamy drugi argument – naszą maskę logiczną to sumowane są tylko te elementy dla których odpowiedni "równoległy" element maski ma wartość .true. **SUM(WYDATKI, WYDATKI < 20)** zsumuje więc tylko te elementy macierzy które są mniejsze od 20. Funkcja **PRODUCT** działa analogicznie do **SUM;** tyle tylko, że podaje nie sumę, ale iloczyn elementów.

Na zakończenie podamy jeszcze kilka innych funkcji. Podobnie jak **SUM** mogą one mieć jeden, dwa lub czasami trzy argumenty, i w zależności od tego ich znacznie jest nieco inne. My ograniczymy się jedynie do najprostszych przykładów – co więcej, podamy je tylko w formie schematycznej. Przyjmijmy więc, że zadeklarowane mamy dwie macierze rzeczywiste i jedną logiczną:

real,dimension(100) :: dluga
real, dimension(10, 10) :: mat; logical, dimension(10, 10) :: maska

oraz kilka zmiennych pomocniczych (o nazwach : licznik, nr_indeksu, warunek)

integer :: licznik; logical :: warunek;

Funkcja **ALL**
warunek = ALL(maska) ma wartość .true. tylko wtedy, jeżeli wszystkie elementy maski mają wartość .true. (All – wszystko).
Funkcja **ANY**
warunek = ANY(maska) ma wartość .true. tylko wtedy jeżeli chociaż jeden element **maska** ma wartość .true. (Any - jakikolwiek, jakiś)
Funkcja **COUNT**
licznik = COUNT(maska) podaje ile razem elementów z **maska** ma wartość .true.

Następnie omówimy funkcję macierzową **PACK,** która pakuje elementy macierzy o randze większej niż jeden (macierz wielowymiarowa, z kilkoma indeksami) w odpowiednio długiej macierzy jednowymiarowej. Kolejność składowania jest naturalna, co dla macierzy **mat** oznacza składowanie kolumnami (pierwsza, druga, trzecia,… kolumna). Tak więc **dluga = PACK(mat).**

Instrukcji Pack możemy używać w inny bardziej skomplikowany sposób. Jeżeli mamy na przykład dodatkowo 100-u elementowy wektor o wartościach zerowych **(integer, dimension(100) :: zero_vector=0),** to możemy spakować tylko te elementy **mat,** dla których dodatkowo "równoległy" element macierzy **maska** jest .true. Ponieważ spakowanych elementów będzie mniej niż 100, brakujące miejsca zapełnione zostaną końcowymi elementami macierzy zero_vector. Wygląda to następująco:
dluga=PACK(mat,maska,zero_vector)
a więc mamy teraz do czynienia aż z trzema argumentami.

Jako kolejną ciekawostkę wspomnimy funkcje **maxloc, minloc.** Przyjmujmy, że zdefiniowaną mamy jednoelementową macierz: **integer, dimension(1) :: indeks1**

indeks1 = Maxloc(dluga) - "lokalizacja maximum" - podaje wartość indeksu (jako wartość indeks1(1)) dla którego mamy do czynienia z największym elementem **dluga.** Jeżeli zdefiniujemy dwuelementowy wektor **integer, dimension(1:2) :: indeksy** oraz napiszemy **indeksy = maxloc(mat)** to wartości elementów **indeksy(1), indeksy(2)** dają nam tę parę indeksów **mat,** które określają największy element **mat. MINLOC** stosuje się do lokalizacji minimalnego elementu macierzy. Zarówno w **MAXLOC,** jak i w **MINLOC** możemy za **mat** umieścić dodatkowy (opcjonalny) argument **maska**. Wtedy otrzymamy lokalizację minimum lub maximum, ale pod kontrolą maski (tylko dla zbioru elementów dla których wartość **maska** jest .true.).

Zakończymy wprowadzając ciekawą macierzową funkcję **CSHIFT** (circular shift). Opiszemy tylko najprostszą wersję jej zastosowania do macierzy jednowymiarowych.
CSHIFT(dluga, 12) powstaje z macierzy **dluga** na której przesunięto o 12 pozycji wszystkie elementy. Elementy, których numer przewyższył dolny (1) idą na koniec, te które przekroczyły górny zakres (100) idą na początek. Tak więc element nr. 22 macierzy **dluga**

staje się elementem nr 10 macierzy **CSHIFT(dluga, 12);** element nr. 2 macierzy **dluga** staje się elementem nr 90 macierzy **CSHIFT(dluga, 12),** itd.

I.A.20. Kilka przykładów prostych programów

Zakończyliśmy omawianie najprostszych elementów języka F90/95, jakie możemy napotkać w programie składającym się z jednego bloku (**program** …. **end program**). Warto w tym miejscu podać kilka dodatkowych przykładów na których możemy jeszcze raz przećwiczyć zdobyte w rozdziałach I.A.1 – I.A.19. umiejętności:

1. Regresja liniowa

Jest to prosty problem, z którym można spotkać się np. na I pracowni fizycznej. Mamy N danych pomiarowych w postaci dwukolumnowej tabelki (x_1, y_1), (x_2, y_2), ….., gdzie indeks przebiega od 1 do N. Przewidujemy, że na wykresie punkty te leżą (z małymi odchyłkami spowodowanymi błędami pomiarowymi) wzdłuż prostej $y = ax + b$. Nie znamy wartości współczynników a , b ale chcemy je wyznaczyć. Wzory:

$$S_x = \sum_i x_i$$

$$S_y = \sum_i y_i$$

$$S_{xx} = \sum_i (x_i)^2$$

$$S_{xy} = \sum_i (x_i y_i)$$

$$S_{yy} = \sum_i (y_i)^2$$

$$a = (NS_{xy} - S_x S_y)/(NS_{xx} - S_x^2)$$

$$b = (S_y - aS_x)/N$$

pozwalają wyliczyć takie wartości a oraz b, aby prosta odtwarzała punkty doświadczalne w sposób możliwie wierny.

Program regresja wczyta dane doświadczalne z pliku tekstowego ’dane.xy’, wyliczy wartości a, b i wydrukuje je do pliku ’regresja.out’:

```
program regresja
implicit none;
integer, paramater ::  N_max= 1000          !!! przyjmujemy, że  N nie będzie większe
                                            !!! niż 1000
integer :: N,  j
real, dimension(N_max) :: x, y
real ::  a, b,  suma_y =0.0, suma_x=0.0, suma_xx=0.0, suma_xy=0.0
character(len=5) :: napis_a =  ’ a =’,  napis_b= ’ b=’

open(55, file=’ dane.xy’)
```

```
read(55, *)  N                              !!! w pierwszej linii  dane.xy  mamy
                                            !!! wartość  N
do j=1, N
read(55, *)  x(j), y(j)
end do;     close(55)

suma_x    = sum(x(1:N))                     !!! używamy funkcji macierzowych
suma_y    = sum(y(1:N))
suma_xx  = dot_product(x(1:N), x(1:N))      !!! używamy iloczynu skalarnego
suma_xy  = dot_product(x(1:N), y(1:N))

a= (N * suma_xy – suma_x*suma_y)/(n*suma_xx – suma_x**2)

b=(suma_y – a* suma_x)/n

open(66, file='regresja.out');

write(66, *) '  dopasowywanie prostej  y = ax+b  do danych doswiadczalnych'

write(66, *) ' ----------------------------------------------------------------------------------'

write(66, fmt='('' liczba  punktow dosw  N= '', i4)')  N

write(66, fmt='( A5, F15.4, 5x, A5, F15.4)')  napis_a, a, napis_b, b

close(66);  stop

end program regresja
```

2. Szukanie zer funkcji metodą bisekcji

Następny przykład: rozwiązywanie równania nieliniowego x = cos(x) dla x>0. Definiujemy funkcję f(x) = x-cos(x). Dla x1=0 f(x1) = -1. Dla x2=π/2 f(x2)= x2>0. Widać więc, że szukane zero funkcji f(x) leży wewnątrz przedziału [x1,x2]. Najprymitywniejsza (nieefektywna, bardzo powolna, ale pewna) metoda numeryczna szukania miejsca zerowego polega na ustawicznym dzieleniu przedziału [x1,x2] na połowę. Niech punkt x_srodek leży w środku [x1,x2]. Wyliczymy wartość f_srodek = f(x_srodek). Jeżeli f_srodek > 0 , to przesuwamy x2 w nowe miejsce (nowe x2 = x_srodek). Jeżeli f_srodek<0, to przesuwamy x1 w nowe miejsce (nowe x1 = x_srodek). Jak widać, długość przedziału [x1,x2] ustawicznie zmniejsza się, dążąc do zera. Po 100 – 150 iteracjach możemy powiedzieć, że aktualny x_srodek dobrze przybliża położenie zera funkcji f (z dokładością absolutną rzędu jednej 0.01). Jeżeli mamy szczęście i w czasie wyliczeń przypadkowo f(x_srodek) = 0, to możemy iteracje zakończyć od razu.

```
program   szukanie_zera

implicit none;
real, parameter:: pi=3.14159
real :: x1=0.0,  x2=  pi/2.0,  x_srodek,  f_srodek,   epsilon=0.01
!!!!!!!!!!!!!!!!!!!!!!!!!
                                            !!! posłużymy się  nazwaną (etykieta
                                            !!! iterujmy)  nieskończoną pętlą DO
iterujmy:   DO                              !!! ustawicznie  powtarzamy, co następuje:
```

```
x_srodek = (x1+x2)/2.0;      f_srodek = x_srodek – cos(x_srodek)

if (f_srodek  .eq.  0.0)   EXIT iterujmy          !!! zero znalezione; kończymy
if (f_srodek > 0.0) then

        x2=x_srodek                               !!! nowa wartość x2
else

       x1 =x_srodek                               !!! nowa wartość  x1
end if

if ( (x2-x1)  <  epsilon)  EXIT   iterujmy        !!! przybliżona lokalizacja zera znaleziona
END DO iterujmy


x_srodek = (x1+x2)/2.0

write(*, fmt='('' przyblizone miejsce zerowe   = '', F15.4)')   x_srodek

write(*, fmt='('' blad jest mniejszy niz = '', F10.4)')  epsilon

stop

end program   szukanie_zera
```

Uwaga: w powyższym przykładzie zastosowaliśmy nieskończoną pętlę DO oraz instrukcję EXIT.

3. Wyliczenie wartości szeregu macierzowego

$$WYNIK = E + \sum_{n=1}^{\infty} A^n / n!$$

gdzie A to kwadratowa macierz 2 x 2, a E to macierz jednostkowa. Oczywiście, sumować do nieskończoności nie możemy – umówimy się że przerwiemy sumowanie dla takiego n, gdy największy element macierzy A^n będzie mniejszy niż pewna określona liczba (np. epsilon = 1.0E-8). Uwaga:

- ze względu na możliwość sporych błędów zaokrągleń musimy stosować podwyższoną precyzję obliczeń;
- problemem zbieżności szeregu nie będziemy się zajmować.

Problem ten wydaje się sztuczny, ale podobne szeregi możemy spotkać w klasycznej mechanice teoretycznej i w mechanice kwantowej.

```
program  szereg_macierzowy
implicit none;
integer, parameter :: DP= selected_real_kind(15)
integer :: n,  N_maxi=100;
real(kind=DP) :: epsilon= 1.0E-8_DP,  mianownik
real(kind=DP), dimension(2, 2) :: E, A,  A_n, WYNIK

E = reshape( (/1.0_DP,  0.0_DP,  1.0_DP,  0.0_DP/),  (/2, 2/) )
open(35, file='macierz.A')                    !!! w pliku tekstowym (jedna linia)
```

```
                                    !!! ułożone są kolumnami
                                    !!! cztery elementy macierzy A
read(35, *) A                       !!! wczytanie całej macierzy
close(35)
                                    !!! przygotowywujemy wartości startowe:

WYNIK = E;  n = 0;     A_n = E ;  mianownik= 1.0_DP

DO
n=1+n;  A_n = matmul(A_n, A);    mianownik = mianownik * n
WYNIK = WYNIK + A_n / mianownik

IF ( abs( MAXVAL(A_n) ) < epsilon )  then
write(*, *)  ' przyblizona zbieznosc osiagnieta '
write(*, *)  '  ponizej  elementy WYNIK  wypisane  w/g  kolejnosci kolumn'
write(*, fmt='(4E20.8E2)')   WYNIK
EXIT
end if
!!!!
IF ( n > N_maxi)   then
write(*,*)  '  po  zsumowaniu  N= ', N,  '   wyrazow  ciagle nie mamy zbieznosci '
EXIT
end if

END DO
stop
end program szereg_macierzowy
```

Uwaga! Zastosowaliśmy znów nieskończoną pętlę DO, instrukcję exit oraz kilka macierzowych funkcji (matmul, oraz maxval).

4. Wyliczenie wyznacznika metodą sprowadzenia do postaci trójkątnej

Metoda ta (inaczej metoda Gaussa) oparta jest na podstawowych własnościach wyznacznika: jeżeli do jakiegoś wiersza dodamy (na sposób wektorowy) kombinację liniową innych wierszy (sumę innych wierszy, przemnożonych przez dowolne liczby) to wartość wyznacznika nie zmieni się (oczywiście oryginalna macierz zmienia się). Jeżeli uda się nam sprowadzić macierz do postaci trójkątnej (same zera pod główną diagonalą macierzy) to wyznacznik wyliczymy trywialnie jako iloczyn elementów z głównej przekątnej. Naszą implementację metody Gaussa przetestujemy na jednym konkretnym przykładzie: macierzy 10 x 10 o elementach $a_{ij} = \sin(C/i + C/j)$, gdzie $C = \pi/8$,

(Porównaj przykład z : A. Jeśmianowicz, J. Łoś, Zbiór zadań z algebry, PWN, Warszawa 1969).

```
program   wyznacznik
implicit none;
integer :: k, i, j
```

```
integer,dimension(1) :: max_j
integer, parameter :: n=10
integer, parameter :: DP=selected_real_kind(14)
real(kind=DP), dimension(n) :: jedna_kolumna
real(kind=DP), dimension(n,n) ::  a
real(kind=DP) :: det, pi8=3.14159265359_DP/8.0_DP
                                                    !!!   a_ij = sin( pi8/i + pi8/j)
do i=1, n; do j=1, n;   a(i, j) = sin(pi8/i + pi8/j)
end do;   end do;
!!!!!!!!!!!!
det =1.0_DP
!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
po_wierszach:  do k=1, n-1
          max_j = maxloc( abs(a(k, :)) )
          j=max_j(1)
                                          !!!  a_kj  jest największym (na moduł)
                                          !!! elementem z k-tego wiersza
                                          !!! przestawiamy j-tą i k-tą kolumnę
                                          !!! ze zmianą znaku wyznacznika
                                          !!! postać macierzy a zostaje zmieniona
                                          !!! (bez zmiany wartości wyznacznika)
          if (j > k) then
          jedna_kolumna=a(:, j)
          a(:, j)=a(:, k)
          a(:, k)=jedna_kolumna
          det =  -det
          end if
                                          !!! po przestawieniu kolumn największy
                                          !!! (na moduł) element jest w pozycji
                                          !!! diagonalnej;  jeżeli okaże się, że jest
                                          !!! zerem, to wyznacznik automatycznie też
                                          !!! będzie zerowy, wiec w takim przypadku
                                          !!! awaryjnie przerywamy obliczenia
    if ( a(k, k)  .eq.  0.0_DP) then
    det=0.0_DP
    EXIT po_wierszach
    end if

                                          !!! normujemy k-ty wiersz - tak aby
                                          !!! a(k, k)=1 (wystarczy normować kolumny
                                          !!! od k do n, bo poprzednie są już zerowe)
   det=det*a(k,k)
   a(k, k:) = a(k, k:)/a(k, k)

                                          !!! od  wiersza i-tego  (i zmienia sie od k+1
                                          !!! do n-tego)
                                          !!! odejmujemy  wiersz k-ty pomnożony
                                          !!! przez odpowiednia
                                          !!! liczbę tak dobrana, aby elementy
                                          !!! kolumny k-tej
```

```
                                        !!!  (leżące pod elementem   a(k,k) ) zostały
                                        !!! wyzerowane

          po_i: do i = k+1, n
          a(i, k:) = a(i, k:) - a(k, k:)*a(i, k)
          end do po_i;
end do po_wierszach
                                        !!! po zakończeniu pętli macierz “a” ma
                                        !!! postać trojkątną
                                        !!! pod główna diagonalą - są same zera;
                                        !!! wyznacznik jest równy iloczynowi
                                        !!! elementów diagonalnych (jest już
                                        !!! zresztą prawie  wyliczony)

     det = det*a(n, n)                  !!!  ostatni składnik iloczynu
!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
write(*, 4444) det
4444 format(" wyznacznik = ", E25.12)
stop
end program wyznacznik
```

5. Rozwiązywanie układu równań L x = b, gdzie L to dolna macierz trójkątna, x to kolumna niewiadomych, a b kolumna wyrazów wolnych:

$$\begin{bmatrix} L_{11} & 0 & 0 & 0 & \dots \\ L_{21} & L_{22} & 0 & 0 & \dots \\ L_{31} & L_{32} & L_{33} & 0 & \dots \\ \dots & \dots & \dots & \dots & \dots \\ L_{n1} & L_{n2} & L_{n3} & \dots & L_{nn} \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ x_3 \\ \dots \\ x_n \end{bmatrix} = \begin{bmatrix} b_1 \\ b_2 \\ b_3 \\ \dots \\ b_n \end{bmatrix}$$

Jak łatwo zauważyć, rozwiązanie istnieje, jeżeli żaden z diagonalnych elementów L nie jest zerowy:

$x_1 = b_1/L_{11}$, $x_2 = (b_2 – L_{21}\, x_1)/L_{22}$, …, $x_i = (b_i – L_{i1}\, x_1 – L_{i2}\, x_2 – L_{i3}\, x_3 - \dots L_{i,i-1}\, x_{n-1})/L_{ii}$

dla i = 2,3,…n..

W naszym przykładzie (poniżej) przyjmiemy dla ustalenia uwagi, że wszystkie elementy b_i = 1, podobnie wszystkie niezerowe elementy L też są równe jeden. Zakładamy, że L jest macierzą o wymiarach 20 x 20.

```
program  trojk
implicit none;
integer, parameter :: n =20
integer :: i, j ;
real, dimension (n) :: b, x
```

```
real, dimension(n, n) :: L
                                                !!! podstawiamy  za  b oraz  za L:
  b = 1.0;
  do i=1,n; do j=1,n;
        if (j > i) then; L(i, j) = 0.0;  else;  L(i, j)=1.0;  end if;
 end do; end do;
!!!
x(1) = b(1)/L(1, 1)
  do i= 2, n
 x(i) = (b(i)  -   sum(  L(i, 1: i-1)*x(1:i-1) ) / L(i, i)
 end do;
                                                !!! L(i, 1:i-1)*x(1:i-1)  jest macierzą
                                                !!! jednowymiarową  od 1 do  i-1
                                                !!! składającą się z elementów:  L(I, 1)*x(1),
                                                !!! L(I, 2)*x(2), ... sum   wykonuje sumę jej
                                                !!! elementów
open(77, file=' rozwiaz.txt')
write(77, *) '  rozwiazania  rown:  Lx=b  gdzie L to dolna macierz trojkatna'
write(77, *) ' ----------------------------------------------------------------------------'
write(77, *) '         j                x(j)    '
do j=1, n;      write(77, fmt='(i4, F16.6)' )   j , x(j);    end do;
close(77);
stop
end program trojk
```

I.B.Bloki: subroutines, functions (zewnętrzne, wewnętrzne); o macierzach ciąg dalszy

Blok funkcji jest bardzo podobny do głównego bloku programu: zaczyna się słowem kluczowym **function nazwa_funkcji (argumenty_formalne_funkcji),** a kończy się słowem kluczowym **end function nazwa_funkcji** (możliwe jest skrócenie: samo **end** lub **end function,** ale nie rekomendujemy takiego postępowania). W środku bloku mamy deklaracje (dotyczące argumentów, potem zmiennych lokalnych używanych przez blok), a potem instrukcje. Powinna pojawić się instrukcja **nazwa_funkcji = wyrażenie**, czyli blok powinien wyliczyć wartość funkcji i obliczoną wartość podstawić pod **nazwa_funkcji.**
Instrukcja **return** przerywa natychmiast wykonywanie funkcji, ze zwrotem kontroli do bloku, który właśnie ją wywoływał (do miejsca, gdzie rozpoczyna się następna instrukcja). Gdy blok funkcji jest już napisany możemy posługiwać się symbolem funkcji

nazwa_funkcji(argumenty_aktualne) w dowolnym wyrażeniu arytmetycznym, identycznie jak to robiliśmy z funkcjami standardowymi należącymi do biblioteki F90/95. Subroutine, w języku polskim procedura (wykonawcza), również wygląda jak blok funkcji; tyle tylko, że zaczyna się od słowa kluczowego:
subroutine nazwa_subroutine(argumenty_formalne). Wywołanie (wykonanie) następuje w jakimś bloku programu w wyniku wykonania instrukcji

call nazwa_subroutine(arumenty_ aktualne).

Bloki, czy to **function** czy **subroutine,** należą do jednej z trzech kategorii:

- **module function, module subroutine** czyli "modułowych"; są one najważniejszą (najbardziej nowoczesną) kategorią, ale zostaną omówione dopiero w części II;
- zewnętrznych (external) będących spuścizną ze starego Fortranu F77;
- wewnętrznych (określonych tylko lokalnie – tzn. wewnątrz bloku w którym zostały zdefiniowane).

I.B.1. Interface, argumenty funkcji, subrutyn, atrybut "intent"

Obecnie zajmiemy się jedynie drugą kategorią procedur, tymi zewnętrznymi. External function, external subroutine stanowią zupełnie odrębny blok, mający własne (lokalne) zmienne, przydzieloną zupełnie niezależnie pamięć (w osobnym miejscu RAM). Deklarowane zmienne lokalne mogą mieć identyczne identyfikatory jak identyfikatory zmiennych w bloku **program**, ale i tak są one jednostkami zupełnie niezależnymi. Komunikacja z innymi blokami – w szczególności z blokiem głównym (program…) następuje za pośrednictwem argumentów funkcji – subrutyny (inna możliwość – mogą istnieć zmienne globalne; tę możliwość omówimy nieco później).

