Poprvé s Fortranem

Fortran je programovací jazyk vhodný pro řešení numerických problémů (Formula Translator). Je nejstarším klasickým programovacím jazykem (1957), jeho podobu udává několik norem: zastaralý Fortran 66/77, novodobý 90/95, objektový 2003/2008 a moderní 2018. Kromě obvyklých programovacích stylů, jako imperativní, strukturované a procedurální programování, poskytuje vynikající možnosti pro práci s poli, objektové programování a moderní paralelizační syntaxi (Coarray Fortran). Podobně jako pro C/C++, které je v oboru jeho konkurentem (spíše bratrem, díky jejich standardizované interoperabilitě), na výběr je několik vesměs volně dostupných překladačů (gfortran z GNU Compiler Collection, ifort a ifx z Intel oneAPI, nvfortran z Nvidia HPC SDK, flang od AMD aj.) pro Linux i Windows. Fortran a C jsou běžně k dispozici ve výpočetních centrech a velké knihovny numerických metod (NAG, IMSL, MKL, ACML, LAPACK) i paralelizační systémy (OpenMP, OpenACC, MPI) mají rozhraní pro oba jazyky. Překladače provázejí různá vývojová prostředí (Visual Studio, NetBeans, Geany, Code::Blocks), v současnosti se jeví jako prvotřídní volba Visual Studio Code. Fortranské aplikace mají typicky textové rozhraní, spouštějí se z příkazového řádku a interagují textově a pomocí textových nebo binárních souborů.

Základy

►Zdrojové soubory mohou mít přípony f90/f95/f03/f08 (s novodobým volným formátováním zdrojového textu) a f/for (s historickým pevným formátováním); překladači nejlépe akceptované přípony jsou f90 a f. Volné formátování má minimum pravidel, mj. délku řádku nejvýše 132 znaků a nedokončené řádky pokračující na dalším řádku jsou ukončeny znakem &. V klíčových slovech jazyka ani ve jménech veličin nezáleží na velikosti písmen. Na řádku může být více příkazů oddělených středníkem. Řádkový komentář se uvozuje vykřičníkem.

►Soubor a.f90 s obsahem:

```
print *,1,1.,'1'   ! příkaz pro výpis celočíselné, reálné a znakové jedničky
end                ! koncový příkaz
```

se přeloží příkazem gfortran a.f90 a přeložený (executable) soubor a.out (Linux) nebo a.exe (Windows) se spustí obvyklým způsobem: ./a.out (Linux) nebo a (Windows). Ve VS Code je vhodné volit extenze Modern Fortran (barevná syntaxe, hover nápověda, Intellisense, goto/peek aparát, linting) a Code Runner (překlad a spouštění programů).

Standardní datové typy

Standardními datovými typy se rozumí typ celočíselný (integer), reálný/floating-point (real), komplexní (complex), logický (logical) a znakový (character). Hardware počítače provádí elementární operace s frekvencí na úrovni GHz, mezi něž se řadí i obvyklé matematické operace (+, *) v několika podtypech integer a real datových typů.

►Celočíselný typ: integer, integer(kind), kde kind (podtyp) odpovídá vyhrazenému počtu bytů: 1, 2, 4, 8. Typ je vždy znaménkový. Obvyklý default je 4bytový (32bitový) a jemu příslušný rozsah hodnot je -2**31 až +2**31-1. Literály: 0, +1, -1_4 a 2147483647_4 jako výsledek funkce huge(1). Př.: integer i; integer(8) :: n=1; i=n; print *,i,n,huge(n).

►Reálný typ: real, real(kind), kde kind je opět počet vyhrazených bytů: vždy je k dispozici 4 (single precision) a 8 (double precision), někdy i 10 (hardwarová extended precision) a 16 (softwarově emulovaná quadruple precision). Pro 4/8 B je přesnost mantisy 24/53 bitů, tj. 7/15 desítkových číslic, a rozsah hodnot je přibližně ±3e38/1e308 jako výsledek huge(1._4)/huge(1._8). Literály: 1., +1., 1._4, 1e0, 1.e+0_4 (vše totéž), 1._8, 1e0_8 (obojí totéž co starobylé 1d0), 1._16. Default je 4bytový (tedy jinak než v C, Pythonu, Matlabu); vhodné je postarat se o podtyp dat ve zdrojovém kódu, lze užít i voleb překladače (gfortran -freal-4-real-8, ifort -r8 ad.). Př.: real(8) :: pi=atan(1._8)*4.

►Komplexní typ: complex, complex(kind) sdílí podtypy s typem real. Literály: (0,1), (0.,1.), (0._8,1._8), (0,1.e0). Př.: complex c,d; c=(-1,0); print *,sqrt(c); d=(1,1); print *,abs(d). Real položky komplexní proměnné: c%re, c%im.

►Logický typ: logical, logical(kind) sdílí podtypy s typem integer. Literály: .false., .true., .true._1. Př.: print *,.true.;.

►Znakový typ: character, character(len) pro 1bytové znaky a řetězce statické délky len. Literály: 'A', 1_'A', 'AB', "abc". Popis pro řetězec dynamické délky je až z novější doby, character(:),allocatable :: s; s='a'; s='abc'; print *,len(s).

►