W czasie wywołania funkcji (lub subrutyny) pojawia się poważny problem: czy typ i własności aktualnych argumentów wywołania zgodne są z typem argumentów formalnych? Dla prostego bloku **function/subroutine** sprawdzenie następuje przez odwołanie się do "kontekstu" użycia argumentów (w jęz. ang. określa się to jako "implicit interface"). Istnieje bardzo prosta reguła – jeżeli deklaracje i typy argumentów formalnych funkcji lub subrutyny są zasadniczo identyczne jak w starym Fortranie F77, to implicit interface jest możliwy. Jeżeli jednak wprowadzmay nowe elementy języka F90/95 – które były nieznane w F77, to wtedy tzw. "explicite interface" jest absolutnie konieczny. Innymi słowy dla "bardziej skomplikowanych" przypadków konieczne jest udzielenie blokowi wywołującemu odpowiednich informacji o argumentach w postaci tzw. bloku interface (explicit interface). Robimy to następująco: w bloku wywołującym tuż po zakończeniu deklaracji zmiennych umieszczamy pomiędzy słowami kluczowymi **interface, …., end interface** okrojoną kopię wywoływanej funkcji (lub subrutyny). Przez okrojenie rozumiemy usunięcie wszystkich instrukcji oraz wszystkich deklaracji zmiennych lokalnych.

A więc
interface

function nazwa_funkcji (argumenty_formalne_funkcji)
deklaracje argumentów formalnych
end function nazwa_funkcji
end interface

Uwaga!

- w interface po function nazwa_funkcji,… należy jeszcze wstawić ewentualne odwołania się do modułów oraz implicit none;
- jeden blok interface może zawierać kilka funkcji;
- w bloku interface NIE WOLNO umieszczać procedur i funkcji modułowych.

Powszechnie przyjęta przez programistów praktyka jest następująca: zawsze piszemy blok interface – niezależnie od tego czy jest on konieczny czy też nie. Na pewno jest on zawsze przydatny (zwłaszcza przy szukaniu błędów, co określane jest w slangu informatycznym jako “debugging”, czyli “odpluskwianie”).

Po przekazaniu tylu ogólnych informacji możemy pokusić się o stwierdzenie, że dla przeciętnego czytelnika jest to wszystko “mętne” i mało konkretne. Przejdźmy więc do przykładu.

Problem:

W przestrzeni trójwymiarowej (Euklidesowej) rozmieszczona jest pewna liczba równoległościanów. Każdy z nich może być zadany przez trzy wektory **a**, **b**, **c**, wychodzące z jednego z wierzchołków:

$\mathbf{a} = (a_x, a_y, a_z)$, $\mathbf{b} = (b_x, b_y, b_z)$, $\mathbf{c} = (c_x, c_y, c_z)$

gdzie a_x, … to składowe wektorów. Który z równoległościanów ma największą objętość ? Informacje pomocnicze :

Objętość V możemy wyliczyć jako wartość bezwzględną wyznacznika z macierzy

M -- V = abs(det(M)), gdzie M dane jest jako

$$M = \begin{bmatrix} a_x & a_y & a_z \\ b_x & b_y & b_z \\ c_x & c_y & c_z \end{bmatrix}$$

(porównaj W.L. Smirnow, Matematyka Wyższa, Tom III/1, PWN, Warszawa, 1965).

W pliku tekstowym krawedz.txt mamy dane dotyczące N takich brył (zakładamy że N<100). Na początku pliku w pierwszej linii mamy wartość N, a następnie N razy występują dane konkretnej bryły (linia komentarza, linia ze składowymi **a**, linia ze składowymi **b**, linia ze składowymi **c**) .

Przejdźmy teraz do naszego programu:

```
program objetosci_rownoleglo
implicit none
integer :: N, licznik;  integer, dimension(1) :: indeksy
real, dimension(3) :: a,b,c;
real, dimension(100) :: objetosci
character(len=72) :: koment
!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
interface
```

```
    real function  V(aa,bb,cc)
    implicit none;
    real, dimension(3), intent(in) :: aa,bb,bc
    end function V
end  interface
!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
open(51, file='krawedz.txt')
read(52, *)  N
do licznik=1, N
     read(51, fmt='(A72)') koment              !!! komentarz jest potrzeby tylko jako
                                               !!! przerywnik
                                               !!! w pliku, oddzielającym dane dla
                                               !!! poszczególnych  brył
    read(51, *) a;  read(51, *) b;  read(51, *) c
    objetosci(licznik) = V(a, b, c)
end do;         close(51)
!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
indeksy = maxloc(objetosci)
write(*, 4444)   indeksy(1),  objetosci (indeksy(1))
4444 format('' bryla nr. '', i2, ''   ma objetosc = '', F15.4, '' (najwieksza objetosc'')
stop
end program objetosci_rownoleglo
!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
real function  V(aa, bb, cc)
                                               !!! argumenty  formalne mogą się nazywać
                                               !!! a, b, c ;
                                               !!! nie spowoduje to konfliktu ze
                                               !!! zmiennymi a, b, c
                                               !!! z głównego bloku
implicit none;
real,dimension(3), intent(in) :: aa,bb,bc
V =  &                                         !!! kontynuacja w następnej linii
aa(1)*(bb(2)*cc(3)-bb(3)*cc(2)) - &
aa(2)*(bb(1)*cc(3)-bb(3)*cc(1)) + &
aa(3)*(bb(1)*cc(2)-bb(2)*cc(1))
RETURN
end function V
!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
```

Wzór na obliczenie wyznacznika 3 x 3 zastosowany w funkcji V traktujemy jako znany Czytelnikowi (por.: A. Mostowski, M. Stark, Elementy Algebry Wyższej, PWN, Warszawa, 1977).

Uwaga! Możliwa jest druga forma deklarowania funkcji, którą zademonstrujemy na tym samym przykładzie (pochyłym drukiem wskażemy różnice; kropki oznaczają: " to samo co w przykładzie u góry").

```
program objetosci_rownoleglo
```

```
……………………..
        interface
function  V(aa,bb,cc)
implicit none;
        real  V
real, dimension(3), intent(in):: aa, bb, bc
        end function V
end  interface
!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
……………….
end program objetosci_rownoleglo

!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
function  V(aa,bb,cc)                  !!! argumenty  formalne mogą się nazywać
                                       !!! a, b, c;
                                       !!! nie spowoduje to konfliktu ze
                                       !!! zmiennymi a, b, c
                                       !!! z głównego bloku
implicit none;

real V

……………
end function V

!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
```

Obie formy są używane równie często.
Przy deklarowaniu argumentów funkcji V napotkaliśmy kolejny NOWY atrybut, tzw. **INTENT** (intencja). Możemy mieć **INTENT(IN)**; oznacza to, że argumenty mogą być przez funkcję tylko przekazane "do środka". Faktycznie przekazywana jest wartość argumentu aktualnego (skopiowana do bloku), a same argumenty w bloku *wywołującym* NIE PODLEGAJĄ ŻADNEJ ZMIANIE !
Możemy mieć nieco inny atrubut: **INTENT(OUT),** co oznacza, że argument aktualny w bloku wywołującym otrzymuje w wyniku działania funkcji (lub subrutyny) określoną wartość, i to niezależnie od tego, jaka była jego wartość w czasie wywołania funkcji. Blok funkcji nie jest w stanie w ogóle wykryć jaka była wartość argumentu aktualnego tuż przed momentem wywołania. Ostatnia możliwość: atrybut **INTENT(INOUT)** jest połączeniem obu poprzednich wypadków; wartość argumentu może wtedy być przekazana do bloku funkcji, a zmieniona wartość na zewnątrz, tzn. do bloku wywołującego. Atrybut INTENT może być dla prostych przypadków (stare programy pisane w F77 przerobione przez "drobne kosmetyczne zmiany" na fortran F90/95) pomijany. Jest to zła praktyka - usilnie odradzamy!

Drugi przykład dotyczyć będzie przybliżonego całkowania metodą trapezów (metoda Simpsona, patrz np. A.Ralston, Wstęp do analizy numerycznej, PWN, Wrocław, 1971).

$$\int_a^b f(x)dx \approx \left[\sum_{j=1}^{N-1} f(x_j) + f(x_0)/2 + f(x_N)/2\right](b-a)/N$$

gdzie x_j = a + j*(b-a)/N, to położenia równoodległych punktów dzielących przedział całkowania [a,b] na N części. W nowym przykładzie wprowadzimy również nowy atrybut : **external**. Jeżeli występuje on, to tylko podwójnie:

1. w przypadku argumentów formalnych funkcji lub subrutyny opatrzonych atrybutem **external** będzie on informował kompilator, że dany argument formalny jest jakąś bliżej nieokreśloną (zewnętrzną) funkcją lub subrutyną, a nie zwykłą zmienną;
2. w przypadku wywoływania bloku z argumentem formalnym o atrybucie **external** przez blok główny (lub jakiś inny blok) należy w tymże bloku zadeklarować dodatkowo listę **aktualnych** argumentów **również z atrybutem external**.

Uwaga! Nie umieszczamy tej listy aktualnych argumentów w INTERFACE, gdyż funkcje te nie są wywoływane przez główny blok, a jedynie podstawiane w miejsce odpowiednich argumentów formalnych.

```
program calki;    implicit none;
integer :: N,  j;    real ::  a, b, pi=3.14159265,  cal
real, external :: f, g                          !!!  atrybut external sygnalizuje tutaj, że
                                                !!!  f  oraz  g  zostaną podstawione jako
                                                !!! argumenty aktualne  (za formalny
                                                !!! argument fun)  w  bloku funkcji
                                                !!! calka;

real,  intrinsic :: sin, sqrt                   !!! funkcje standardowe  sin, sqrt
                                                !!! używane  w  podobnym  kontekscie
                                                !!! muszą być zdeklarowane jako
                                                !!!  intrinsic (zamiast external)
character(len=4) ::  nazwa_funkcji
INTERFACE
      function calka(a, b, N, fun); implicit none;
      real :: calka
      real,  intent(in) :: a,b
      integer,  intent(in) :: N
      real, external :: fun
      end function calka
      !!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
      subroutine podstaw(nazwa_funkcji, a, b, N, aa, bb, NN, napis)
                                                !!! ta subrutyna  powoduje przypisanie
                                                !!! określonych wartości zmiennym
                                                !!! a, n, N,   nazwa_funkcji;
                                                !!! jej użycie nie jest absolutnie konieczne
                                                !!! ale wygodne, zwłaszcza w przypadku,
                                                !!! jeżeli kilkakrotnie podstawiamy nowe
                                                !!! wartości tym samym zmiennym
                                                !!! (a więc chodzi jedynie o skrócenie zapisu
                                                !!! kodu)
      implicit none;
      real, intent(out) :: a, b
      integer, intent(out) :: N
      integer, intent(in) :: NN
```

```
        real, intent(in) :: aa, bb
        character(len=4), intent(in) :: napis
        character(len=4), intent(out) :: nazwa_funkcji
        end subroutine podstaw
        !!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
        subroutine   pisz(nazwa_fun,a,b,N,cal); implicit none;
                                                !!!parę razy piszemy te same zmienne;
                                                !!! wprowadzenie subrutyny  pisz
                                                !!!pozwala nam  na skrócenie kodu
        real, intent(in) :: a, b, cal
        integer, intent(in) :: N
        character(len=4), intent(in) :: nazwa_fun
        end subroutine pisz
END INTERFACE
!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
write(*,*)  ' calki z roznych  funkcji;   przyklady:   '
write(*,*) ' ------------------------------------------'
!!!!!!!!!!!!!!!!!!!
rozne_funkcje: do   j=1, 4
select case(j)
     case(1)
     call podstaw (nazwa_funkcji, a, b, N,  -1.0, 1.0, 101, 'f')
     cal=calka(a, b, N, f)
     case(2)
     call podstaw (nazwa_funkcji, a, b, N,  0.0, 1.0, 100, 'g')
     cal=calka(a, b, N, g)
     case(3)
     call podstaw (nazwa_funkcji, a, b, N, 1.0E-16, 1.0, 100, 'sqrt')
                                                !!! całka od 0 do 1 z sqrt(x);
                                                !!! zamiast a=0
                                                !!! podstawiliśmy 1.0E-06
                                                !!! (prawie zero) gdyż  argument  sqrt
                                                !!! nie może być ujemny ani  zero
     cal=calka(a, b, N, sqrt)
     case(4)                                    !!!  całka z funkcji sinus
     call podstaw( nazwa_funkcji, a, b, N,  0.0, 2.0*pi, 101, 'sin')
     cal=calka(a, b, N, sin)
end select
!!
call pisz(nazwa_funkcji, a, b, N, cal)
end do rozne_funkcje
!!
stop
end program calki
!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
function calka(a, b, N, fun)
implicit none;
real :: calka
real, intent(in) :: a,b
```

```
integer, intent(in) :: N
real, external :: fun                    !!! external sygnalizuje, że fun jest
                                         !!! jakąkolwiek funkcją
                                         !!! zewnętrzną, a nie zwykłą zmienną;


                                         !!!  teraz zmienne lokalne potrzebne do
                                         !!! wykonania bloku całka:
real :: dx,  x_j, suma
integer :: j
!!!!!!!!!!!!!!!
   suma= 0.5 * ( fun(a) + fun(b) );
   dx = (b-a) / N
   x_j = a
   do j=1,N-1
   x_j = x_j + dx
   suma=suma+ fun(x_j)
   end do;
calka = suma*dx
end function calka
!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
function f(x)                            !!! przykładowa nieparzysta funkcja
implicit none;
integer, parameter :: DP=selected_real_kind(15)
real :: f
real, intent(in) :: x
f = sin(x) *  exp(-x*x) +   x* (exp(x) + exp(-x)) + &
   cos(x) * ( x + x**3/3.0 + x*11/11.0)
end function f
!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
function g(x)                            !!!   przykładowy prosty wielomian:
                                         !!!   1+x/2 + x**2/3 + x**3/4+ x**4/5
implicit none;
real :: g
real,intent(in) :: x
!!!
real,dimension(4) :: &      kontunuacja w nastepnej linii
a =  (/0.5, 0.33333333, 0.25, 0.2/)
real :: suma;   integer :: j
!!!
suma=1.0
do j=1, 4;  suma=suma+ a(j)*x**j;  end do;
g=suma
end function g
!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
subroutine podstaw(nazwa_funkcji, a, b, N, aa, bb, NN, napis)
implicit none;
real, intent(out) :: a,b
integer, intent(out) :: N
real, intent(in) :: aa, bb
```

```
integer, intent(in) :: NN
character(len=4), intent(in) :: napis
character(len=4), intent(out) :: nazwa_funkcji
N=NN;   a=aa;   b=bb;
nazwa_funkcji = napis
end subroutine podstaw
!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
subroutine   pisz(nazwa_funkcji, a, b, N, cal)
implicit none;
real, intent(in) :: a,b,cal
integer, intent(in) :: N
character(len=4), intent(in) :: nazwa_funkcji
write(*, fmt='(" dolna granica   a=", F15.8)') a
write(*, fmt='(" gorna granica   b=", F15.8)') b
write(*, fmt='(" ilosc przedzialow    N  =", I8)') N
write(*, fmt='(" calka z funkcji  ",A4," = ",E25.12)')  nazwa_funkcji, cal
write(*, *) ' ====================================='
end subroutine pisz
```

W programie podanym powyżej możliwe jest opuszczenie INTERFACE. Nie wolno jednak nigdy zapomnieć o atrybutach **external** – zarówno w bloku calka, jak i w bloku głównym.

Kolejny (dosyć sztuczny) przykład zademonstruje użycie **INTENT(INOUT)**. Zadanie jest następujące: wczytać z pliku (o nazwie 100x3.txt) 100 trójek liczb (x,y,z) . Są to trójki nieposortowane – należy je ponownie zapisać na tym samym pliku ale tak, aby elementy każdej trójki były uporządkowane: x<y<z. Przy okazji nauczymy się jak usuwać stare, niepotrzebne pliki.

```
program   sortuj_trojki
implicit none;   integer :: j, k
integer, parameter :: N=100,  DP=selected_real_kind(15)
real(kind=DP), dimension(N, 3) :: trojki;    real(kind=DP) :: x,y,z
   INTERFACE
   subroutine  sortuj(xx, yy, zz);    implicit none;
   integer, parameter:: DP=selected_real_kind(15)
   real(kind=DP),  INTENT(INOUT) :: xx, yy, zz
   end subroutine sortuj
   END INTERFACE
!!!!!!!!!!!!!!!!
open(55, file='100x3.txt')
do  j=1, 100 ;      read(55, *)  x, y, z;
call   sortuj(x, y, z)                          !!! po wykonaniu subrutyny  stare wartości
                                                !!! x, y, z
                                                !!! zostaną zatarte;
                                                !!! w ich miejsc otrzymamy nowe wartości
                                                !!! tak przestawione, aby   x<y<z)
 trojki(j, 1)=x;  trojki(j, 2)=y;  trojki(j, 3) =z;
end do;
!!!
close(55,  status='delete')                     !!! (status='delete') po zamknięciu plik
```

```
                                        !!! zostanie zmazany

open(55, file='100x3.txt')              !!! plik o nazwie   100x3.txt  nie istniał;
                                        !!! próba otwarcia
                                        !!! nieistniejacego pliku spowoduje to, że
                                        !!! zostanie on  założony przez system
                                        !!! operacyjny (na razie zupełnie pusty plik)

do j=1,100;
write(55, 4444)  trojki(j, 1),  trojki(j, 2), trojki(j, 3);
end do;
close(55)
                                        !!! brak słowa kluczowego status
                                        !!! traktowany jest domyślnie jako
                                        !!! close(55, status='keep')
                                        !!! czyli  ''zachowaj''   plik
4444 format(3F25.12)
stop
end program sortuj_trojki
!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
 subroutine  sortuj(xx, yy, zz);  implicit none;
                                        !!! argumenty formalne subrutyny
 integer, parameter:: DP=selected_real_kind(15)
 real(kind=DP), INTENT(INOUT) :: xx, yy, zz
                                        !!! zmienne lokalne  subrutyny
integer :: ktory_mini, ktory_maxi
real(kind=DP) ::  mini, maxi, srodek
                                        !!! (x,y,z) nie mają nic wspólnego  z  (x,y,z)
                                        !!! z głównego  bloku,  mimo identycznej
                                        !!! nazwy
ktory_mini=1;   ktory_maxi=3
mini=xx;   maxi=zz
                                        !!! przyjmujemy roboczą hipotezę,
                                        !!! że xx jest najmniejszy, a zz największy
                                        !!! teraz   sprawdzamy oraz aktualizujemy
                                        !!! (jeżeli hipoteza była fałszywa)
if (yy < mini)  then;   mini=yy;  ktory_mini=2;   end if
if (zz < mini)  then;    mini=zz;  ktory_mini=3;   end if
!!!!!!!!!!!!!!!!!!!
if (yy > maxi)  then;  maxi=yy;  ktory_maxi=2;   end if
if (xx > maxi)  then;  maxi=xx;  ktory_maxi=1;   end if
!!!!!!!!!!!!!!1
select case(ktory_mini+ktory_maxi)
case(3) ;   srodek = zz                 !!! gdyż   k_mini+k_maxi= 1+2
case(4);   srodek  = yy
case(5);   srodek  = xx
end select
xx= mini;  yy=srodek; zz= maxi
```

```
                                          !!! nowe, zaktualizowane wartości
                                          !!! argumentów
                                          !!! zostają  przekazane do bloku
                                          !!! wywołującego
end subroutine sortuj
```

Kolejny, ostatni już w tym rozdziale przykład dotyczy prostego problemu "automatycznej" obsługi wielu plików z danymi. Dla ustalenia uwagi przyjmujemy, że w 8 plikach podane są tabelki funkcji (jako dwie kolumny: w pierwszej będzie argument x, a w drugiej wartość funkcji y). Aby "automatyczna" obsługa była możliwa założymy, że nazwy plików wykazują dużą regularność; nazywają się: tabelka.1, tabelka.2, tabelka.3,…. (tak więc zmienia się tylko rozszerzenie nazwy pliku – i jest to cyfra zmieniająca sie od 1 do 8). Przyjmiemy, że w plikach z danymi nad tabelką w pierwszej linii mamy podaną liczbę punktów N.

Każda tabelka opisuje jakąś serię pomiarów tej samej wielkości (fizycznej, chemicznej lub jakiejś innej), która posiada minimum. Naszym zadaniem jest lokalizacja tego minimum (dla każdej serii pomiarowej osobno) – i to w miarę dokładnie. Program nasz powinien więc automatycznie otwierać plik dla danej serii pomiarowej, lokalizować w przybliżeniu minimum i przez interpolację paraboliczną w okolicy minimum (dopasowanie paraboli do trzech punktów doświadczalnych najbliższych mimimum) poprawić lokalizację minimum. Dopasowanie paraboli (a*x*x+b*x+c) do trójki punktów (x_{j-1},y_{j-1}), (x_j,y_j), (x_{j+1},y_{j+1}), czyli wyliczenie stałych a,b,c wymaga od nas rozwiązania trzech liniowych równań z trzema niewiadomymi a,b,c. Zrobimy to metodą Cramera (A. Mostowski, M. Stark, Algebra Wyższa, PWN, Warszawa, 1965) posługując się wyznacznikami o rozmiarach 3x3.

A oto program:

```
program   minimum
implicit none;
real, dimension(1000) :: x, y             !!! nie więcej niż 1000 punktów w serii
                                          !!! pomiarowej
integer :: j, k, N;   real ::  x_mini, y_mini    !!! zmienne określające położenie minimum
character(len=1), dimension(8) :: numer
  INTERFACE
  subroutine  find_mini(x, y, N, x_mini, y_mini)
  implicit none;
  real, dimension(1000),  intent(in) :: x, y
  real, intent(out) :: x_mini, y_mini
  integer, intent(in) :: N
  end subroutine find_mini
  END INTERFACE
!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
open(66, file='wyniki.txt')               !!! wyniki programu drukujemy do pliku
                                          !!! "wyniki.txt"
!!
do j=1, 8
numer(j) = achar(  iachar('1') + j-1  )
                                          !!! użycie funkcji standardowych  achar,
                                          !!! iachar  spowoduje, że
```

```
                                        !!! liczba 1  przejdzie w litere  '1'; liczba  2
                                        !!! przejdzie w literę   '2' itd.
end do;
!!!!!!!!!!!!!!!!!!!!!!!

po_plikach:   do   j=1, 8
   open(55, file='tabelka.'//numer(j) )          !!! otwieramy po kolei różne pliki z danymi
    write(66, *) ' ---------------------------------'
    write(66, fmt='(" plik z danymi =", A)')   'tabelka.'//numer(j)
                                        !!! w wynikach piszemy informację jaki
                                        !!! właśnie plik otwarliśmy
    !!!!!!!!!!!!!!!
   read(55,*)  N
        do k=1, N;  read(55,*)  x(k),  y(k);  end do;
   close(55)
!!!!!!!!!!!!!!!!!!!!!!!
   call  find_mini(x, y, N, x_mini, y_mini)
   write(66, fmt = '(" pozycja minimum,  x_mini,  y_mini =",  2F16.6)' )  &
    x_mini, y_mini
end do po_plikach
!!!
write(66,*) ' ----------------------------'
close(66)
stop
end program minimum
!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
subroutine  find_mini(x, y, N, x_mini, y_mini)
implicit none;
real, dimension(1000), intent(in) :: x,y
real,  intent(out) :: x_mini, y_mini
integer, intent(in) :: N
!!!!!!!!!!!!!!!   zmienne lokalne:
real :: x1, x2, x3, y1, y2, y3, a, b, c, wyznacznik
integer, dimension(1) :: pozycja_najmniejszego
integer :: poz
real, dimension(3,3) :: m,  m0                 !!! dwie pomocnicze macierze
!!
INTERFACE                                       !!! wprowadzamy interface gdyż
                                                !!! blok  find_mini będzie
                                                !!! wołał inny blok (funkcje det3)
real function det3(arr)   !
real, dimension(3, 3), intent(in) :: arr
end function det3
END INTERFACE
                                                !!! znajdujemy pozycję najmniejszego
                                                !!! elementu i  punkty sąsiednie;
                                                !!! ta trójka podaje wstępną lokalizację
                                                !!! minimum
pozycja_najmniejszego = minloc( y(1:N) )
```

```
poz=pozycja_najmniejszego(1)
                                        !!! uwaga:  minloc jest fukcją standardową
                                        !!! o wartości macierzowej (!!!) i to
                                        !!! nawet wtedy gdy ta macierz ma tylko
                                        !!! jeden element;
                                        !!! (minloc jest macierzą o tylu elementach,
                                        !!! ile indeksów ma argument minloc)
write(*, *)  poz
x1=x(poz-1);  y1=y(poz-1)               !!! zapamiętujemy trzy pary elementów
                                        !!! macierzowych jako
                                        !!! zwykle zmienne ; nie jest to
                                        !!! konieczne (tzn. pomocnicze zmienne),
                                        !!! ale oszczędzi to nam sporo pisania
x2=x(poz);      y2=y(poz)
x3=x(poz+1);  y3=y(poz+1)
!!!
                                        !!! teraz interpolacja kwadratowa
                                        !!! zakładamy ze przez trzy punkty (x1, y1),
                                        !!! (x2,y2),   (x3,y3)  przechodzi parabola
                                        !!! o równaniu:
                                        !!! a x*x + b*x + c;   obliczamy a, b, c
                                        !!! znajdujemy dla jakiego x_mini
                                        !!! parabola ma minimum;
                                        !!! odpowiedź:
                                        !!! x_mini = -0.5*b/a;
                                        !!! y_mini = a*x_mini**2 + b*x_mini +c
                                        !!! do obliczenia niewiadomych a, b, c
                                        !!! stosujemy wzory Cramera
                                        !!! (rozwiązanie układu 3-ch równań z 3-ma
                                        !!! niewiadomymi)
!!!

m0 =  reshape(  &
(/x1**2, x2**2, x3**2, x1, x2, x3, 1.0, 1.0, 1.0/),  (/3, 3/) )
                                        !!!  tzw. macierz Vandermonda
wyznacznik=det3(m0)
!!!
m = m0;  m(:, 1) = (/y1, y2, y3/)
                                        !!! m jest takie jak m0 z tym, że
                                        !!! pierwsza kolumna zostaje zastąpiona
                                        !!! kolumną wyrazów y1, y2, y3
a = det3(m)/wyznacznik
                                        !!! wyjaśnienie wzorów Cramera: patrz
                                        !!! Mostowski, Stark, Algebra Wyższa
!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
m = m0;  m(:, 2) = (/y1, y2, y3/)
                                        !!! m jest takie jak m0 z tym, że
                                        !!! druga kolumna zostaje zastąpiona
                                        !!! kolumną wyrazów y1, y2, y3
b = det3(m)/wyznacznik
```

```
!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
m = m0;  m(:, 3) = (/y1, y2, y3/)
                                        !!! m jest takie jak m0 z tym, że
                                        !!! trzecia kolumna zostaje zastąpiona
                                        !!! kolumną wyrazów y1,y2,y3
c = det3(m)/wyznacznik
!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
x_mini = -0.5*b/a
y_mini =  a*x_mini**2 + b*x_mini + c
end subroutine find_mini
!!!!!!!!!!!!!!!!!!!!!!!!!
!!!!!!!!!!!!!!!!!!!!!!!!!
!!!!!!!!!!!!!!!!!!!!!!!!!
real function  det3(arr)
real, dimension(3, 3), intent(in) :: arr
det3 = arr(1, 1)* ( arr(2, 2)*arr(3, 3)-arr(2, 3)*arr(3, 2) ) - &
       arr(1, 2)* ( arr(2, 1)*arr(3, 3)-arr(2, 3)*arr(3, 1) ) + &
       arr(1, 3)* ( arr(2, 1)*arr(3, 2)-arr(2, 2)*arr(3, 1) )
                                        !!!  to był wzór na wyznacznik 3x3   (patrz
                                        !!! Mostowski, Stark, Algebra Wyższa)
end function det3
!!!!!!!!!!!!!!!!!!!!!!!!!
!!!!!!!!!!!!!!!!!!!!!!!!!
!!!!!!!!!!!!!!!!!!!!!!!!!
```

I.B.2. Funkcje i subrutyny wewnętrzne. CONTAINS

Każdy blok programu może mieć swoje prywatne procedury (funkcje lub subrutyny), czyli tzw. *procedury wewnętrzne*. Nie są one określone *nigdzie poza obszarem bloku w którym są zdefiniowane*. Nie wymagają INTERFACE - faktycznie *wprowadzanie interface jest dla nich zabronione* (inaczej: mówimy że interface dla nich jest automatyczny).

Definiujemy je identycznie jak inne procedury ale:

1. dopiero po zakończeniu wszystkich instrukcji w bloku czyli na samym końcu;
2. po słowie kluczowym **CONTAINS;**
3. **procedura wewnętrzna ma dostęp do wszystkich zmiennych zadeklarowanych w bloku nadrzędnym** (są dla niej zmiennymi globalnymi) – a więc używa się identycznych, co w bloku nadrzędnym identyfikatorów zmiennych;
4. w przypadku konfliktu nazw zmiennych, czyli wtedy, gdy nazwa (ponownie zadeklarowanej) zmiennej w procedurze wewnętrzej i w bloku nadrzędnym jest taka sama, wtedy w procedurze wewnętrznej używana jest zmienna lokalna; ale zmienna bloku nadrzędnego nie ulega zmianie. Można to sobie wyobrazić w następujący sposób. Blok wewnętrzny dodaje do dażdej deklaracji swoich zmiennych lokalnych "niewidzialny" indeks (po prostu nazwę procedury wewnętrznej); tak więc optycznie dwie zmienne: zewnętrzna i wewnętrzna **mogą wyglądać** tak samo, ale takie same nie są.

Przykładowo, napiszmy funkcję
$f(x,y) = g(1/y+x)/1 + g(1/y+x)^2/2! + g(1/y+x)^3/3! + \ldots$

gdzie $g(x) = \cos(x)\, e^{-x} + 1/(\pi + x^2) + 1/(\pi + x^4) + \text{asin}(1.0/3.0)$.
Możemy to zrealizować w sposób następujący:

```
real function   f(x, y)
implicit none;
real, intent(in) ::  x, y;
real :: f
real, parameter :: pi=3.14159
integer, parameter :: N_max = 50
integer :: j;
real :: suma, silnia,  z,   arcussinus
character(len=20) :: napis = ’  liczymy   f ’
!!!!!!!!!!!!!!!!!!
suma= 0.0;    silnia = 1.0;   z =   x + 1.0/y
arcussinus = asin(1.0/3.0)
!!!
do j=1, N_max;
   silnia= silnia*j;    rekurencyjny wzor na silnie
   suma=suma+  g(z)**j/silnia
end do
f = suma
!!!
                                          !!! koniec instrukcji;    jako ostatnie
                                          !!! w bloku
                                          !!! występuje  teraz CONTAINS

    CONTAINS
     real function  g(x)                  !!! x -- ta sama nazwa zmiennej  co dla
                                          !!! bloku funkcji f
                                          !!! ale nie ma  błędu
     real, intent(in) :: x
     character(len=20) :: napis = ’  teraz funkcja g ’
     g = cos(x)*exp(-x) + 1.0/(pi + x**2) + 1.0/(pi+ x**4) + arcussinus
                                          !!! uwaga:  pi nie deklarowaliśmy,  bo pi
                                          !!! jest  zdefiniowane w bloku nadrzędnym
                                          !!! a więc jest określone   w bloku   g;
                                          !!! tak samo  blok  g  także wie
                                          !!! jaka jest wartość
                                          !!! zmiennej  “arcussinus” obliczonej
                                          !!! w bloku nadrzędnym
     write(*, *)   napis
                                          !!! zostanie wydrukowany napis ‘teraz
                                          !!! funkcja g’
                                          !!! mimo pozornego konfliktu ze zmienną
                                          !!! globalna    ‘napis’
                                          !!! z bloku nadrzędnego;
                                          !!! przypominamy:  zmienna lokalną
                                          !!! ‘napis’   wyobrażamy sobie
                                          !!!  jako  zmienną   napis_z_bloku_g,  gdzie
                                          !!! przyrostek “ _z_bloku_g”
```

!!! jest przed nami ukryty

end function g
end function f

Ponieważ bloku interface nie potrzeba (a przez to "nasze życie staje się prostsze") więc używanie CONTAINS oraz procedur wewnętrznych jest powszechnie stosowaną praktyką.

I.B.3. Funkcje mogą mieć wartości macierzowe

Funkcje mogą mieć wartości macierzowe (uprzedzając dalszy wykład – jest to możliwe tylko dla jednej kategorii macierzy: o określonych wymiarach oraz liczbie elementów; właśnie tych, które jak dotąd poznaliśmy). W F90/95 nazywa się je "explicite shape arrays", czyli są to macierze o określonym kształcie. Dla innych kategorii macierzy, które omówimy nieco później, czyli dla tzw. "assumed shape arrays", oraz dla macierzy dynamicznych (tworzonych dynamicznie) tzw. "allocatable arrays" nie jest to możliwe.

Jeżeli wartością funkcji jest macierz, to blok INTERFACE jest absolutnie konieczny (nie można go opuścić).

Możemy teraz przejść do konkretnego prostego przykładu. Chcemy rozwiązać równanie różniczkowe drugiego rzędu opisujące ewolucję oscylatora anharmonicznego (tzw. oscylator Duffinga):

$$d^2x/d^2t = a\,sin(\omega t) + bx^3 + c\,dx/dt + f\,x$$

gdzie x(t), to położenie zależne od czasu t, prędkość v(t) = dx/dt (pierwsza pochodna) , a, b, c, f oraz omega (częstość kątowa), to dane stałe. Po prawej stronie równania mamy siły, kolejno: periodyczną zewnętrzną siłę wymuszającą (sinus), siłę anharmoniczną (x do trzeciej), siłę tarcia oraz zwykłą siłę harmoniczną (człon liniowy w x).

Jedno równanie oscylatora anharmonicznego (drugiego rzędu) można zamienić na 2 równia pierwszego rzędu posługując się podstawieniem v(t) = dx/dt:

$$dx/dt = v$$
$$dv/dt = a\,sin(\omega t) + bx^3 + c\,v + f\,x$$

Rozwiązanie równania przy podanych konkretnych warunkach początkowych: w chwili t_0, położenie x oraz prędkość v mają wartości x_0 , v_0 chcielibyśmy mieć w postaci trójkolumnowej tabelki (pierwsza kolumna wartości czasu t w interwałach co dt, druga wartość x(t), trzecia v(t)). Taka forma jest przydatna, jeśli chcemy oglądnąć wykres rozwiązania. Profesjonalny pakiet graficzny (np. *grapher*, *gnuplot*, *xmgr*, itp) zamieni naszą tabelką na gotowy wykres lub wydruk – i to praktycznie całkiem automatycznie, czyli bez dalszej obróbki danych.

W celu rozwiązania naszego zadania posłużymy się metodą Rungego-Kutty drugiego rzędu.

Przykładowo: dla jednego tylko równania różniczkowego i jednej zmiennej, czyli dla dx/dt = f(x,t) pojedyńczy krok metody drugiego rzędu Rungego-Kutty startującej z danego punktu (t_0, x_0), a usiłującej podać wartość $x(t_0+dt)$ dla chwili t_0+dt (dt jest dane; powinno być bardzo małe) może być podany przez następującą receptę

1. oblicz k1= dt $f(x_0, t_0)$;
2. oblicz k2= dt $f(x_0 + 0.5\,k1, t_0 + 0.5\,dt)$;
3. oblicz $x(t_0+dt) = x(t_0) + k2$.

Uogólnienie na przypadek układu kilku równań pierwszego rzędu jest trywialne - jedyna zmiana to ta, że k1 oraz k2 będą teraz wektorami.

Przejdźmy do naszego programu:

```
program  rownanie_rozniczkowe
implicit none;
integer  :: ilosc_krokow, j                    !!!   ilosc kroków
real, parameter :: pi2 = 2.0 * 3.14159265
real, dimension(2) :: xv_stare, xv_nowe,  k1, k2
real :: t_stare, t_nowe,  dt
real ::  a, b, c, d, omega, okres
real :: x0, v0, t0
!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
  interface
  function  lewe_strony(xv, t, a, b, c, d, omega)
                                               !!! lewe strony 2-ch równań różniczkowych
  implicit none;
  real, dimension(2) :: lewe_strony
  real, dimension(2), intent(in) :: xv
  real,  intent(in) :: t, a, b, c, d, omega
  end function lewe_strony
  end interface
!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
open(55, file='duff.dat')                      !!!  wczytujemy wartości a, b, c,  ……
read(55, *)  a, b, c, d
read(55, *) omega
read(55, *)  x0,v0,t0
read(55, *)  ilosc_krokow
close(55)
okres = pi2/omega;                             !!! znany wzór z fizyki (szkoła średnia…)
dt = okres/ilosc_krokow                        !!! przyjmujemy, że śledzimy rozwiązanie
                                               !!! po jednym pełnym okresie
                                               !!! periodycznej siły wymuszającej
                                               !!! (tej z sinusem)
!!!!!!!!!!!!
!!!!!!!!!!!!
open(66, file='duff.tab')                      !!! otwieramy trójkolumnową tabelkę z
                                               !!! wynikami,
xv_stare=(/ x0, v0/);   t_stare=t0             !!! podstawiamy warunki początkowe
                                               !!! równania różniczkowego
!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
do j=1, ilosc_krokow                           !!! w każdym wykonaniu pętli wyliczamy
                                               !!! nowe położenie, prędkośc i czas,
                                               !!! oraz pomocniczo
                                               !!! nową trójkę liczb: k1, k2  xv_nowe
   k1= dt*lewe_strony(xv_stare, t_stare, a, b, c, d, omega)
   k2= dt*lewe_strony(xv_stare +0.5*k1,  t_stare +0.5*dt, a, b, c, d, omega)
   xv_nowe = xv_stare + k2;   t_nowe=t_stare+dt
   write(66, 6666)  t_nowe, xv_nowe
   6666 format(3F15.8)
```

```
                                        !!! t_nowe xv_nowe jest drukowane
                                        !!! nasz nowy punkt przyjmujemy teraz
                                        !!! jako punkt startowy w następnym kroku
   t_stare= t_nowe;
   xv_stare=xv_nowe
end do
close(66)
stop
end program rownanie_rozniczkowe
!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
   function lewe_strony(xv, t, a, b, c, d, omega)
   implicit none;
   real, dimension(2) :: lewe_strony
   real, dimension(2), intent(in) :: xv
   real,  intent(in) :: t, a, b, c, d, omega
                                        !!! teraz zmienne lokalne:
    real :: x, v
    x=xv(1);   v=xv(2)
    lewe_strony = &
     (/ v,  d*x + c* v + b*x**3 + a* sin(omega*t)/)
    end function lewe_strony
```

Zwróćmy uwagę, że macierzowa funkcja “lewe_strony” ma ogromnie dużo parametrów formalnych. Dzieje się tak dlatego, że wczytywane parametry a, b, c,…. są określone tylko w bloku głównym, lecz nie w bloku funkcji. Musimy więc ich wartości przekazać jako argumenty. W tym miejscu widać, że przydałoby się bardzo mieć zmienne “wspólne” (globalne), które byłyby określone nie tylko w bloku głównym, ale i w bloku funkcji (bez przekazywania przez argumenty). Okaże się to możliwe, gdy tylko wprowadzimy tzw. moduły i procedury modułowe. Ale o tym później.

I.B.4. Cztery kategorie macierzy w fortranie F90/95: explicite shape array, automatic array, assumed shape array, allocatable array. Atrybut SAVE

Naprawdę istnieje 5 kategorii macierzy, ale piąta (tzn. “ assumed size array”) jest spuścizną po F77; rekomenduje się, aby zapomnieć o tej możliwości – zamiast niej F90/95 wprowadza bardziej ogólne “ assumed shape array”.

I.B.4.1. Explicte shape array, automatic array (macierze o określonym kształcie; oraz macierze robocze w procedurach - automatyczne)

W bloku głównym spotkaliśmy już pierwszy typ “ explicite shape array” - macierze o ustalonym kształcie (przypominany: ściśle określona liczba wymiarów czyli indeksów; dla każdego indeksu określone granice zmienności; “od - do”). Macierze “explicite shape array” oczywiście mogą występować w blokach (funkcji, subrutyn). Granice zmienności indeksów są określone (jak pamiętamy, z wielu poprzednich przykładów) przez liczby całkowite, parametry całkowite (też liczby), lub przez najprostsze wyrażenia arytmetyczne o wartości całkowitej (dozwolona suma liczb, iloczyn, itp.).

Dla przypomnienia, deklaracje wyglądają następująco:

```
integer, parameter :: N= 100
real, dimension( 2:7,  1:N) ::  alfa
integer, dimension(N,  2*N+1) :: beta
character(len=3), dimension(-N:N,  N,  2:N+1) :: gamma
```

Jeżeli jednak macierz jest argumentem procedury, to mamy pewne bardzo ważne udogodnienie. Mianowicie "explicite shape array" nie oznacza w tym kontekście, że granice zmienności indeksów muszą być natychmiast ustalone. Wystarczy, że mogą być wyliczone z dostępnej informacji. Tak więc *możemy przekazać granice zmienności indeksów przez inne argumenty formalne procedury (najczęściej stosowana praktyka),* lub przez zmienne globalne (czyli modułowe, ale o tym później). Przykłady:

```
…….
real function f(a, b, M, c, N)
integer, intent(in) :: N, M
                                        !!! te zmienne określą granice zmienności
                                        !!! indeksów
real, dimension(10, 2:10), intent(inout) :: a   !!! explicte shape array   formal argument
real, dimension(M), intent(in) :: b             !!! explicite shape array

real, dimension(10, 1:M,  N:N+M),  intent(in) :: c
                                        !!! explicite shape array

!!!!!!!!!! teraz zmienne  lokalne funkcji
real, dimension(M) :: bb                !!! automatic array
real, dimension(10, 1:M, N:N+M)  :: cc  !!! automatic array
…….
```

W powyższym przykładzie spotkaliśmy od razu drugi typ macierzy, występujących wyłącznie w procedurach (funkcjach, subrutynach). Są to macierze automatyczne – inaczej macierze robocze (work array). Jest to odmiana "explicte shape array", występująca wyłącznie w procedurach **nie będące argumentam** i - granice zmienności indeksów (przynajmniej jednego) nie są określone od razu, ale mogą być wyliczone przez argumenty procedury. Macierz automatyczna wygląda jak argument formalny procedury - explicite shape array o granicach podanych przez parametry procedury, ale argumentem procedury NIE JEST. Macierze automatyczne są tworzone "automatycznie" w czasie konkretnego wywoływania procedury. Są więc namiastką macierzy dynamicznych.

Po wyjściu z procedury macierze automatyczne ***znikają*** *– nie ma żadnej możliwości ("wprost") zapamiętania wartości ich elementów*.

Jako konkretny przykład napiszemy funkcję macierzową "reskaling(A, n, c)", której argumentem jest macierz kwadratowa n x n, drugim argumentem jest n, trzecim liczba c, natomiast wartością jest macierz A – c E (gdzie E to macierz jednostkowa n x n):

```
function  reskaling(A, n, c)
implicit none;
integer, intent(in):: n
real, dimension(n, n), intent(in) :: A  !!! explicite shape array, n  przekazywane
                                        !!! jako  drugi   argument
```

```
real, dimension(n, n) :: reskaling        !!! "reskaling"  to wartość funkcji;
                                          !!! (przypominamy że intent jest
                                          !!! tutaj  niedozwolony)
real, intent(in) :: c
!!!!!!!!!!!!!! argumenty lokalne
integer :: j, k
do j=1, n;   do k=1, n;
   if (j  .eq.  k) then
   reskaling(j, k) =  a(j, k) – c
   else
    reskaling(j, k) = a(j, k)
   end if
end do;  end do;
end function reskaling
```

Przypominamy: w bloku wywołującym musimy umieścić odpowiedni **interface.**

Małe podsumowanie i zastrzeżenia:

1. Macierze "explicte shape array" mają granice określone przez liczby całkowite, stałe całkowite lub proste wyrażenia wyliczające się do stałych całkowitych.
2. Istnieją tylko 3 sytuacje w których "explicite shape array" może mieć granice nie będące stałymi:
   a) jeżeli macierz jest argumentem formalym (dummy argument) procedury (funkcji lub subrutyny) – wtedy jej granice mogą być przekazane jako inne argumenty formalne procedury,
   b) jeżeli macierz jest macierzą automatyczną w procedurze (czyli *de facto* nie jest dokładnie "explicte shape array", ale jest blisko z tą ostatnią spokrewniona) – jej granice wtedy mogą być określone przez argumenty formalne procedury (tak samo jak w punkcie 2a)
   c) jeżeli macierz jest wynikiem funkcji (funkcje macierzowe) – wtedy jej granice też mogą być przekazane przez inne argumenty formalne (czyli zasadniczo znów jest to to samo co w pkt. 2a).

I.B.4.2. SAVE

Ta sama reguła odnosi się do dowolnych **lokalnych** zmiennych w procedurach: *po wykonaniu procedury są one zapominane (przestają istnieć).* Można jednak temu zapobiec dodając w deklaracji zmienych lokalnych procedury atrybut **SAVE,** instruujący kompilator, że wartości tymczasowe zmiennych lokalnych w okresach pomiędzy wywołaniami procedury mają być zapamiętane (dygresja: robione jest to na ogół przez chwilowe umieszczanie zmiennej na stosie). Atrybutu **SAVE** dla macierzy automatycznych nie należy dodawać; jeśli zrobimy to pośrednio (przez ogólną deklarację save), to takie **SAVE** zostanie zignorowane.

Przykład:

```
……
real function f(a, n)
integer, intent(in) :: n;
```

```
real, dimension(n), intent(in) :: a
                                        !!! lokalne zmienne
integer, dimension(n) :: work_a
real, dimension(10, 10),  save  ::  work_b    !!! macierz lokalna work_b będzie cały
                                        !!! czas  pamiętana;
                                        !!! ale macierz work_a  (automatyczna)
                                        !!! nie będzie (po zakończeniu bloku)
……
subroutine   wykonaj(a, n)
integer, intent(in) :: n;
real, dimension(n), intent(inout) :: a
                                        !!!  lokalne zmienne
integer, dimension(n) :: work_a
real, dimension(10, 10) :: work_b
SAVE                                    !!! ogólna dyrektywa zapamiętania
                                        !!! wszystkich zmiennych lokalnych,
                                        !!! dla których jest to tylko mozliwe
                                        !!! (więc  SAVE NIE DOTYCZY  macierzy
                                        !!! automatycznej work_b)
                                        !!! (oraz SAVE nie dotyczy
                                        !!! argumentów procedury)
```

I.B.4.3. Assumed shape array jako argumenty procedur

Assumed shape array to macierze (będące argumentami formalnymi procedur) o kształcie poznawanym dopiero z kontekstu użycia, czyli o takim samym kształcie, jaki ma argument aktualny - przy wywołaniu procedury.

Tak więc assumed shape array może być jedynie argumentem formalnym procedury - przy absolutnej konieczności podania INTERFACE (lub umieszczenia procedury w module, wtedy INTERFACE jest automatyczny; o tym będzie później).

Assumed shape array nie może wystąpić w bloku głównym i nie może być wartością funkcji, nie może być składnikiem bardziej złożonych typów zmiennych (o tym też będzie później…).

Deklarując "assumed shape array" jesteśmy zobowiązani tylko do zadeklarowania (przez podanie dwukropku) liczby indeksów (wymiaru) macierzy. Granic zmienności nie określamy (przynajmniej dla jednego indeksu). Procedura jest jednak w stanie określić te granice przy konkretnym wywołaniu poprzez skojarzenie argumentu aktualnego (argumentu wywołania) z deklaracją parametru formalnego.

Pora na przykład. Przyjmijmy, że chcemy mieć uniwersalną procedurę (funkcję) przeskalowywującą macierze kwadratowe – taką jak z rozdziału I.B.3.1, ale z argumentem "assumed shape array":

```
function  reskaling(A, c)              !!! zakres zmienności  indeksów w macierzy
```

```
                                        !!! A nie jest określony
implicit none;
real, dimension(:, :), intent(in) :: A  !!! assumed shape array
                                        !!! dwa drukropki wskazują, że jest to
                                        !!! macierz dwuwymiarowa  (dwa indeksy)
real, dimension( size(A, 1), size(A, 2) ) :: reskaling
                                        !!! explicite shape array
                                        !!! reskaling : wartość funkcji ; (intent jest
                                        !!! tu niedozwolony)
                                        !!! liczba elementów w pierwszym
                                        !!! indeksie
                                        !!! i drugim indeksie  jest
                                        !!! wyliczana za pomocą funkcji
                                        !!! standardowej size
                                        !!! domyślnie przyjmujemy, że dolna
                                        !!! granica każdego indeksu jest
                                        !!! równa  1

real, intent(in) :: c
                                        !!! teraz pomocnicze zmienne (lokalne):
integer :: j, k, n, m
n= size(A, 1);   m =size(A, 2)
!
   if  (n  .ne.  m)  then
   write(*, *) ' blad: macierz niekwadratowa ;  stop !!!'
   stop
   end if
!
do j=1, n;  do k=1, n;
      if (j  .eq.  k)  then
      reskaling(j, k) =  a(j, k) – c
      else
      resklaling(j, k) = a(j, k)
      end if
end do; end do;
end function resklaling
```

Przypominamy o INTERFACE w bloku wywołującym.

Inne (fragmentaryczne) przykłady argumentów będących “assumed shape array”:

```
subroutine  dziwny_przyklad(a, n,  b, c, d, f, g)
implicit none;
integer, intent(in) :: n
character(len=10), dimension(n), intent(inout) :: a
                                        !!! explicit shape array
character(len=10), dimension(:, :, :), intent(in) :: b
                                        !!! assumed shape array
integer, dimension(n, :), intent(in) :: c
                                        !!! również assumed
```

```
                                        !!! shape array
                                        !!! (ze wzgledu na drugi indeks)
logical, dimension(2*n+1,  5:), intent(inout)  ::  d
                                        !!! d  to także assumed  shape array
                                        !!!  gdyż zamiast drugiego indeksu mamy
                                        !!! w deklaracji  "5:"  oznacza  to
                                        !!!  że dolna granica indeksu nr 2  jest
                                        !!! ustalona (wynosi 5),
                                        !!! ale górna granica JUŻ NIE JEST

integer, dimension( size(a,1) , n+1,  :,  :n), intent(out) :: f
                                        !!! f  to również assumed shape array:
                                        !!! granice 3-go indeksu nie są ustalone, a
                                        !!! dolna granica 4-go indeksu także nie jest
                                        !!! ustalona
!!!!!!!!!!!!!   teraz argumenty lokalne:
real, dimension( size(b,1), size(b,2), size(b,3) ) ::   g
                                       !!! automatic array

integer :: dolna_granica,  gorna_granica

dolna_granica= lboud(f, 4)              !!! wylicznie dolnej granicy 4-go indeksu
                                        !!! macierzy  f
gorna_granica=ubound(d, 2)              !!! wyliczanie górnej granicy 2-go indeksu
                                        !!! macierzy d
```

I.B.4.4. Macierze dynamiczne, "alokowalne" (allocatable); instrukcje: alllocate, deallocate, allocated

Macierze dynamiczne (tworzone i niszczone dowolnie w trakcie działania programu) deklarujemy dodając do deklaracji atrybut **allocatable**. Podobnie jak w przypadku "assumed shape array" w atrybucie **dimension** podajemy jedynie dwukropek w miejsce każdego indeksu żeby zaznaczyć, jaka będzie ranga (liczba indeksów) macierzy dynamicznej. Tworzymy je za pomocą instrukcji **allocate**, niszczymy (zwalnianie zajętej pamięci) przez instrukcję **deallocate.** Już po utworzeniu macierzy posługujemy się nią jak każdą inną macierzą.

Jeśli mamy wątpliwości, czy konkretna macierz istnieje (np. czy aby nie została zniszczona przez jakąś procedurę w trakcie egzekucji programu), to możemy posłużyć się instrukcją **allocated.**

Zastrzeżenia:

1. *allocatable array NIE MOŻE być argumentem FORMALNYM procedury*
2. *MOŻE być (i często bywa) argumentem aktualnym (argumentem wywołania); zachowuje się wtedy jak zwykła "explicte shape array"*
3. *Wartość funkcji NIE MOŻE być macierzą dynamiczną (allocatable array)*
4. *Nie wolno używać macierzy dynamicznych (podobnie assumed array) jako elementów zmiennych złożonych (tzw.* ***derived type,*** *o tym będzie później).*

Przechodzimy do przykładu. Przyjmiemy, że

```
program  macierze_dynamiczne
implicit none;
integer :: n, m
integer,  parameter :: DP=selected_real_kind(15)
integer ::   error
real(kind=DP), allocatable, dimension(:) ::   a, aa
real, allocatable, dimension(:, :)  :: b
real, dimension(10) :: wynik
…….
INTERFACE
subroutine  wykonaj1
……
subrutine wykonaj2
……
subroutine wykonaj3
……..
…….
END INTERFACE
…….
…….
n = 10;  m= 100;
allocate( a(10), stat=error )
                                    !!! powstaje macierz  a(1:10)
                                    !!! "stat" jest słowem kluczowym
                                    !!! (opcjonalnym, tzn. może ale nie musi
                                    !!! wystąpić) -
                                    !!! za wartość error podstawiane jest 0,  jeśli
                                    !!! operacja tworzenia macierzy się udała
                                    !!!  lub coś innego (zależy to od
                                    !!! konkretnego kompilatora),  jeżeli się nie
                                    !!! udała
                                    !!! (np.  gdy przekroczyliśmy  zakres
                                    !!! dostępnej pamięci  komputera)

if (error  .eq.  0)   then
call  wykonaj1(a)
end if                              !!! po utworzeniu macierz dynamiczna
                                    !!! może być
                                    !!! aktualnym argumentem (arg.
                                    !!! wywołania); przypominamy że
                                    !!! NIGDY  jednak nie może być
                                    !!! formalnym argumentem
deallocate(a1, stat=error)
                                    !!! gdy nie potrzebujemy już a1 zwalniamy
                                    !!! pamięć dynamiczną:
……..
if (allocated(aa) )    call wykonaj1(aa)    !!! (skrócona forma  if)
                                    !!! macierz  aa   nie była utworzona więc
                                    !!! allocated=.false.   a więc
```

```
                                        !!! ''call wykonaj1(aa)''  nie będzie
                                        !!! wykonane
……….
allocate( a( 5:12), aa(n:n+m) )         !!! tworzenie od razu 2-ch macierzy,
                                        !!! tutaj  nie skorzystaliśmy z możliwości
                                        !!! sprawdzenia powodzenia
                                        !!! instrukcji "allocate" (brak słowa
                                        !!! kluczowego stat)
……..
deallocate(a,aa)                        !!! niszczymy  a, aa
……..
allocate( a(10), aa(11:20), b(10,10), stat=error)
                                        !!! jednocześnie tworzymy 3 macierze
if (error  .eq.  0) then
a = 1.0;  aa = 8.0;
b(:, 5) = a;   b(:, 3)=aa
end if                                  !!! za 5-tą  kolumnę  b  powstawiamy
                                        !!! 1, 1, 1, 1,…..
                                        !!! za 3-cią  kolumnę  b  podstawiamy
                                        !!! 8, 8, 8,…..
 deallocate(a, b)
……
deallocate(aa)
……..
end program macierze_dynamiczne
```

Przykład ten zasadniczo wyjaśnił już prawie wszystko.

I.B.5. Funkcje i subrutyny mogą być rekurencyjne, tzn. mogą wywoływać same siebie

I.B.5.1.Subrutyny rekurencyjne

Rekurencji w F77 nie było. Obecnie w F90/95 możemy ją stosować (ale zalecamy ostrożność; o błąd łatwo, a konieczność stosowania rekurencji dla początkującego programisty nie jest zbyt duża). Przy stosowaniu rekurencji *Interface* w bloku wywołującym jest obowiązkowy (albo zamiast tego można takie procedury umieścić w module – interface jest wtedy "automatyczny"; ale o tym później). Zasady są dosyć proste - przed słowami **subroutine nazwa** wstawiamy słowo **recursive (recursive subroutine nazwa_subrutyny).** Druga zasada: musimy mieć możliwość przechowywania pośrednich wyników w czasie działania rekursji (zgodnie z logiką używanego algorytmu rekurencyjnego).

Niezniszczalnym "bokserskim workiem trenigowym" do nauki rekurencji jest funkcja n! = 1•2•3•…•n. Wzór rekurencyjny jest oczywisty: n! = n • (n-1)!
Oczywiście nie musimy stosować rekurencji, gdyż bez kłopotu możemy silnię wyliczyć w taki oto sposób:

```
integer function bez_rekurencji(n)
                                        !!! prymitywna wersja bez sprawdzania czy
                                        !!! argument  jest właściwy;  bez
```

```
                                        !!! sprawdzania czy wartość funkcji
integer,intent(in) :: n;
                                        !!! nie jest aby zbyt duża (nie mieści się w  4
                                        !!! bytach)
integer :: k; real :: silnia
silnia=1.0
do k=1,n;  silnia=silnia*k; end do;  bez_rekurencji=silnia
 end function bez_rekurencji
```

Alternatywnie możemy jednak użyć rekurencji, albo używając wersji z funkcją rekurencyjną albo wersji z subrutyną rekurencyjną. Oto wersja z subrutyną rekurencyjną:

```
recursive subroutine wylicz_silnia(n, silnia)
implicit none;
integer, intent(in)   :: n
integer, intent(out) :: silnia
!!!!!!!!!!!!!!!!!!!!!
if (n  <  0)  then
write(*, *) ' blad: niewlasciwy argument silnia'
stop                                    !!! zatrzymujemy program
end if;

if (n  <  2) then                       !!! silnia dla n=0 oraz n=1
silnia = 1;  return                     !!! przerywamy;  koniec subrutyny
end if
                                        !!!
                                        !!! teraz dla n=2, 3,….
     call wylicz_silnia(n-1, silnia)    !!! wywołujemy   (n-1)!
     silnia = silnia*n                  !!! i wyliczamy n! stosując wzór
                                        !!! rekurencyjny
end subroutine wylicz_silnia
```

Uwaga! n! jest funkcją rosnącą w "zawrotnym" tempie. Podanych powyżej "uproszczonych" procedur możemy używać jedynie dla bardzo małych wartości argumentu n (np. do 8) . Dla większych argumentów trzeba wziąć za wartość funkcji nie zwyczajne **integer**, ale **real** podwójnej dokładności, a i wtedy tzw. *overflow* (zbyt duże wartości funkcji) występują dla całkiem niedużych wartości argumentu.

I.B.5.2. Funkcje rekurencyjne

Ta sama silnia może być zapisana w wersji funkcyjnej. Dodajemy wtedy słowo kluczowe **result** w deklaracji funkcji. Dodatkowo (nawiązując do uwag podanych powyżej) dokonujemy teraz obliczeń w podwójnej precyzji:

```
recursive function silnia(n) result(n_silnia)
implicit none;
integer, parameter :: DP= selected_real_kind(15)
integer, intent(in) :: n
```

```
real(kind=DP) :: n_silnia
!!!!!!!!!!!!!!!!!!!!
if (n  <  0)  then
write(*, *) ' blad: niewlasciwy argument silnia'
stop                                        !!! zatrzymujemy program
end if;
!!!
if (n < 2) then                             !!! dla n=0 oraz n=1
n_silnia = 1;  return                       !!! przerywamy i koniec obliczeń
end if
                                            !!!
                                            !!! teraz dla  n=2, 3, 4,...
n _silnia = real(n, kind=DP) * silnia(n-1)
end function silnia
```

Mimo, że zaczyna to być już trochę nudne jednak znów przypominamy: blok wywołujący procedurę rekursyjną musi mieć INTERFACE (albo funkcje rekursyjne są umieszczone w module, co już wystarcza, bo procedury modułowe mają z definicji określony INTERFACE - patrz część I. C. skryptu).

Następny przykład prostej funkcji rekurencyjnej, to funkcja obliczająca wartość symbolu Newtona *N(n,k)= n!/[k! (n-k)!];* skorzystamy tu z rekurencyjnego wzoru (trójkąt Pascala) *N(n+1,k+1) = N(n,k) + N(n,k+1):*

```
recursive function newton(n,k)  result(n_po_k)
integer, parameter :: DP=selected_real_kind(15)
real(kind=DP) :: n_po_k
integer, intent(in) :: n, k
if ( k < 0  .or.  k > n  .or.  n < 0)  then
n_po_k = 0.0_DP
return
end if
                                            !!! przyjmujemy, że tzw. niewłaściwe
                                            !!! symbole Newtona zerują się
!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
if (n < 2) then
n_po_k = 1.0_DP
return
end if
                                            !!! przyjmujemy, że dla n=0 oraz dla n=1
                                            !!! symbole Newtona są równe 1
!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
                                            !!! a  teraz rekurencja:
n_po_k = newton(n-1,k) + newton(n-1,k-1)
end function newton
```

I.B.5.3. Dalsze przykłady

Kolejny przykład dotyczy szybkiego rekurencyjnego sortowania (tzw. quicksort) ciągu liczb. Kolejne wyrazy posortowanego ciągu powinny rosnąć (a przynajmniej nie maleć). Metoda quicksortu jest prosta i szybka, ale kosztowna jeśli idzie o pamięć komputera. Polega ona na wybraniu dowolnego elementu (np. pierwszego) i nazwaniu go "środkowym" (wyliczenie prawdziwego środkowego elementu jest nie mniej trudne niż samo sortowanie). Umieszczamy go w centralnym miejscu, następnie na lewo układamy elementy mniejsze a na prawo większe. Do "środkowego" elementu dostawiamy dodatkowe elementy o tej samej wartości co element środkowy (o ile takie istnieją). Jak widać oryginalny ciąg składa się teraz z podciągów: lewego, środkowego i prawego. Mamy też już pewne szczątkowe uporządkowanie. Teraz całą procedurę powtarzamy osobno dla lewego ciągu i osobno dla prawego itd. tak długo, aż ciąg zmieni się na ciąg całkowicie uporządkowany.

Ze względu na dynamiczne wywoływanie procedur i związaną z tym "administrację" stosu, metoda quicksortu staje się opłacalna do sortowania dopiero dla ciągów powyżej 8 elementowych (mniej więcej). Dla ciągów krótkich sortowanie najlepiej jest robić metodą wprost. W podanym poniżej przykładzie zostosujemy subrutynę "sortuj8" do sortowania krótkich ciągów a rekurencyjną subrutynę "sortuj" do sortowania długich ciągów.

Ze względów dydaktycznych "sortuj8" jest napisana w sposób bardzo rozwlekły i nieekonomiczny, za to (mamy nadzieję) całkowicie zrozumiały. Dla osób które preferują ekonomię, profesjonalizm i oszczędność środków polecamy zamiast "sortuj8" króciutki i prosty program oparty o metodę Shella (np. z Numerical Recipes).

Teraz przejdźmy do naszego programu i przykładu:

```
program testuj_sortowanie
implicit none;
integer, parameter :: N_max=100              !!! maksymalna liczba elementów do
                                             !!! sortowania
real, dimension(N_max) :: zbior_liczb
integer ::  j, N
INTERFACE
recursive subroutine sortuj(zbior)
real, dimension(:), intent(inout) :: zbior
end subroutine sortuj
END INTERFACE
!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
open(55, file='sortuj.dat')
                                             !!!!!!!!!!!!!!!!!!!!!!!!!!
read(55, *)   N
do j=1, N
read(55, *)   zbior_liczb(j)
end do;                                      !!! wczytywanie zbioru z pliku
                                             !!! w każdej linii   jedna liczba
close(55)
!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
write(*, 4444)  zbior_liczb(1:N)             !!! wydruk kontrolny na ekran
4444   format(4F20.8)                        !!! po 4 liczby w linii
write(*, *) ' ------------------------------------'
!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
call sortuj(zbior_liczb(1:N))
!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
```

```
write(*, *) '  posortowany zbior : '
write(*, 4444)  zbior_liczb(1:N)              !!! wydruk posortowanego zbioru
stop
end program testuj_sortowanie
!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
subroutine sortuj8(zbior)                     !!! sortowanie  wprost przez wstawianie
real, dimension(:), intent(inout) :: zbior
!!!!!    lokalne zmienne
real, dimension(8) :: bufor, bufor2           !!! macierze pomocnicze
real :: element                               !!! zmienna pomocnicza
integer :: N,  j,  k
N=size(zbior)
!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
if (N  >  8) then                             !!! zabezpieczenie
write(*, *) '  sortuj8:     N  za duze';   stop
end if
!!!!!!!!!!!!!!!!!!!!!!!!!!!!
if (N  .eq.  1)  then                         !!! jednego elementu nie sortujemy
return                                        !!! kończymy
end if
!!!!!!!!!!!!!!!!!!!!!!!!!!                    !!! jeżeli program dalej działa oznacza to,
                                              !!! że   N >1

bufor(1:N) = zbior                            !!! kopiujemy zbiór do macierzy
                                              !!! pomocniczej,

                                              !!! pętla  po j   dokona uporządkowania
                                              !!! elementów w macierzy “bufor”
   petla_j : do   j=2, N;  element=bufor(j)
                                              !!! zakładamy, że elementy od 1 do j-1 są
                                              !!! już (względem siebie) posortowane,
                                              !!! jeśli nowy j-ty element jest większy
                                              !!! (lub równy) niż k-ty (z poprzednich
                                              !!! elementów)
                                              !!! oraz mniejszy (lub równy) od k+1  to
                                              !!! element j-ty wstawiamy w środek;
                                              !!! wyjątki: element< bufor(1)
                                              !!! (wtedy element wstawiamy z lewej)
                                              !!! oraz   element>bufor(j-1)
                                              !!! (wtedy element wstawiamy z prawej)
         !!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
         if (bufor(1)  .ge.  element) then    !!! wyjątek nr 1
         bufor2(1) = element
         bufor2(2:j) = bufor(1:j-1)
         bufor(1:j)=bufor2(1:j);   cycle petla_j
                                              !!! skok do nowej iteracji (dla  j większego
                                              !!! o  jeden), dalsze instrukcje aż do
                                              !!! “end do” są opuszczone
```

```
        end if
        !!!!!!!!!!!!!!!!!!!!!!!!!!!!                    !!!  wyjątek nr 2
        if (bufor(j)  .le.  element) then
        cycle petla_j
                                                        !!! nic nie trzeba robić, bo element j-ty jest
                                                        !!! od razu uporządkowany
                                                        !!! opuszczamy instrukcje u dołu i
                                                        !!! rozpoczynamy  nową iterację dla
                                                        !!! następnego j
        end if


        !!!!!!!!!!!!!!!!!!!!!!!!!!!!                    !!! a tutaj normalna sytuacja
        do k=1, j-1;
        if (bufor(k)  .ge.  element  &
            .and. bufor(k+1)  <  element)  then
        bufor2(1:k) = bufor(1:k)
        bufor2(k+1) = element
        bufor2(k+2:j) = bufor(k+1:j-1)
        bufor(1:j) = bufor2(1:j)
        cycle petla_j
        end if
        end do;
        !!!!!!!!!!!!!!!!!!!!!!!!!!!!
    end do petla_j
zbior(1:N) = bufor(1:N)
end subroutine sortuj8
!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
recursive subroutine sortuj(zbior)                      !!! właściwa subrutyna sortująca dla
                                                        !!! długich  ciągów
real, dimension(:), intent(inout) :: zbior
                                                        !!! teraz lokalne zmienne
integer :: j,  N, N_lewy, N_prawy, N_srodek
integer ::  licznik_lewy, licznik_prawy
                                                        !!! uwaga:  N = size(zbior)     (liczba
                                                        !!! elementów zbioru)
real ::  srodek, element
real,dimension(:), allocatable ::lewy, prawy            !!! macierze tworzone dynamicznie
!!!!!!!!!!!!!!!!!!!!!!
   interface
   subroutine sortuj8(zbior)
   real, dimension(:), intent(inout) :: zbior
   end subroutine sortuj8
   end interface
!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
N=size(zbior)
if   (N  .le.  8)  then                                 !!! ciąg krótki; używamy sortuj8
```

```
call sortuj8(zbior)
return                                  !!! i  kończymy
end if
!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!

srodek=zbior(1);                        !!! przyjęto roboczą hipotezę, że zbior(1)
                                        !!! jest w przybliżeniu średnim
                                        !!! (środkowym)  elementem

N_srodek=1;  N_lewy=0;  N_prawy=0;
                                        !!! zliczamy, ile elementów będzie na lewo i
                                        !!! na prawo  od elementu "srodek"
do j=2, N;  element=zbior(j)
   if (element  <  srodek) then
    N_lewy=1+N_lewy
    else if(element  >  srodek) then
        N_prawy =  1+N_prawy
        else
        N_srodek = 1+N_srodek
    end if
end do;
                                        !!! tworzymy macierze dla lewego i prawego
                                        !!! podciągu
if (N_lewy > 0)     allocate( lewy(N_lewy));
if (N_prawy > 0)  allocate( prawy(N_prawy));
!!!!!!!!!!!!!!!!!!!!!!!
licznik_lewy=0;  licznik_prawy=0
!
do j=2, N;  element=zbior(j)            !!! wypełniamy macierze "lewy"  oraz
                                        !!! "prawy"
  if (element  <  srodek) then
  licznik_lewy = 1+licznik_lewy
  lewy(licznik_lewy)  = element
  else if (element > srodek) then
      licznik_prawy = 1+licznik_prawy
      prawy(licznik_prawy)=element
  end if
end do;
                                        !!! sortujemy  rekurencyjnie "lewy"  i
                                        !!! "prawy"
if (N_lewy  > 0 ) then
call sortuj(lewy(1:N_lewy) )
zbior(1:N_lewy) = lewy(1:N_lewy)        !!! posortowane elementy wkładamy do
                                        !!! macierzy
                                        !!! zbiór  (oryginalna postać macierzy jest
                                        !!! niszczona)
end if
!!!!!!!!!!!!!!!!!!!!!!
if (N_prawy>0 ) then
call sortuj(prawy(1:N_prawy) )
```

```
zbior(N_lewy+N_srodek+1:N) = prawy(1:N_prawy)
end if
!!!!!!!!!!!!!!!!!!!!!!
zbior(N_lewy+1:N_lewy+N_srodek)=srodek
!!!!!!!!!!!!!!!!!!!!!!!!
end subroutine  sortuj
```

I.B.6. Argumenty warunkowe (optional) w procedurach; słowa kluczowe (keywords)

W rozdziale I.B.4.3. omawialiśmy przykłady procedur. Obecnie przejdziemy do kolejnej nowinki dotyczącej procedur - do argumentów warunkowych, w slangu informatycznym "opcjonalnych" (optional). **Mogą** być używane przy wywoływaniu procedur, **ale nie muszą**. Spotkaliśmy się z nimi już przy kilku okazjach; konkretnie wtedy, gdy używaliśmy funkcji standardowych, które mogły być wywoływanie z jednym, dwoma lub nawet z trzema argumentami. W zależności od tego, ile było argumentów oraz jak były użyte dana funkcja standardowa działała podobnie, lecz jednak nieco inaczej.

W praktyce argumenty warunkowe używane są najczęściej przy szukaniu błędów – czyli przy tzw. debuggingu. W czasie uruchamiania programu bardzo często wskazane jest, aby procedury wykonywały dodatkowe testy, drukowały dane pomocnicze itp. Ale gdy program już działa zadowalająco potrzeba taka znika – dodatkowe wydruki wtedy raczej przeszkadzają. Argumenty warunkowe nadają się idealnie do zastosowania w takiej właśnie sutuacji.

Formalne argumenty opcjonalne deklarujemy w procedurze normalnie, dodając jednak atrybut **optional.** Argumenty aktualne (wywołania) identyfikujemy z opcjonalnymi:

1. przez zachowanie kolejności użycia argumentów – tylko jeżeli jest to jednoznaczne;
2. przez używanie słów kluczowych do identyfikacji argumentów – nie ma wtedy konieczności zachowania naturalnej kolejności argumentów.

Słowem kluczowym jest tu **formalna** nazwa argumentu. Użycie: **formalna_nazwa= argument_aktualny (**patrz przykład poniżej).

Zdrowy rozsądek sugeruje, że wprowadzenie takiej konstrukcji byłoby w programie bezużyteczne, jeśli nie mielibyśmy "w środku" procedury jakiejś możliwości sprawdzenia, które argumenty zostały użyte, a które nie. Możliwości takiej dostarcza nam logiczna funkcja standardowa **present**. W końcu przypominamy o INTERFACE.

W podsumowaniu zaprezentujemy schemat ilustrujący użycie powyższych zasad:

```
program  warukowe_argumenty
implicit none;
real :: a, b, c;
logical :: blad
……
INTERFACE
subroutine  przyklad1(x, dokladnosc, error)
real, intent(inout) :: x
real, intent(in), optional :: dokladnosc
logical, intent(inout), optional :: error
end subroutine przyklad1
```

```
END INTERFACE
……
a=1.0;   b=1.0;  c=1.0;  blad = .false.
……
call przyklad1(a, c, blad)                    !!! tak możemy wołać procedurę
……..
call przyklad1(x=a, dokladnosc=c, error=blad)
                                              !!! identycznie jak powyżej
……..

call przyklad1(x=a, error=blad)               !!! a teraz  nie używamy 2-go argumentu
……
call przyklad1(error=blad, x=a)               !!! kolejność argumentów może
                                              !!! być zmieniona gdyż
                                              !!! słowa kluczowe jednoznacznie
                                              !!! określają, który
                                              !!! argument aktualny należy
                                              !!! skojarzyć
                                              !!! z odpowiednim argumentem
                                              !!! formalnym
stop
end program warunkowe_argumenty
!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
subroutine  przyklad1(x, dokladnosc, error)
real, intent(inout) :: x
real, intent(in), optional :: dokladnosc
logical, intent(inout), optional :: error
……
if  present(dokladnosc) then                  !!! uwaga na użycie  funkcji standardowej
                                              !!! PRESENT !
…….
end if
!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
if  present (error)  then
…….
end if
……
end  subroutine przyklad1
```

Uwaga! Identyfikacja argumentów za pomocą słów kluczowych jest możliwa zawsze. Polecamy ją jako użyteczną. Argumentami warunkowymi początkujący programista natomiast nie posługuje się nazbyt często…

I.C. Podstawowe wiadomości o modułach: zmienne "globalne"; procedury w modułach

W starym fortranie F77 występowała *bardzo ważna* deklaracja **COMMON,** np.

…….
COMMON/nazwa_commonu/ lista_zmiennych
…….

Występuje ona również (jako spadek po F77) w F90/95, *ale nie będziemy jej używać zastępując ją modułami*.

Tytułem zwięzłej informacji podajemy, że zmienne zadeklarowane jako należące do tego samego COMMON'u (ta sama nazwa_commonu) mogły być przydzielone do kilku bloków (np. głównego bloku oraz kilku bloków procedur) przez powtórzenie w nich tej samej deklaracji COMMON (najlepiej przez skopiowanie) w odpowiednich blokach, a skutkiem było powstanie wspólnego (fizycznie) dla tych bloków obszaru pamięci zawierającego zmienne z *listy_zmiennych*. Zmiana wartości jakiejś zmiennej należącej do COMMONU np. w bloku nr. 1 jest natychmiastowo odczuwalna np. w bloku nr. 2. Innymi słowy było to stworzenie zmiennych globalnych (w odróżnieniu od generalnej zasady, że wszystkie zmienne są lokalne dla danego bloku).

Uwaga! Jeżeli już musimy używać Common (programy adoptowane ze starego F77), to nie używajmy modułów i innych nowych konstrukcji F90/95. Oczywiście, jest to możliwe przy zachowaniu pewnych reguł, ale bardziej prawdopodobne będzie, że może się to zakończyć dla początkującego programisty w opłakany sposób. Zaistnieją błędy, których nawet nie będziemy w stanie dokładnie zdiagnozować.

I.C.1. Najprostszy przypadek : same zmienne, brak procedur w module. Zmienne globalne tworzymy przez umieszczenie ich w module

Zasada jest prosta: pomiędzy

module nazwa_modulu …….. end module nazwa_modulu

umieszczamy deklaracje zmiennych i parametrów. Moduł jeśli występuje w tym samym pliku źródłowym to musi wystąpić PRZED tym naszym programem, który go będzie wykorzystywał; jeśli moduł jest w innym pliku, to ten plik musi być wcześniej skompilowany i w odpowiedni sposób (na poziomie "linkera", czyli programu konsolidującego) podłączony do naszego programu.

Blok, lub kilka bloków wykorzystujących **modul,** muszą jako swoją PIERWSZĄ deklarację mieć "**use nazwa_modulu".** Powoduje to, że zmienne modułowe będą w tym bloku określone jako zmienne globalne *z tymi samymi identyfikatorami zmiennych* (tę zasadę zresztą można obejść), co identyfikatory zmiennych występujące w module.

Jeżeli jest tylko taka możliwość (i jeśli nie jest to niewygodne dla programisty) możemy w procedurach zupełnie nie używać argumentów formalnych, a zamiast nich używać zmiennych globalnych (z odpowiedniego modułu).

Uwaga na ogólne zasady naczelne:

1. *Dany blok naszego programu może wykorzystywać zero, jeden lub kilka modułów (np. use nazwa_modulu1; use nazwa_modulu2; ….)*
2. *Sam moduł może wykorzystywać (zawierać) inny (wcześniejszy) moduł - zawiera wtedy zmienne globalne swoje i "superglobalne", czyli właśnie te z wykorzystanego modułu*
3. *Hierarchia "moduł wykorzystujący moduł, itd " nie może się ani bezpośrednio ani pośrednio "zamykać"; NIEDOZWOLONA jest sytuacja, gdy moduł odwołuje się (pośrednio lub bezpośrednio) sam do siebie. Uwaga: jest to częste źródło błędów u początkujących programistów. Wniosek: jeśli jesteś początkującym programistą, to nie używaj więcej niż jeden moduł w całym swoim programie!*

Sposób utworzenia modułu i wykorzystania go zademonstrujemy na (jak zwykle) dosyć prostym przykładzie. Jest to zagadnienie aproksymacji wielu funkcji nieparzystych zależnych od zmiennej kątowej (w radianach) o wartościach określonych na równo oddalonych "węzłach" (patrz np. J. Legras, Praktyczne metody analizy numerycznej, WNT, Warszawa, 1974). Konkretnie: niech funkcja nieparzysta $f(\varphi)$ będzie określona na odcinku $[-\pi,\pi]$. Jej wartości znamy jedynie dla wartości argumentów: $\varphi_j = j\pi/(n+1)$ gdzie $j=1,2,3,...,n$ (n jest liczbą węzłów, np. n=20) czyli znamy jedynie wartości $f(\varphi_j)$. Wielomianem aproksymującym f będzie miał postać $f(\varphi) \cong c_1 \sin(\varphi) + c_2 \sin(2\varphi) + c_3 \sin(3\varphi) + ...+ c_n \sin(n\varphi)$ gdzie $c_j = 2\, d_j/(n+1)$, $j=1,2,3,...,n$ oraz $d_j = \sin(j\,\varphi_1) f(\varphi_1) + \sin(j\,\varphi_2) f(\varphi_2) + \sin(j\,\varphi_3) f(\varphi_3) +.....+ \sin(j\,\varphi_n) f(\varphi_n)$.

A teraz nasze zadanie: w plikach dane1.txt, dane2.txt, dane3.txt znajdują się dane pomiarowe : po 100 punktów (nieparzystych) wartości funkcji (dla 3-ch różnych funkcji) i dla tego samego układu węzłów co podany powyżej. My jedynie chcemy znaleźć wzory aproksymacyjne dla każdego przypadku, czyli wartości liczbowe współczynników $c_1, c_2 , ... c_{100}$ i zapisać je na plikach, np. o nazwach" funk1.apr, funk2.apr, funk3.apr.

Jak widać z podanych wzorów, aby wyliczyć współczynniki "c" za każdym razem będziemy musieli pomocniczo obliczać te same wartości $\sin(j\,\varphi_k)$. Ułatwimy więc sobie życie i posłużymy się modułem, wyliczymy wartości funkcji tylko jeden raz i zmagazynujemy je w macierzy "sinusy".

```
Module   stale                          !!! pierwszy moduł
implicit none;
integer, parameter :: DP=selected_real_kind(15)
real(kind=DP), parameter :: pi=3.14159265359_DP
end module stale
!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
Module  zmienne                         !!! drugi moduł ( korzysta z pierwszego)
use stale
implicit none;
integer, parameter :: n=20
integer :: i,  j
real(kind=DP), dimension(n, n) :: sinusy
real(kind=DP), dimension(n) :: c, fi,  funkcja
end module zmienne
!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
program  aproksymacja
use zmienne
```

```
!!!
  interface                                    !!!  subrutyny nie mają w ogóle
                                               !!! argumentów; śmiało moglibyśmy
                                               !!!  więc (w tym przypadku)
                                               !!! opuścić INTERFACE
  subroutine oblicz_sinusy
  use zmienne
  end subroutine oblicz_sinusy
  !!!
  subroutine aproksymuj
  use zmienne
  end subroutine aproksymuj
  end interface
                                               !!! uwaga:
                                               !!! w obecnym przypadku
                                               !!! bloki subrutyn nie posługują
                                               !!! się zmiennymi lokalnymi (tylko
                                               !!! globalnymi zmiennymi
                                               !!! z modułów), co jest dosyć często
                                               !!! stosowaną praktyką

call   oblicz_sinusy
!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!   pierwsza funkcja:
open(55, file='dane1.txt')                     !!! wczytujemy dane
do j=1, n;  read(55, *) funkcja(j);   end do;
close(55)
call aproksymuj
!!!
open(66, file='funk1.apr')
do j=1, n;   write(66, fmt='(F15.8)') c(j);  end do;
                                               !!! zapisujemy wyniki
close(66)
!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!! druga funkcja:
open(55, file='dane2.txt')
do j=1, n;  read(55,*) funkcja(j);   end do;
close(55)
call aproksymuj
!!!
open(66, file='funk2.apr')
do j=1, n;   write(66, fmt='(F15.8)')  c(j);   end do;
close(66)
!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!! trzecia funkcja:
open(55, file='dane3.txt')
do j=1, n;   read(55, *) funkcja(j);   end do;
close(55)
call aproksymuj
!!!
open(66, file='funk3.apr')
do j=1, n;   write(66, fmt='(F15.8)')  c(j);   end do;
close(66)
```

```
!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
stop
end program  aproksymacja
!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!

subroutine oblicz_sinusy
use zmienne
!!!
do j=1, n
fi(j)= pi*real(j, kind=DP)/real(n+1, kind=DP)
                                          !!! wyliczamy wartości węzłów
end do;
!!!
do j=1, n;   do k=1, n;
sinusy(j, k) = 2.0_DP *  &
sin( real(j, kind=DP)* fi(k) )/ real(n+1, kind=DP)
                                          !!! wyliczamy wartości sinusów
end do;   end do;                         !!! w węzłach
end subroutine oblicz_sinusy
!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
subroutine aproksymuj
use zmienne
!!!
do j=1, n
c(j) = sum(  sinusy(j,:) * funkcja(:)  )  !!! tutaj wyliczamy wartości
                                          !!! współczynników  c
end do;
end subroutine aproksymuj
```

Jak wcześniej wspomnieliśmy, niektórzy programiści w czasie pisania krótkich programów (takich do policzenia czegoś prostego, "natychmiast") w ogóle nie używają innych zmiennych niż modułowe. Piszą oni np. jeden moduł i wkładają do niego wszystkie zmienne, których chcieliby używać w programie głównym oraz w blokach procedur. Potem go dołączają do każdego bloku. Ta procedura nie jest być może godna zalecania ze względu na nabywanie złych przyzwyczajeń – niemniej jednak (w praktyce fortranowskiej) jest stosunkowo często spotykana.

I.C.2. Drugi prosty przypadek: zmienne oraz procedury w module, czyli tworzenie bloków z narzędziami (do obsługi naszych programów)

Jeżeli już skończyliśmy deklarować w module nasze zmienne i parametry, to możemy napisać słowo kluczowe **CONTAINS,** a po nim włączyć do modułu nasze bloki funkcji i subrutyn (do tej pory takie bloki były zewnętrzne, tzn. stały poza głównym pogramem; z wyjątkiem "lokalnych" funkcji wewnętrzych – patrz podrozdział I.B.2).

Umieszczenie procedury (funkcji lub subrutyny) w module jest korzystane ponieważ:

1. *ma ona dostęp do wszystkich zmiennych zadeklarowanych w module (jako do "swoich zmiennych globalnych") - dosyć podobnie jak to miało miejsce dla procedur wewnętrznych (porównaj - I.B.2)*

2. *nie trzeba (a faktycznie NIE WOLNO) wprowadzać INTERFACE (ani w module ani w blokach programu wykorzystujących ten moduł), gdyż z definicji wszystkie procedury z modułu mają względem siebie automatycznie określony interface (tzw. przypadek "explicite interface").*

Z tego właśnie powodu programiści w F90/95 nieczęsto stosują interface – wolą zamiast tego umieszczać swoje procedury w modułach. Interface stosują tylko wtedy gdy muszą, np. jeśli włączają do swojego programu fragmenty cudzych programów – jako procedury zewnętrzne. Wtedy interface jest bardzo zalecany (a w wielu przypadkach jest absolutnie konieczny).

Tę zaletę modułów docenimy naprawdę dopiero wtedy, gdy nauczymy się nowej klasy zmiennych - *tzw. "derived types" - czyli typów pochodnych, lub złożonych*. Ewentualne pisanie bloków interface dla takich typów zmiennych byłoby nadmiernie uciążliwe (byłyby one niesamowicie długie) - a więc nikt tego nie robi, bowiem wszyscy deklarują takie złożone zmienne i procedury do ich obsługi w odpowiednich modułach.

Tytułem wstępnej informacji: derived types to "zbiór" wielu zmiennych występujących pod jedną nazwą; jako pojedyncze składowe czyli swoje "pola" ma zmienne proste, które już poznaliśmy, czyli logical, integer, real, complex, character i macierze, oraz ewentualnie jeszcze nie przedyskutowane wskaźniki , czyli inaczej pointery.

Przykład modułu (ze zmiennymi i z procedurami), który podamy obecnie, będzie sztuczny (ale za to bardzo krótki):

```
module narzedzia
implicit none;
integer, parameter :: N= 2
integer :: j
real ::   alfa, beta
complex, dimension(N, N) ::  z1, z2, z3, z4, z5, z6
 CONTAINS
     subroutine dodaj_macierze(x1, x2, x3)
     complex, dimension(N, N),  intent(inout) :: x1, x2, x3
     x3 = x1 + x2
    end subroutine dodaj_macierze
   !!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
   function mnozenie_macierzy(x1, x2)
   complex, dimension(N, N), intent(in) :: x1, x2
   complex, dimension(N, N) :: mnozenie_macierzy
   mnozenie_macierzy=matmul(x1, x2)
  end function mnozenie_macierzy
  !!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
  subroutine   wypisuj(x1)
  complex, dimension(N, N), intent(in) :: x1
  do j=1, N                                    !!! posługujemy się j, zadeklarowanym
                                               !!! wcześniej w module
  write(*, fmt='(2F16.8, 6x,2f16.8)'  )   x1(j, :)
 end do
                                               !!! 6x w formacie oznacza – zrób 6
                                               !!! odstępów,
                                               !!! subrutyna wypisuj wydrukuje na
                                               !!! ekranie macierz x1 (w/g kolejności
```

```
                                                  !!! wierszy  x1)
                                                  !!! po 2 zespolone elementy (oddzielone od
                                                  !!! siebie 6 odstępami) na jedną linie ekranu
 end subroutine wypisuj
!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
end module narzedzia
!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
program   przyklad_I_C_2
use narzedzia
open(55, file='z1.dat');   read(55, fmt='(4F16.8)')  z1
                                                  !!! wczytanie , kolumnami po dwie liczby
                                                  !!! zespolone (a więc po 4-y liczby real)
close(55)
!!!
open(55, file='z2.dat');   read(55, fmt='(4F16.8)')  z2;   close(55)
!!!
open(55, file='z3.dat');   read(55, fmt='(4F16.8)')  z3;   close(55)
!!!
                                                  !!! teraz liczymy   z1 + z2;   z1 + z3
                                                  !!! i drukujemy na ekranie
call  dodaj_macierze(z1, z2, z5);   call wypisuj(z5)
call  dodaj_macierze(z1, z3, z5);   call wypisuj(z5)

                                                  !!! teraz obliczoną wartość iloczynu z1*z3
                                                  !!! wydrukujemy na ekranie
wypisuj( mnozenie_macierzy(z1, z3) )
stop
end program przyklad_I_C_2
```

II. Z krótką wizytą na średnim poziomie zaawansowania

II.A. Nowe typy zmiennych

II.A.1. Definiowanie, deklarowanie i używanie typów "pochodnych", inaczej "złożonych" lub "strukturalnych" (derived type)

"Derived types" są typem zmiennych z którymi większość Czytelników spotkała się już w szkole – przy nauce Pascala. Nazywały się wtedy rekordami. Podobny typ istnieje również w języku C i w wielu innych językach programowania. Są to zmienne idealnie nadające się do obsługi małych (a także dużych) baz danych – pod jedną nazwą zmiennej magazynujemy w poszczególnych "polach" praktycznie dowolną ilość informacji różnego typu.

Zmienne pochodne muszą oczywiście najpierw być zdefiniowane (wcześniej od deklaracji zmiennych); dopiero do ukończeniu definicji możemy je deklarować. Po zadeklarowaniu możemy już za nie podstawiać konkretne dane i używać ich.

Dla koncentracji uwagi przyjmijmy, że zajmujemy się katalogowaniem materiałów (części) elektronicznych. Właśnie konkretny katalogowany "materiał" będzie naszym przykładem.

Definiowanie i deklarowanie

```
……..
type material                                  !!! definiujemy nowy typ i jego
                                               !!! identyfikator
   character(len=20) ::   nazwa                !!! "nazwa" to jego pierwsza składowa
                                               !!! (inaczej  "identyfikator"
                                               !!! pierwszego pola  typu "materiał")
   real ::    cena
   integer :: pozostalo_w_magazynie
end type material

!!!!!!!!!!!!!!!!!  teraz inny typ:
type adres                                     !!! przyda się do katalogowania adresów
                                               !!! dostawców i producentów
character(len=50) :: nazwa_adresata
character(len=20) ::   miasto, ulica
integer ::  numer_domu
character(len=6) :: kod_pocztowy
end type adres

……..
……..
                                               !!!  teraz można juz zadeklarować
type(material) ::  czesc1, czesc2, mater       !!!  zmienne
type(material), dimension(100) :: uklady_scalone
                                               !!! a  teraz deklarujemy macierz
……
type (adres) :: adres_poczty, adres_celny
type(adres), dimension(50) :: producenci, dostawcy
……..
                                               !!! przerywamy
```

Nadawanie określonej wartości danej zmiennej odbywa się następująco: "zbiorowo", za pomocą specjalnego konstrukora (o identycznej nazwie jak nazwa definicyjna typu):

```
                                               !!! kontynuujemy nasz przykład
czesc1 =  material('' opornik 100 ohm'', 0.85, 7)
!!!
adres_poczty= adres('' Poczta osiedlowa'', '' Nowe Miasto'', '' ul. Osiedlowa'', &
                    12,  '' 30-658'')
……..
```

albo też po kolei przez podstawianie poszczególnych pól. Poszczególne pola są udostępniane, jeśli stosujemy następującą składnię: **nazwa_zmiennej%nazwa _pola**. Tak więc to samo co zrobiliśmy "zbiorowo" u góry możemy zrobić w sposób następujący:

```
…..
czesc1%nazwa = ’ opornik 100 ohm’
czesc1%cena =  0.85
czesc1%pozostalo_w_magazynie = 7
                                                !!! teraz podstawimy dane
                                                !!! adresowe dostawcy nr. 5
                                                !!! (piąty element macierzy dostawcy)

dostawcy(5)%nazwa_adresata = ’  Firma  Nowak& Synowie’
dostawcy(5)%miasto =  ’ Stara Wies ’; dostawcy(5)%ulica = ’’
                                                !!! brak ulicy  (napis o długości zero)
dostawcy(5)%numer_domu = 357
                                                !!! dostawcy(5)%kod_pocztowy = ’??? ’
                                                !!! brak danych
                                                !!! a więc nie podstawiamy  NIC
```

Na poszczególnych polach możemy oczywiście robić dowolne (dozwolone) operacje – stosownie do typu zmiennej polowej: np. chcemy doliczyć do ceny opornika podatek VAT (22 procent), oraz zmienić liczbę części pozostałych w magazynie (bo np. jeden opornik już sprzedaliśmy).

```
czesc1%cena = czesc1%cena*1.2;
czesc1%pozostalo_w_magazynie = czesc1%pozostalo_w_magazynie - 1
```

Jak widać czesc1%cena zachowuje się jak zwykła zmienna typu real. Jeśli nie posługujemy się polami tylko “gołą” nazwą zmiennej złożonej, to jedyna możliwa w takiej sytuacji operacja, to podstawienie jednej zmiennej za drugą, np.

```
czesc2 = czesc1
uklady_scalone(12) =  czesc1
```
itp.

Uwaga! *Derived types* jako swoje pola mogą posiadać nie tylko zmienne skalarne ale też macierze (tylko o określonych i stałych granicach, tzn. “explicite shape” array) oraz inne (wcześniej zdefiniowane) typy.

Prosty (może niepraktyczny ?) przykład: grafik komputerowy chce zdefiniować jako swoje narzędzie nowy typ: koła w przestrzeni 2-wymiarowej; każde koło określone jest przez promień, położenie środka, oraz kolor (do wypełnienia wnętrza koła) podany jako kombinacja natężenia (w umownych jednostkach) podstawowych kolorów, tzn. czerwonego, zielonego, niebieskiego (każdy kolor o natężeniu 0, 1, 2 lub 3 – czyli cztery poziomy intensywności). Można to zrobić, dla przykładu, tak:

```
type punkt
real ::  x, y                                   !!!  współrzędne punktu
end type punkt

!!!!!!!!!!
type kolo
```

```
type(punkt) :: srodek
real ::  promien
integer :: czerwony, zielony, niebieski
end type kolo

                                                !!! teraz deklaracje zmiennych
type(punkt) ::  r0, r1, r2, r3
type(punkt), dimension(6) :: szesciokat
!!!
type(kolo) ::   centralne,  pomocnicze
type(kolo), dimension(1000) :: moje_tlo
!!!!
……..
r0%x = 0.0;    r0%y = 0.0;
centralne%srodek =  r0                          !!! podstawienie od razu całego (złożonego)
                                                !!! r0
centralne%promien =   5.0
centalne%niebieski = 0                          !!! nie musimy podstawiać pol “po kolei”
centralne%czerwony = 3
centralne%zielony = centralne%czerwony
```

Można to robić inaczej, np. nie musimy w ogóle posługiwać się pomocniczą zmienną r0, to samo zrobimy “używając kilka razy” znaku procentu:

```
centalne%srodek%x = 0.0
centralne%srodek%y=0.0
```

Jako ostatni będzie podany przykład z macierzą jako składnikiem typu złożonego. Wiadomo, że krzywe stopnia pierwszego oraz drugiego na płaszczyźnie xy mogą być zadane równaniem $a_{11}\, x^2 + a_{22}\, y^2 + 2\, a_{12}\, x\, y + b_1\, x + b_2\, y + c = 0$, gdzie a_{ij} to elementy macierzy 2 x 2 , b_j to elementy macierzy 2x1 (jedna kolumna) , c - jest skalarem. Możemy więc określić:

```
type krzywa_2_stopnia
  character(len=20) :: nazwa_krzywej
  real, dimension(2,2)  ::  a
  real, dimension(2) :: b
  real :: c
end type krzywa_2_stopnia
!!!!
type(krzywa_2_stopnia) ::   elipsa,  prosta    !!!   deklaracja zmiennych
……..
elipsa%nazwa_krzywej = ’ elipsa’
eplisa%a = reshape( (/2.0,  0.0,  0.0,  1.0/), (/2, 2/) )
elipsa%b = (/0.0, 0.0/)  ;   elispa%c = -1.0
prosta%nazwa_krzywej = ’  prosta ’
prosta%a = 0.0;    prosta%c=0.0;    prosta%b = (/ 1.0,   -1.0/)
…….
```

Uwaga! *Używanie typów pochodnych jako argumentów procedur stwarza trudności z INTERFACEM. Aby ich uniknąć od tej pory przyjmiemy zasadę, że zarówno definicje "derived types" oraz procedury na nich operujące umieszczone będą w modułach.*

Pokażemy teraz przykład, który będzie odnosić się do operowania wielomianami. Wielomian n-tego stopnia $w(x) = c_0 + c_1 x + c_2 x^2 + c_3 x^3 + \ldots. c_n x^n$ możemy określić przez podanie jego stopnia n (przyjmijmy, że jest nie większy niż 100) oraz zbioru współczynników $c_0, c_1, c_2, c_3,...$ Dla wielomianów można łatwo określić:

1. dodawanie 2-ch wielomianów,
2. mnożenie 2-ch wielomianów,
3. mnożenie wielomianu przez liczbę.

Powiedzmy, że chcemy analitycznie (ściśle) wyliczać całki (w granicach od zera do jeden) z takich wielomianów, drukowac dane dotyczące wielomianów, itp.

$$\int_0^1 w(x)dx = c_0 + c_1 x/2 + c_2 x/3 + c_3 x/4 + \ldots$$

A oto nasz programik:

```
module wielomiany_narzedzia
implicit none;
integer, parameter :: N=100
real, parameter :: zero=0.0

  type  wielomian
   integer ::   stopien
   real, dimension(0:N) :: wspolczynniki
  end type wielomian

CONTAINS
 function dodaj(w1, w2)
 type(wielomian) :: dodaj                    !!! typ "wielomian" był zdefiniowany
                                             !!! wcześniej w module; możemy więc teraz
                                             !!! deklarować zmienne
                                             !!! o typie  ''wielomian''
 type(wielomian), intent(in) :: w1, w2
 integer ::  N_max
 N_max = max(w1%stopien, w2%stopien)
 dodaj%stopien = N_max
 dodaj%wspolczynniki(0:N_max) = w1%wspolczynniki(0:N_max) + &
 w2%wspolczynniki(0:N_max)
 end function dodaj
!!!!!!!!!!!!!!!!!!!!!!!!!!!
 function mnozenie(w1, w2)
 type(wielomian) :: mnozenie
 type(wielomian), intent(in) :: w1, w2
 real :: suma
 integer :: j,  k,  N_max
 N_max = w1%stopien * w2%stopien
```

```
     do k=0,N_max
       suma=zero
       do j=0, k
       suma=suma+w1%wspolczynniki(j)* &
        w2%wspolczynniki(k-j)
        end do
       mnozenie%wspolczynniki(k) = suma
      end do
  end function mnozenie
!!!!!!!!!!!!!!!!!!!!!!!!!
  subroutine convert(liczba, wiel)
  type(wielomian), intent(inout) :: wiel
  real,intent(in) :: liczba
  wiel%stopien = 0
  wiel%wspolczynniki(0) = liczba
  end subroutine convert
!!!!!!!!!!!!!!!!!!!!!!!!!
  real function calka(wiel)
  type(wielomian), intent(in) :: wiel
  real :: suma
  integer :: j
  suma=zero
  do j=0,wiel%stopien
  suma=suma+ wiel%wspolczynniki(j) /real(j+1)
  end do
  calka=suma
  end function calka
end module  wielomiany_narzedzia
!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
program  test_wielomianow
use wielomiany_narzedzia
implicit none;
integer :: j
real, dimension(0:N) :: macierz_pom
type(wielomian) :: stala, w1, w2
                                        !!! w1, w2  są to robocze zmienne  (definicja
                                        !!! typu wielomian – patrz moduł)
type(wielomian), dimension(3:5) :: w
real, dimension(3:5) :: wartosc_calki
!!!!!!!!!!!!!!!
                                        !!!  podstawiamy  wielomian w1=x+x**2+ ...
                                        !!! + x**8
                                        !!!  oraz   wielomian w2= 1+ x + x**3/4
                                        !!!+ x**4/5

w1%wspolczynniki=zero
w2%wspolczynniki=zero                   !!! wstępnie zerujemy współczyniki (od
                                        !!! zera do 100)
```

```
!!!
macierz_pom=zero
macierz_pom(0:4) = (/1.0, 1.0, 0.0, 0.25, 0.20/)
                                                !!! podstawiamy tylko niezerowe
                                                !!! współczynniki
!!!
w2 = wielomian(4, macierz_pom )                 !!! korzystamy z "konstruktora typu"
                                                !!! podstawiamy 4 jako stopień wielomianu
                                                !!! oraz  jego niezerowe elementy


                                                !!! zamiast używania konstruktora typu
                                                !!! możemy podstawiać osobno
                                                !!! poszczególne  pola:
w1%stopien = 8
do j=1, 8;   w1%wspolczynniki(j)=1.0;   end do;
!!!!!!!!!!!!!!!
                                                !!! będziemy  wyliczać wielomiany w(3) =
                                                !!! w1+w2 oraz  w(4) =  w1*w2 i
                                                !!! w(5) =  2*w1 + w2

                                                !!! a więc:
w(3) = dodaj(w1,w2)
!!!
w(4)=mnozenie(w1,w2)
!!!
call convert(2.0, stala)
w(5)=dodaj( mnozenie(stala,w1), w2)
!!!
                                                !!! teraz wyliczamy całki (w granicach od 0
                                                !!! do 1)   z   w(3),  w(4),   w(5) :

do j=3,5
wartosc_calki(j) = calka(w(j))
write(*, fmt='(''j ='',i3,'' calka(j) = '',F15.8)' )   j, &
            wartosc_calki(j)
end do
!!!!!
                                                !!! dla kontroli  drukujemy współczynniki
                                                !!! w(3)
write(*, *) '  drukujemy wspolczynniki c_j wielomianu &
          &   w(3) = c_0 + c_1*x + c_2*x**2 + ...  '
                                                !!! uwaga (przypomnienie):  kontynuacja
                                                !!! napisu do następnej linii  wymaga
                                                !!! dodatkowego   znaku   &   na początku
                                                !!! drugiej linii

                                                !!! teraz drukujemy na ekranie małą
                                                !!! tabelkę współczynników  wielomianu
                                                !!! w(3)
```

```
write(*, *) ' ****************************************'
write(*, *) '   j          c_j   '
write(*, *) ' --------------------'
do j=0, w(3)%stopien
write(*, fmt='(i3,E15.6)')   j, w(3)%wspolczynniki(j)
end do
stop
end program test_wielomianow
```

II.A.2 Wstępne informacje o zmiennych wskaźnikowych (czyli zmiennych pointerowych lub po prostu pointerach) i o przydzieleniu danemu pointerowi (assigment) zmiennej (o atrybucie target), na którą pointer pokazuje

Zacznijmy od tego, że pointery są bardzo chętnie wykorzystywane w zaawansowanych algorytmach przez informatyków. Natomiast fizyk, chemik, czy inżynier wykorzystuje je nieczęsto, gdyż dla tej właśnie kategorii programistów najważniejsze są regularne struktury, w szczególności macierze i operacje na macierzach. Dlatego w skrypcie dla początkujących programistów nie będziemy pisać o pointerach zbyt dużo. *Naszym zdaniem temat wymaga całkiem osobnego, poważnego i dosyć zaawansowanego wykładu (może są to rzeczy trudne?).*

Niemniej jednak spróbujmy (w ramach zapowiedzi) odpowiedzieć od razu na pytanie: po co są pointery w F90/95 ? Krótka odpowiedź :

1. są bardzo wygodne jako aliasy macierzy i segmentów macierzy;
2. są niezbędne w bazach danych, gdzie składowana informacja miewa **nieregularną** postać;
3. nadają się świetnie do budowania stosów, rozgałęzionych grafów, obsługi dynamicznych kolejek, przeszukiwania nieregularnych struktur itp.

Pointery są znane z Pascala i z języka C. Traktowane są tam jako adresy w pamięci (komputera) co pozwala *na pośredni dostęp* do odpowiednich zmiennych. W F90/95 ten sposób myślenia (tzn. że pointery to po prostu adresy) nie jest całkiem ścisły. Zmienną pointerową określimy w F90/95 po prostu jako zmienną mogącą być aliasem do jakiejś zmiennej, macierzy, fragmentu macierzy, innego pointera (ale odpowiedniego typu) lub innej, dużej i skomplikowanej struktury. (Uwaga: sama zmienna pointerowa - jeżeli idzie o zapotrzebowanie na pamięć komputera - jest na ogół bardzo mała).

Zmienna pointerowa jest dynamicznie stowarzyszona (powiązana) z tymi właśnie danymi lub nie jest stowarzyszona (z niczym) w zależności od decyzji programisty. Konkretny pointer może więc (w danej chwili) znajdować się w jednym z 3-ch stanów:

1. *może być niezdefiniowany (undefined), co jest normą na początku programu, w chwili startu, tzn. jeszcze przed wykonaniem pierwszej instrukcji.*
2. *może być "wyzerowany" (null) czyli niejako "uziemiony" (jeśli użyjemy slangu fizycznego) co oznacza, że chwilowo nie jest aliasem niczego, ale nie jest już niezdefiniowany.*
3. *może być związany, lub stowarzyszony (associated), tzn że jest aliasem czegoś, innymi słowy że wskazuje (lub celuje) na jakąś zmienną tak, jakby celował w "tarczę" (target)*

Jeżeli pointer jest w stanie 2 lub 3 (null lub associated) to dozwolone jest stosowanie standardowej funkcji logicznej **associated(...)** która udzieli odpowiedzi .true. jeśli pointer jest stowarzyszony (asocciated) lub .false. jeżeli pointer jest wyzerowany (nullified). Funkcja ta jest bardzo użyteczna zwłaszcza w procedurach oraz w przypadku dużych, złożonych programów.

Od strony syntaktycznej (gramatyki języka) fakt bycia pointerem określony jest przez atrybut "**pointer"** dodany do typu zmiennej w czasie jej deklaracji. Taki pointer może wskazywać tylko na odpowiedni typ zmiennej pod warunkiem że w czasie deklaracji została ona opatrzona dodatkowym atrybutem **"target"** (lub ewentualnie na pointer tego samego typu).

Przejdźmy do przykładu jak deklarować pointery i obiekty na które one pakazują i co ważniejsze jak używać pointerów w programie.

```
program  pointery1
implicit none
logical ::   is_associated
!!!
integer, pointer ::   ip1, ip2, ip3, pokazuj_na

!!!  zmienne ip1, ip2… mogą być pointerami czyli wskazywać ale tylko i wyłącznie do
!!!  zmiennych typu integer (uwaga: tylko tych opatrzonych atrybutem target),
!!!  ewentualnie do innych pointerow (tego samego typu)
!!!

integer, target  ::   liczba, licznik, j

!!!  jest atrybut target, więc:  zmienne liczba, licznik j   mogą mieć aliasy
!!!  (czyli mogą być wskazywane przez pointery)
!!!
!!!  teraz nie możemy uzyc  funkcji associated(ip1)
!!!  bo  ip1 ma jeszcze status  " undefined"

licznik = 10;   liczba = 77;
!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
ip1 => licznik                          !!! "pointer assigment"-- czyli !!!związanie
                                        !!! zmiennej
                                        !!! pointerowej ze
                                        !!! zmienna licznik; ip1 teraz celuje na
                                        !!! (tarcze)  licznik;
                                        !!! czyli ip1 jest (chwilowo) aliasem
                                        !!! zmiennej licznik
                                        !!! is_associated=associated(ip1)

write(*, fmt='(L4)')  ' is_associated = ', is_associated
                                          !!! to wydrukuje  wartosc: true
write(*, *) ' ip1 wskazuje na zmienna licznik o chwilowej wartości → ', ip1
!!!   zostanie wypisana liczba 10
j =  -9  +  ip1;     write(*, *)  ' j = ', j          !!! zostanie wypisana liczba 1
```

```
                                        !!! uzycie  ip1  w instrukcji pisania lub w
                                        !!! instrukcji  artymetycznej jest legalne
                                        !!! ip1 jest traktowane wtedy jako



                                        !!! zostanie wypisana liczba 10
                                        !!! użycie  ip1  w instrukcji pisania lub w
                                        !!! instr. arytmetycznej jest legalne
                                        !!! ip1 jest traktowane wtedy jako alias
                                        !!! zmiennej
                                        !!! "liczba"  (co zachodzi w wyniku
                                        !!! tzw "wyłuskiwania"; ten oficjalny
                                        !!! termin polski !!! jest  sztuczny;
                                        !!! będziemy raczej używać  oryginalnego
                                        !!! angielskiego "dereferencing")

j =  -9 + ip1;    write(*, *) ' j = ',  j
                                        !!!  zostanie wypisana liczba 1

ip2 = > licznik                         !!! teraz  ip2 tez pokazuje na
                                        !!! licznik (czyli licznik ma  dwa  aliasy)
ip3 = > licznik                         !!! teraz aż trzy
pokazuj_na = > ip2                      !!! pointer pokazuj_na
                                        !!! wskazuje teraz na TO SAMO na co
                                        !!! wskazuje
                                        !!! ip2  (czyli na target : licznik)
nullify(ip2)                            !!! pointer op2 zostal wyzerowany
                                        !!! jest już aliasem    żadnej zmiennej
write(*, *) pokazuj_na                  !!! ale pointer "pokazuj_na" dalej
                                        !!! pokazuje pokazuje na
                                        !!! licznikraz pokazemy inne uzycie  funkcji

                                        !!! standardowej   associated
                                        !!! (nie z jednym, tylko z dwoma
                                        !!! argumentami;
                                        !!! pierwszy to pointer,   drugi target)

is_associated = associated(ip3, licznik)

write(*, fmt = '(L4)') '' is_associated = '',  is_associated
                                        !!! drukowana będzie wartość: true
is_associated = associated(ip3, liczba)

write(*, fmt = '(L4)') '' is_associated = '',  is_associated

                                        !!! drukowana będzie wartość: false
                                        !!! gdyż, co prawda, ip3 ma status
                                        !!!"associated",  ale  jest
                                        !!! aliasem do INNEGO  "target'u"
```

```
stop
end program pointery1
```

Inne przykłady deklaracji pointerów:

```
logical,  pointer  :: x1,  x3
logical,  target ::   yes,  no,  error
!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
integer,  paramater :: DP = selected_real_kind(15)
real(kind = DP),  pointer ::  p1,   p2,   work1
real(kind = DP),  target ::   xz1,  xz2,  xz3,  w
real,  pointer :: r1;  real,  target ::  re1

                                          !!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
complex,  pointer :: com1
complex,  target ::  c1,  c2,  c3
c1  = > com1;   c2 = > com1;   c3 = >c1
                                          !!! c1      = >  xz1  byłoby nielegalne,
                                          !!! bo c1  może byc aliasem jedynie
                                          !!! zmiennych typu complex (z atrybutem
                                          !!! target)

nullify(c1);                              !!! teraz  c1 nie pokazuje na nic

                                          !!! Uwaga:  nullify(c1)  nie spowoduje
                                          !!! wyzerowania !!! pointera  c3  mimo,  że
                                          !!! zarówno  c1 jak c3
                                          !!! pokazywały na com1; nadal  więc  c
                                          !!! będzie  pokazywał  na com1

x1  = > error                             !!! x1 pokazuje na zmienną error
x1  = > yes                               !!! ten sam  x1 teraz już pokazuje na
                                          !!! zmienną yes
work1  = > xz3;  p1  = > xz1; work1  = > w
                                          !!! uwaga:   r1  = > w  byloby bledem
                                          !!! bo typy real  oraz
                                          !!! real(kind = DP)  sa rozne
```

Pointery do typów pochodnych (struktur, rekordów, “derived types”):

```
type(dane)                                !!! definicja typu
 character(len = 10) :: nazwa
 real ::  alfa
 integer,  dimension(5:100) :: matr
```

```
end type dane
......
type(dane),  pointer ::  p_d
type(dane),  target ::   dane1,  dane2,  dane3
                                          !!! zmienne,  które mogą być
                                          !!! wskazywane
p_d  = >  dane3

p_d%nazwa  =  'moje dane';  p_d% alfa =  1.0
                                          !!! co  daje takie same rezultaty  co:
                                          !!! dane3%nazwa =  moje dane';
                                          !!! dane3%alfa = 1.0  !!!  itd.
                                          !!! gdyż p_d jest aliasem zmiennej dane3
```

Pointery do macierzy (nie określamy jakie będą granice tych macierzy, tylko jaki będzie ich kształt – czyli ilość indeksów, lub wymiarów)

```
real,  dimension(:),  pointer ::   p1_mat1,   p1
integer ,  dimension(:,  :),  pointer  :: p2,    p2b
integer,  dimension(:),  pointer ::  iwsk
…..
real,  dimension(100),  target ::  a100,  b
real,  dimension( 10),  target ::   a10
integer,  dimension(5,  3),  target :: i5
integer,  dimension(100),  target ::  i100
integer,  dimension(10,  10),  target :: i10x10
real,  dimension(10,  10,  10) :: duza
.............
p1_mat1  = > a100;   p1  = > a10          !!! obie instrukcje sa legalne
                                          !!! zwróćmy uwagę, że
                                          !!! p1  = > duza; byloby nielegalne
                                          !!! (zła liczba indeksow)

p2  = > i5;   p2b  = > i10x10             !!! te obie instrukcje  są legalne

                                          !!! ponadto  legalne jest aliasowanie
                                          !!! odpowiednich  segmentów macierzy:
!!!  ------------------------------------------------------------------------------
iwsk  = >  i100(5:10)                     !!! lub
iwsk  = >  i100(2:50:2)                   !!! lub nawet:
iwsk  = >  i10x10(5,  :)                  !!! wskazanie 5-go wiersza jest
                                          !!! legalne gdyż ten
                                          !!! segment jest macierzą jednowymiarową
```

Uwaga! *Nie jest dozwolone używanie pointera do segmentu macierzy o indeksach zadanych wektorem*:

```
                                          !!! integer,  dimension(3) ::  wektor = (/2, &
```

```
                                        !!! 5, 6/)
                                        !!! real,  dimension(:),  pointer :: pdm
                                        !!! real,  dimension(100),  target :: ABC
                                        !!! pdm   = >  ABC(wektor)
                                        !!! byłoby NIELEGALNE
```

Jeszcz jeden przykład wyjaśniający bliżej, jak w praktyce działa aliasowanie segmentów macierzy:

```
program aliasy_segmentow_macierzy
implicit none;
integer :: i,  j
integer,  dimension(3,  5),  target :: matr
integer,  dimension(3,  2)  ::  mala  = 100
integer,  dimension(:,  :),  pointer :: p1,  p2
                                        !!!
do i  =  1,  3; do j  =  1,  5; matr(i,  j)  =  0*i+j
end do; end do;
                                        !!! podstawiamy za matr takie liczby które
                                        !!! od  razu  pozwolą się zorientować
                                        !!! (patrzac na !!! wartość  matr(i,  j) )
                                        !!! jakie są  wartosci  i oraz j

p1  = > matr(:,  1:2);  p2 = > matr(:,  4:5)   !!! aliasowanie segmentów
mala  =  mala + p1                             !!! używanie aliasów
write(*,  fmt = '(3i5)')  mala
                                        !!! wypisane zostaną  (jedna
                                        !!! kolumna na  jeden wiersz)
                                        !!! liczby  111,  121,  131; 112,  122,  132;

write(*,  *)  p1(3,  2)                 !!!  wypisane zostanie  32
                                        !!!---------------------------------
                                        !!!  write(*,  *)  p2(3,  5)
                                        !!!  to byłby błąd,  gdyż  dla p2 obowiązuje
                                        !!! “naturalna”
                                        !!!  numeracja; tzn. p2  jest aliasem
                                        !!! “nowej”
                                        !!!  macierzy  o granicach: nowa(1:3,  1:2)

write(*,  *) matr(3,  5)                !!!  wypisane zostanie  35
write(*,  *) p2(3,  2)                  !!!  (ten sam element
                                        !!! co poprzednia instrukcja - czyli
                                        !!! matr(3,5) )
stop
end program aliasy_segmentow_macierzy
```

***Ważna reguła. Nie można deklarować macierzy pointerów* !!!**

Regułę tą można jednak obejść, gdyż macierze pointerów można "symulować" posługując się sprytnie pomocniczą definicją odpowiednich typów złożonych (derived type).

Przykładowo: chcemy mieć coś, co odpowiada kwadratowej macierzy pointerów; każdy element tej macierzy ma wskazywać z kolei na jakąś jednowymiarową macierz typu **integer**. Możemy to zrobić nastepująco, wykorzystując fakt, że pointer może wchodzić w skład typu złożonego (derived type) jako jedno z jego pól (może być też kilka takich pól).

```
type  pointer_do_wektora                    !!!  nowa deklaracja typu złożonego
integer,  dimension(:), pointer ::  p
end type  pointer_do_wektora
!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
integer,  dimension(1000),  target  ::   moje_liczby  = (/ (j,   j = 1,  1000) /)
                                            !!! deklaracja zmiennej "mp"
                                            !!! symulującej macierz pointerów:
type(pointer_do_wektora),  dimension(10,  10) ::  mp
                                            !!!  teraz już możemy się tym posługiwać,
                                            !!! np.
mp(5,  7)%p  = > moje_liczby(1:100)
mp(5,  8)%p = >  moje_liczby(101:300)
write(*,  *)  mp(5,  8)%p(177)              !!!  patrz wyłuskiwanie (dereferencing)
                                            !!!  wydrukowana zostanie wartość 177
```

II.A.3. Różnica pomiędzy zwykłą instrukcją podstawiania a instrukcją podstawiania w przypadku pointerów (assigment)

Przyjmijmy, że zadeklarowaliśmy:

```
integer,  pointer :: wsk1,  wsk2
integer,  target ::  mini,  maxi
mini =  1;  maxi  =  999
wsk1  = > mini;  wsk2  = > maxi
write(*,  *)  wsk1,  wsk2 ,  mini,  maxi
                                            !!!  wydrukowane zostaną liczby: 1 ,  999,
                                            !!! 1,  999
```

Rozważmy teraz jaki będzie wynik następujących instrukcji:

1.
```
wsk2  = > wsk1; write(*,  *) wsk1,  wsk2,  mini,  maxi
```
2.
```
wsk1 = > mini; wsk2 = > maxi;
wsk1  =  wsk2;   !!!  uwaga:  mamy tutaj znak " = "  a nie  znak "  = >"
write(*,  *) wsk1,  wsk2,  mini,  maxi
```

Otóż w przypadku 1. wydrukowane zostaną liczby: 1, 1, 1, 999; wsk1 oraz wsk2 będą aliasami do mini, a na maxi nie będzie wskazywać żaden pointer. Natomiast w przypadku 2. wydrukowane zostaną liczby: 999, 999, 999, 999; wsk1 będzie nadal aliasem do mini,

wsk2 nadal aliasem do maxi natomiast wartość mini zostanie zmieniona. Będzie się działo to samo jak przy instrukcji : mini = maxi. Powód jest następujący : przy wystąpieniu normalnego znaku " =" wsk1 oraz wsk2 traktowane są jako aliasy maxi oraz mini (wyłuskiwanie czyli *dereferencing*); faktycznie mamy więc do czynienia z instrukcją mini = maxi. Tyle tylko, że komputer będzie ją wykonywał odrobinę dłużej niż zwyczajną instukcję mini = maxi.

II.A.4. Allocate i deallocate dla pointerów

Pointery są strukturą dynamiczną. Możemy za ich pomocą dynamicznie rezerwować pamięć. Rozważmy następującą sytuację:

```
integer :: ierror
integer, pointer :: ip
real, dimension(:, :), pointer :: p_do_macierzy
allocate(ip, stat = ierror)          !!! przypominamy ze "stat = ierror mozna opuścić"
allocate(p_do_macierzy(2, 3) , stat = ierror)
```

Są to instrukcje jak najbardziej legalne mimo, że nie zdefiniowaliśmy dla pointerów odpowiednich "targetów". Co więc robią te instrukcje? *Pierwsza z nich tworzy dynamicznie NOWĄ zmienną typu integer BEZ NAZWY opatrzoną już atrybutem target i rezerwuje dla niej miejsce w pamięci. Pointer ip wskazuje na to miejsce – jest jego aliasem.* Jak można posługiwać się zmienną bez nazwy? Oczywiście może być dostępna tylko za pomocą pointera. Tak więc np. instrukcje:

```
ip = 11; ip = ip + 1
```

podstawią w to nie nazwane miejsce w pamięci liczbę 11, a następnie powiększą liczbę o jeden. Oczywiście, przydatność czegoś takiego dla zmiennej typu integer jest raczej mizerna. Natomiast w przypadku allokowania pointera do macierzy jest inaczej - jest to całkiem przydatne narzędzie.

Instrukcja

```
allocate(p_do_macierzy(10, 2:3) , stat = ierror)
```

spowoduje powstanie w pamięci nienazwanej macierzy 2 x 3 typu real, dostępnej tylko za pośrednictwem pointera. Jeżeli chcemy zainicjować jej pierwszą kolumnę jedynkami, drugą dwójkami, a trzecią trójkami, to możemy napisać instrukcje:

```
p_do_macierzy = reshape( (/1.0, 1.0, 2.0, 2.0, 3.0, 3.0/), (/2.3/) )
```

Po zainicjowaniu możemy naszej nie nazwanej macierzy używać w sposób dowolny, posługując się nie jej identyfikatorem (bo formalnie biorąc go nie ma) ale pointerem ją wskazującym, czyli aliasem "macierzy nienazwanej". Podobnie możemy używać instrukcji w rodzaju:

```
po_do_macierzy (2, 1) = po_do_macierzy(2, 1) + 1.0
```

itp.

Jeżeli już nie potrzebujemy pointerów, powinniśmy zwolnić pamięć przez nie wskazywaną prze użycie deallocate. Np.

```
...
integer, dimension(:), pointer :: pi
....
```

```
allocate(pi(100))
……
deallocate(pi)                    !!! zwolnienie pamięci przynależnej do
                                  !!! nienazwanej macierzy
nullify(pi)                       !!! uzmienie pointera
```

Jeśli spróbowalibyśmy wykonac instrukcję **nullify** *przed deallocate* to nie dałoby się zwolnić pamięci (pozostałaby ona niedostępnym "śmieciem w pamięci wisząc tam" aż do końca wykonania programu) .

Na odwrót, jeśli mielibyśmy nieco odmienną sytuację:

```
integer, dimension(:), pointer :: pi
integer, allocatable, dimension(:), target:: moje_dane
allocate(moje_dane(100))
pi = >moje_dane
```

to prawidłowa kolejność postępowania przy deallokacji byłaby następująca:

```
nullify(pi); deallocate(moje_dane);
```

Próba uczynienia na odwrót (tzn. deallocate jako pierwsze) przy pomocy samego pointera spowoduje poważny błąd:

```
Deallocate(pi)                    !!! zatrzymanie programu  blad
```

gdyż taka instrukcja usiłowałaby deallokować niezależną macierz "moje_dane".

II.A.4.1. Pointery jako składowe struktur (derived type).

W rozdziale poprzednim, mówiąc o symulowaniu macierzy pointerów, już zaznaczyliśmy, że pointery mogą być składowymi (polami) typów złożonych (*derived types*). Możliwości wykorzystania pointerów jest wiele. Podamy kolejny przykład (w oderwaniu od jakichkolwiek konkretnych zastosowań):

```
type adres
   Character(len = 25) ::  miasto,   ulica
   Integer :: nr_domu,  nr_mieszkania
end type adres
!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
type  dane_osobowe
  character(len = 25) :: imie,   nazwisko,    nr_dowodu
  integer ::  wiek
  type(adres) :: dane_adresowe
end type  dane_osobowe
!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
type   dziwne_informacje
   integer,  dimension(:),  pointer ::  moje_dlugi
  type(dane_osobowe),  pointer ::  znajomy1,  znajomy2, znajomy3
   character(len = 50) :: notatki
```

```
end type dziwne_informacje
……
```

Warto jednak podać przykład bardziej konkretny, który dobitnie przekonałby Czytelnika, że pointery w strukturach złożonych mogą być konieczne i bardzo pożyteczne. Dotyczny on typu pochodnego (derived type) z polem macierzowym. Wiemy ,że takie pole musi być macierzą o kształcie podanym w sposób jawny (explicit shape array) z granicami, które są znanymi stałymi. Czy możliwe jest wprowadzenie pola o zmiennej długości?. Takiego, które z zależności od potrzeby mogłoby mieścic kilka elementów, a w razie innej potrzeby dużo więcej? Nie możemy tego zrobić za pomocą pola-macierzy . Możemy jednak dokonać tego za pomocą pola-pointera do macierzy. Np. możemy zdefiniować typ:

```
type przykladowy                          !!! definicja typu
character(len = 35) :: nazwa
real,  dimension(:),  pointer :: wsk
end type przykladowy
!!!!!!!!!!!!!!!!!!!!
type(przykladowy) :: zmienna1,  zmienna2  !!! deklaracje zmiennych
integer :: j
                                          !!!  teraz podstawiamy za zmienna1;
                                          !!! chcemy żeby miala w sobie trzy
                                          !!! elementy:
zmienna1%nazwa  =  'zmienna nr. 1'
allocate(zmienna1%wsk(3) )
zmienna1%wsk (1)  =  1.0; zmienna1%wsk (2)  =  2.0; zmienna1%wsk (3)  =  3.0;
                                          !!! a teraz chcemy  żeby zmienna2  miała
                                          !!! 100  elementów  (wartosci - od 1
                                          !!! do 100)
allocate(zmienna2%wsk(100));
do j = 1,  100; zmienne2%wsk(j)  =  real(j); end do
```

Na razie jeszcze może wyglądać to jak zabawa; że może to być naprawdę przydatne przekonamy się rozważając (w dużym uproszczeniu) następujący modelowy problem:

Przyjmijmy, że na pliku tekstowym 'proteins.lib' mamy dane dotyczące dużej liczby białek; każde białko określone jest nazwą, liczbą aminokwasów i ich liniową sekwencją, Aminokwasów jest w przyrodzie około 20 - każdy więc zakodujemy (w naszej bazie) jako małą liczbę całkowitą (np. dwucyfrową -- od 11 do 30). Teraz z laboratorium przekazano nam dane uciętego segmentu (białka tniemy za pomocą enzymów) jakiegoś nowego badanego białka. Podejrzewamy, że jest to segment jakiegoś znanego już nam białka; ale jak sprawdzic wsród tysięcy danych o które chodzi? A może segment pasuje do kilku różnych białek? Pozostaje nam przeszukać bazę i do każdego białka przystawiać segment aby sprawdzić, czy pasuje. Oczywiście białka mają różna długość – struktura naszej bazy jest więc dosyć nieregularna. Nic nie szkodzi -- posłużymy się pointerami. Oto przykładowy program:

```
module prot_tools
implicit none;
  type protein
```

```
 character(len = 25) :: name
 integer :: aminoacids_num
 integer,  dimension(:),  pointer :: aminoacids
                                          !!!  pojedyńczy aminokwas kodowany jest
                                          !!!  przez liczbę od 11 do 30 (20
                                          !!!  aminokwasów)
 end type protein
!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
CONTAINS
 subroutine info_output(prot,  i_output)
 type(protein),  intent(in) :: prot
 integer,  intent(in) :: i_output
 integer :: i
 write(unit = i_output,  fmt = 1111)  prot%name
 write(unit = i_output,  fmt = *) prot%aminoacids_num,  &
  '   <---  aminoacids_number in protein'
 write(unit = i_output,  fmt = 2222)  prot%aminoacids
                                          !!!  wypisywanie sekwencji aminokwasow
                                          !!!  po 10  liczb w linii
 1111 format(A25)
 2222 format(10i3)
 end subroutine info_output
!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
 subroutine info_input(prot,  i_input)
 type(protein),  intent(inout) :: prot
 integer,  intent(in) :: i_input
 integer :: i,  licznik,  N
 read(unit = i_input,  fmt = 1111)  prot%name
1111  format(A25)
                                          !!! to samo może być jako
                                          !!! read(i_input,1111)
                                          !!! (tzn. bez słów kluczowych)

 read(unit = i_input,  fmt = *) N         !!! pomocnicza zmienna
                                          !!! skracająca ilość pisaniny
 prot%aminoacids_num = N
 allocate(prot%aminoacids(N))
 read(unit = i_input,  fmt = '(10i3)')  prot%aminoacids
 end subroutine info_input
!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
 logical function seeker(prot,  segment)  !!! funkcja sprawdzająca
                                          !!! czy segment pasuje do białka
 type(protein),  intent(in) :: prot,  segment
 integer :: r,  i,  N,  M,  error
 logical :: nie_sa_rowne
 integer,  dimension(segment%aminoacids_num) :: i_seg,  i_prot
                                          !!! automatyczne macierze  i_seg,
                                          !!! i_prot,    (explicite shape)
                                          !!! wprowadzone glownie do przyspieszenia
                                          !!! pracy komputera
```

```
                                        !!!
  N = prot%aminoacids_num
  M = segment%aminoacids_num
  seeker  =  .false.                    !!! zakładamy wstępnie,   że
                                        !!! wartość funkcji jest false
    if (M>N) return                     !!! koniec,    bo białko za krótkie

i_seg  =  segment%aminoacids(1:M)
!!!
i_loop: do i = 1,  N-M+1
      i_prot = prot%aminoacids(i:i+M-1)
                                        !!! przesuwamy badany fragment bialka
                                        !!! od poczatku aż do ogona,
                                        !!! a wybrany fragment porównujemy z
                                        !!! segmentem.
                                        !!! Teraz  porównywanie, czy dwie
                                        !!! macierze  NIE sa aby równe
                                        !!! (element po elemencie)

        nie_sa_rowne  =  .false.        !!! wstępne założenie,  że jednak są równe
          r_loop: do r = 1,  M
          if (i_seg(r) .ne. i_prot(r) )  then
          nie_sa_rowne  =  .true.;   exit r_loop
          end if
          end do r_loop
!!!!
      if ( .not. nie_sa_rowne) then
                                        !!! warunek spełniony tylko jeśli macierze
                                        !!! SĄ RÓWNE
      seeker = .true.
      return                            !!!   kończymy; macierze są równe
      end if
end do i_loop
end function seeker
!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
end module prot_tools
!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
program  protein_library
use prot_tools
integer ::  i,   N_max,   segment_lenght,   ierror
type(protein) ::   prot_segment,   protein_work
type(protein),  dimension(:),   allocatable :: library


!!!  wczytujemy z pliku  naszą bibliotekę białek:

open(unit = 55,   file  =  'protein.lib')
```

```
read(55,  *)  N_max                          !!! liczba skatalogowanych białek
allocate(library(N_max),  stat = ierror)     !!! przydzielamy dynamicznie
                                             !!! pamięć RAM na nasze dane

  if (ierror .ne. 0)  then; write(*,  *) ' błąd'; stop; end if

  do i = 1,  N_max                           !!! wczytujemy dane poszczególnych białek
  call info_input(protein_work,   55)
  library(i) = protein_work
  end do;  close(55)
                                             !!! wczytujemy z pliku dane segmentu;
                                             !!! będziemy (dalej) sprawdzać, czy ten
                                             !!! segment nie pojawi sie w ktorymś z
                                             !!! białek
open(55,   file = 'segment.dat')
call info_input(prot_segment,  55)
close(55)
                                             !!!  teraz przeszukujemy bibliotekę aby
                                             !!!  odpowiedzieć na pytanie,
                                             !!! które białko  ma w sobie
                                             !!!  interesujący nas segment;
                                             !!!  ewentulne trafienia wypisujemy do
                                             !!!  pliku 'trafienia.dat'
open(66,  file = 'trafienia.dat')
write(66,  *)  '  dane dla segmentu'
write(66,  *) ' ----------------------------'
call info_output(prot_segment,  66)

do i = 1,  N_max                             !!! pętla po numerach różnych białek
if  (seeker(library(i),  prot_segment)) then
write(66,  *) '   dane bialka '
call info_output(library(i),  66)
end if
end do
!!!
deallocate(library)
                                             !!! zwalniamy pamięć dynamiczną
stop
end program protein_library
```

II.A.4.2. Pointery jako składowe “derived type”– ciąg dalszy

Rozważmy następującą definicję

```
program  pointery_test
                                             !!! definicja
type   nowy_typ
 integer ::  numer
```

```
  real,   dimension(3) ::   dane
  type(nowy_typ),   pointer :: pokazuj_nastepny
end type nowy_typ
```

Jak widać, typ złożony może posiadać wewnątrz siebie pointer do (innej) zmiennej też należącej do tego samego typu! Pozwala to tworzyć powiązane listy, powiązane struktury drzewiaste itp.

Kontynuując nasz przykład możemy napisać:

```
                                                !!!   deklaracje
type(nowy_typ),   target :: pierwszy,   drugi,   ostatni
                                                !!!
pierwszy%numer = 1; pierwszy%dane  =  (/1.0,  2.0,  3.0/);
pierwszy%pokazuj_nastepny  = >  drugi
                                                !!!
drugi%numer = 3; drugi%dane  =  (/11.0,  22.0,  33.0/);
drugi%pokazuj_nastepny  = >  trzeci
                                                !!!
ostatni%numer = 4; ostatni%dane  =  (/111.0,  222.0,  333.0/);
nullify (ostatni%pokazuj_nastepny )          !!! zerujemy, bo nie ma nastepnego
!!!!

write(*,  *)    pierwszy%pokazuj_nastepny%numer
                                                !!!   wypisane zostanie 2

write(*,  *)    pierwszy%pokazuj_nastepny%pokazu_nastepny%numer
                                               !!!  wypisane zostanie z kolei   3
                                                !!! (przypominamy wyłuskiwanie, czyli
                                                !!! dereferencing)
end program pointery_test
```

Bardziej ciekawy przykład dotyczy tworzenia listy (o nieznanej z góry długości) i rozszerzania jej lub zmniejszania w miarę potrzeb (tej cechy nie posiadają macierze). Naszym zadaniem jest napisanie programu tworzącego tabelkę jakiejś szybko zmiennej funkcji f(x) w przedziale zmienności argumentu x od 0 do 1. Ma to być "gęsta tabelka" co w tym przypadku oznacza że kolejne wartości tablicowanej funkcji muszą się różnić nie więcej niż o zadany z góry epsilon (np. eps = 0.001). Postać naszej funkcji jest taka, że jej wyliczenie jest bardzo kosztowne (dużo czasu maszynowego) dlatego przyjmujemy że wykluczone jest powtarzanie obliczeń w punktach (dla tych wartości argumentu) dla których już to raz zrobiliśmy. Przyjmiemy więc następującą metodę postępowania. Na początek tworzymy listę składającą się tylko z dwóch par, pierwszej: x = 0, y = f(0), drugiej: x = 1, y = f(1). Sprawdzamy, czy f(0)-f(1) różni się więcej niż od eps; ponieważ się różnią, więc wyliczamy NOWĄ wartość funkcji w pośrodku przedziału czyli y = f(0.5) dla x = 0.5. Nową wartość wstawiamy w środek starej dwuelementowej tabelki tak aby poszczególne wartości x w nowej tabelce były uporządkowane jako rosnące. Teraz sprawdzamy tak samo jako poprzednio tym razem już 3- elementową tabelkę. Jeśli trzeba, doliczamy wartości

funkcji w punktach pośrednich (tzn . albo dla x = 0.25 i/albo dla x = 0.75). Postępujemy tak dalej. Tabelka rośnie. Zakończymy gdy okaże się, że tabelka ma już tę własność, której od niej oczekujemy.

Teraz mamy napisać program; postać funkcji opracowywuje ktoś inny,a my mamy napisać resztę programu i przetestować go. Dlatego (chwilowo) zamiast prawdziwej f(x) wstawimy jakąś prostą, zastępczą funkcję; do testowania to wystarczy.
Oto nasz nowy programik:

```
                                                !!!  tworzenie dwukierunkowej listy

program  gesta_tabelka
implicit none
real ::   eps = 0.1,   dx,   y,  x
real ::   a =  0.0,    b = 1.0
integer :: i
logical  :: sa_nowe_elementy
!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
type  element
  real ::  x,  y
  type(element),  pointer :: nalewo,  naprawo
end type element
                                                !!! będziemy tworzyc listę par (x,  y) przy
                                                !!! czym każda para
                                                !!! będzie wskazywac do pary poprzedniej
                                                !!! (o mniejszym x)
                                                !!! oraz  do pary następnej  (o większym x)
                                                !!! poszczegolna para (x,  y) oraz jej
                                                !!! wskaźniki "na prawo" i "na lewo"
                                                !!! stanowią  razem nasz element  (derived
                                                !!! type)
!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
                                                !!! teraz deklaracja zmiennych
                                                !!! pointerowych typu element
type(element),   pointer :: pierwszy,   biezacy,  &
                  nowy,  nastepny,   czwarty,  szosty
                                                !!! nie deklarujemy zmiennych typu target
                                                !!! (nie trzeba)
!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
interface
  real function f(x)
  real,  intent(in) :: x
  end function f
end interface
!!!!!!!!!!!!!
allocate(pierwszy); allocate(nastepny);
                                                !!! tworzymy 2 nienazwane zmienne  o typie
                                                !!! “element”
                                                !!!  (i o automatycznie tworzonym atrybucie
                                                !!!  target)
                                                !!!  wskazują je pointery “pierwszy”,
```

```
                                        !!! “nastepny”
                                        !!!  zmienna nr 1 wskazuje na null
                                        !!! (wyzerowany)
                                        !!! (wewnętrzny pointer "nalewo"),   oraz
                                        !!! na zmienna nr 2
                                        !!!  (wewnętrzny pointer "naprawo")
                                        !!!  wewnętrzny pointer "nalewo" zmiennej
                                        !!! nr 2 wskazuje
                                        !!!  na zmienną nr 1,   a wewnętrzny pointer
                                        !!! "naprawo" wskazuje
                                        !!!   na null; patrz instrukcje  poniżej
pierwszy%x = a; pierwszy%y = f(a)
nastepny%x = b; nastepny%y = f(b)
                                        !!!
nullify(pierwszy%nalewo)
pierwszy%naprawo  = >nastepny
nastepny%nalewo  = > pierwszy
nullify(nastepny%naprawo)
                                        !!!  w tej chwili nasza tabelka składa się
                                        !!! tylko z 2-ch elementów

sa_nowe_elementy  =  .true.
                                        !!!  zakładamy, że tabelka na początku nie
                                        !!! jest gęsta,
                                        !!!  tzn. pomiedzy (przynajmniej) dwoma jej
                                        !!! y-wymi elementami
                                        !!! jest większa różnica niż "eps" i dlatego
                                        !!! trzeba doliczyc nowe elementy
                                        !!! ulokowane  pośrodku

nowe_elementy: DO; if (.not. sa_nowe_elementy) EXIT nowe_elementy
   biezacy  = >pierwszy                 !!! rozpoczynamy od poczatka tabelki
   sa_nowe_elementy  =  .false.         !!! wstepnie zakładamy że
                                        !!! nowych elementow   już nie będzie
                                        !!! potem to zweryfikujemy

   przebieg_tabeli: do                  !!! pętla od początku do końca tabelki
    nastepny  = >biezacy%naprawo
        if (.not. associated(nastepny))  EXIT przebieg_tabeli
                                        !!! ten exit następuje dla ostatniego
                                        !!! elementu  tabelki
        if ( abs(biezacy%y-nastepny%y) > eps ) then
                                        !!! jeżeli ”y” sasiedniej pary różnią się
                                        !!! więcej niż o eps,  to doliczamy nowy
                                        !!! element ulokowany w środku pary;
                                        !!! zmieniamy stare pointery  tak, aby nowy
                                        !!! element  można było wepchnąć  do
                                        !!! środka

        sa_nowe_elementy  =  .true.
```

```
                                        !!! trzeba więc doliczyć nowy element
      allocate( nowy)                   !!! tworzymy dynamicznie nową pamięć
                                        !!! i podstawiamy
      biezacy%naprawo  = > nowy
      x =  (biezacy%x + nastepny%x)*0.5
      y = f(x) ;   nowy%x = x;nowy%y = y
      nowy%nalewo  = > biezacy
      nowy%naprawo  = > nastepny
      nastepny%nalewo  = > nowy
      biezacy = >nowy                   !!!  < =   pozycja "bieżący" została
                                        !!! przesunięta na prawo tabelki
      else
                                        !!! w tym przypadku nie dołączamy
                                        !!! nowych elementów
      biezacy  = > biezacy%naprawo      !!! ale pozycja "bieżący"
                                        !!! została przesunięta (o jedno oczko) na
                                        !!! prawo tabelki
      end if
   end do przebieg_tabeli
END DO nowe_elementy

                                       !!! Wypiszmy na ekranie “gęstą tabelkę”
biezacy  = > pierwszy                  !!! funkcji
do                                     !!! pętla po wszystkich elementach nowej
                                       !!! "gęstej" tabelki.
write(*,  fmt = '(" x,   f(x)  = ",   2F15.8)')  &
    biezacy%x,  biezacy%y
if (.not. associated(biezacy%naprawo) ) EXIT
biezacy  = >biezacy%naprawo
end do
!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
write(*,  *) ' Teraz dla treningu przećwiczmy usuwanie elementów z listy'
write(*,  *)  'przyjmimy  np.  ze chcemy usunac z naszej gestej tabelki'
write(*,  *) ' element nr.5 (mniejsza z tym po co...) '

                                       !!! odszukujemy 5-ty element
biezacy  = > pierwszy
do i = 2,  5; biezacy = >biezacy%naprawo; end do
                                       !!! obecnie "bieżący"  pokazuje na 5-ty
                                       !!! element;
                                       !!! odszukujemy 4-ty i szósty element
czwarty  = >biezacy%nalewo
szosty  = >biezacy%naprawo
                                       !!! dostosowywujemu pointery 4-go i 6-go
                                       !!! elementu listy
czwarty%naprawo  = >szosty
szosty%nalewo    = >czwarty
                                       !!! usuwamy  4-ty  (zwalnianie pamieci)

deallocate(biezacy)    !!  status pointera “biezacy” jest teraz undefined
```

```
                                        !!! jeszcze raz drukujemy tabelke (z
                                        !!! usuniętym 5-tym elementem)
                                        !!!
biezacy  = > pierwszy                   !!! wypiszemy   tabelkę funkcji
do                                      !!! pętla po wszystkich elementach
                                        !!! poprawionej "gęstej" tabelki.
write(*,  fmt = '(" x,   f(x)  = ",   2F15.8)')  &
    biezacy%x,  biezacy%y
if (.not. associated(biezacy%naprawo) ) EXIT
biezacy  = >biezacy%naprawo
end do

stop
end program  gesta_tabelka
!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
real function   f(x)
                                        !!! prosta zastępcza funkcja wstawiona dla
real,  intent(in) :: x                  !!! przeprowadzenia testu
f  =  sin(x)
end function f
```

II.A.5.1. Pointery jako argumenty subrutyn i funkcji

Pointery mogą być argumentami subrutyn i funkcji. Jak zwykle w takim przypadku obowiązuje INTERFACE (lub zamiennie - należy umieścić procedurę w module).
Ponadto pointery będące argumentami nie mogą mieć atrybutu **INTENT.**

Jako najprostszy przykład z taką subrutyną pokażemy małą modyfikację programu z poprzedniego podrozdziału (II.A.4 2).

```
module  tools
type element
   real :: x,  y
   type(element),  pointer :: nalewo,  naprawo
end type element
CONTAINS
!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
real function   f(x)
real,  intent(in) :: x
f  =  sin(x)
end function f
!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
subroutine wstaw_nowy_element(biez,  now,  nastepny)
        type(element),  pointer :: biez,  now,  nastepny
        real :: x,  y
        allocate(now)
```

```
        biez%naprawo  = > now
        x =  (biez%x + nastepny%x)*0.5
                                                        !!! uwaga: "następny" jest zmienna
                                                        !!! modułowa (globalna)
                                                        !!! więc subrutyna ma do niego dostęp
        y = f(x) ;   now%x = x;now%y = y
        now%nalewo  = > biez
        now%naprawo  = > nastepny
        nastepny%nalewo  = > now
end subroutine wstaw_nowy_element
end module
!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
program  gesta_tabelka_2
use tools
implicit none
type(element),   pointer :: pierwszy,   biezacy,  &
                    nowy,  nastepny
real ::   eps = 0.1
real,  parameter ::   a =  0.0,    b = 1.0
logical  :: sa_nowe_elementy
                                                        !!!
allocate(pierwszy); allocate(nastepny);
pierwszy%x = a; pierwszy%y = f(a)
nastepny%x = b; nastepny%y = f(b)
                                                        !!!
nullify(pierwszy%nalewo)
pierwszy%naprawo  = >nastepny
nastepny%nalewo  = > pierwszy
nullify(nastepny%naprawo)
sa_nowe_elementy  =  .true.
!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
nowe_elementy: DO; if (.not. sa_nowe_elementy) EXIT nowe_elementy
   biezacy  = >pierwszy   ! rozpoczynamy od poczatku tabelki
   sa_nowe_elementy  =  .false.
                                                        !!!
   przebieg_tabeli: do   ! petla od poczatku do konca tabelki
    nastepny  = >biezacy%naprawo
        if (.not. associated(nastepny))  EXIT przebieg_tabeli
        if ( abs(biezacy%y-nastepny%y) > eps ) then
        sa_nowe_elementy  =  .true.
        call wstaw_nowy_element(biezacy,  nowy,  nastepny)
        biezacy = >nowy                                 !!! pozycja "biezacy" zostala
                                                        !!! przesunięta na prawo tabelki
        else
        biezacy  = > biezacy%naprawo                    !!! pozycja "biezacy"
                                                        !!! została przesunięta na prawo tabelki
        end if
   end do przebieg_tabeli
```

```
END DO nowe_elementy
!!!
biezacy  = > pierwszy                    !!! wypiszemy tabelkę funkcji
do                                       !!! pętla po wszystkich elementach nowej
                                         !!! "gęstej" tabelki.
write(*,  fmt = '(" x,   f(x)  = ",   2F15.8)')  &
    biezacy%x,  biezacy%y
if (.not. associated(biezacy%naprawo) ) EXIT
biezacy  = >biezacy%naprawo
end do
!!!
stop
end program  gesta_tabelka_2
```

II.A.5.2 Pointery jako wartości funkcji

Pointery mogą być wartościami funkcji . Jak zwykle w takim przypadku obowiązuje INTERFACE (lub zamiennie - należy umieścić procedurę w module).
Ponadto pointer będący rezulatem działania funkcji musi wystąpić (w definicji funkcji) ze słowem kluczonym RESULT.

Jako prosty przykład pokażemy funkcję, która wycina z dowolnej macierzy jej k-tą kolumnę. Korzysta ona z pointerów do macierzy. Ogromna zaleta - granice wskazywanej macierzy (argumentu oraz wyniku) mogą być dowolne – w czasie definiowania funkcji zupełnie się tym “nie zajmujemy”.

```
module funkcja_pointerowa
implicit none;
integer,   dimension(:),  pointer ::  p
integer,   dimension(:,  :),   pointer :: p_macierz
CONTAINS
   function kolumna(k,  p2) result(p1)
  integer,   intent(in) :: k
   integer,   dimension(:),  pointer ::  p1
   integer,   dimension(:,  :),   pointer :: p2
   p1  = > p2(:,  k)
  end function kolumna
end module funkcja_pointerowa
!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
program testuj_f_p
use funkcja_pointerowa
implicit none
integer,   dimension(3,  3),   target ::  m111222333
integer,  dimension(3) ::  kolumna_m
m111222333  =  reshape( (/1,  1,  1,  2,  2,  2,  3,  3,  3/),   (/3,  3/) )
write(*,  1111) m111222333                !!!  wypisanie macierzy kolumnami po
                                          !!! 3 elementy
```

```
1111 format(3I5)
!!!!!!!!!!!
p_macierz  = > m111222333
kolumna_m  =  kolumna(1,  p_macierz)
write(*,  1111) kolumna_m                    !!! wypisanie kolumny nr 1

                                             !!! teraz nieco inaczej dla kolumny nr 2
                                             !!! za pomocą  drugiego  pointera; bez
                                             !!! korzystania z !!! macierzy pomocniczej
                                             !!! kolumna_m
p  = >  kolumna(2,  p_macierz)
write(*,  1111)  p
stop
end program testuj_f_p
```

III. Zamiast zakończenia

W niniejszym skrypcie nie omówiliśmy wielu problemów, a inne problemy potraktowaliśmy bardzo pobieżnie. Na wyższym poziomie zaawansowania Czytelnik będzie musiał (!) sam uzupełnić swoje wiadomości. Na razie jednak radzimy popracować, żeby nabyć biegłości chociaż na poziomie podstawowym; z pewnością zajmie to i tak bardzo dużo czasu . Wydaje się, że warto (tytułem zachęty do dalszej nauki) podać chociaż jeden dodatkowy przykład wskazujący, co opuściliśmy w trakcie naszego pierwszego spotkania z F90/95, a z czym także warto się zaznajomić. Chodzi nam o tworzenie własnych operatorów.

III.A.1. Definiowanie swoich własnych operatorów

Przez własne operatory rozumiemy to, że programista może wprowadzić krótkie synonimy na dużo dłuższe nazwy konkretnych procedur. Jako przykład weźmy problem tzw. iloczynu prostego dwóch macierzy (dla prostoty zakładamy, że są to macierze kwadratowe). Niech np. alfa i beta będą macierzami 2 x 2. Ich iloczyn prosty definiujemy jako macierz (gamma) 4 x 4 której poszczególne elementy będą równały się iloczynowi odpowiednich elementów alfa i beta, np. $gamma_{11} = alfa_{11}\, beta_{11}$ itp.

Poniżej w krótkim "przygotowawcznym" programie pokażemy jak można wyliczać iloczyn prosty w sposób standardowy. Na razie nie spotkamy w tym przykładzie nic nowego:

```
module iloczyn_prosty
CONTAINS
subroutine ip(a,  b,  c)
real,  dimension(:,  :),  intent(in) ::a,  b
real,  dimension(ubound(a,  1)*ubound(b,  1),   &
ubound(a,  1)*ubound(b,  1)),  intent(inout) :: c
integer :: i,  j,   ij,   r,  s,  rs,   N,  M
                                             !!! macierz a jest N x N;
                                             !!! macierz b jest M x M
N = ubound(a,  1);  M = ubound(b,  1)
do i = 1,  N; do j = 1,  M; ij = j + (i-1)*M
do r = 1,  N; do s = 1,  M; rs = s + (r-1)*M
```

```
c(ij,  rs) =  a(i,  r)*b(j,  s)                    !!!  < =  definicja iloczynu prostego
                                                   !!! uwaga: indeksami macierzy c
                                                   !!! są  pary  (i,  j) ,  (r,  s)  lub
                                                   !!!  możemy używać indeksów
                                                   !!! “kompozytowych: np.
                                                   !!! ij  =   j+(i-1)*M
end do; end do; end do; end do;
end subroutine ip
end module iloczyn_prosty
!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
program testuj1_iloczyn_prosty
use iloczyn_prosty
real,   dimension(2,  2) :: alfa,  beta
real,  dimension(4,  4) :: gamma
alfa = reshape( (/1.0,  1.0,  2.0,  2.0/),  (/2,  2/) )
beta = reshape( (/3.0,  3.0,  4.0,  4.0/),  (/2,  2/) )
write(*,  *)  ' wypisujemy  macierze alfa i beta po kolumnach'
write(*,  *) ' ------------------------------'
write(*,  *) ' alfa'
write(*,  1111)   alfa;   1111 format(4F12.4)
write(*,  *) ' ------------------------------'
write(*,  *) ' beta'
write(*,  1111) beta
write(*,  *) ' a teraz iloczyn prosty  alfa x beta   = '
call ip(alfa,  beta,  gamma)
write(*,  1111)   gamma
stop
end program testuj1_iloczyn_prosty
```

Zademonstrujemy teraz, co rozumiemy przez własny operator. Otóż zamiast pisać **call ip(alfa, beta, gamma)** wygodniej i krócej byłoby napisać np. **gamma = alfa .razy. beta ,** co miałoby być synonimem poprzedniej instrukcji. Otóż możemy dokonać tego następująco:

```
module iloczyn_prosty
!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
  INTERFACE OPERATOR (.razy.)
   module procedure ip
  END INTERFACE
!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
CONTAINS
function ip(a,  b) result (c)
real,  dimension(:,  :),  intent(in) ::a,  b
real,  dimension(ubound(a,  1)*ubound(b,  1),   &
ubound(a,  1)*ubound(b,  1)) :: c
integer :: i,  j,   ij,   r,  s,  rs,   N,  M
```

```
N = ubound(a,  1);  M = ubound(b,  1)
do i = 1,  N; do j = 1,  M; ij = j + (i-1)*M
do r = 1,  N; do s = 1,  M; rs = s + (r-1)*M
c(ij,  rs) =  a(i,  r)*b(j,  s)
end do; end do; end do; end do;
end function ip
end module iloczyn_prosty
                                        !!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
program testuj2_iloczyn_prosty
use iloczyn_prosty
real,   dimension(2,  2) :: alfa,  beta
real,  dimension(4,  4) :: gamma
alfa = reshape( (/1.0,  1.0,  2.0,  2.0/),  (/2,  2/) )
beta = reshape( (/3.0,  3.0,  4.0,  4.0/),  (/2,  2/) )
write(*,  *)  ' wypisujemy  macierze alfa i beta po kolumnach'
write(*,  *) ' ------------------------------'
write(*,  *) ' alfa'
write(*,  1111)   alfa;   1111 format(4F12.4)
write(*,  *) ' ------------------------------'
write(*,  *) ' beta'
write(*,  1111) beta
write(*,  *) ' a teraz iloczyn prosty  alfa x beta   = '
gamma  =  alfa .razy. beta
write(*,  1111)   gamma
stop
end program testuj2_iloczyn_prosty
```

Jeśli popatrzymy uważnie na powyższy program to widzimy, jaka jest metoda definiowania nowych operatorów.

Zupełnie analogicznie możemy rozszerzyć znaczenie starych operatorów, czyli + , -, /, * pod warunkiem, że nowe rozszerzenie nie koliduje ze starymi znaczeniami (niejednoznaczności są zabronione). Możemy także rozszerzyć znaczenie = (assigment operator) itp. Są to operatory o rozszerzonym znaczeniu, czyli tzn. “overloaded operators”.

W dodatku do definiowania własnych operatorów F90/95 pozwala definiować (w modułach) tzw. procedury generyczne, tzn. procedury o jednej nazwie które jednak działają inaczej w zależności od typu argumentów (który to typ argumentów rozpoznają automatycznie). Procedurą generyczną jest np. znana nam już funkcja standartowa “sin”. Może ona pracować prawidłowo dla argumentów typu integer oraz real (i to dla różnych parametryzacji typu “kind”). Co więcej, może jako argument przyjąć macierz – wynikiem jest wtedy wyliczenie nowej macierzy o elementach równych sinusom z poszczególnych elementów macierzy -argumentu.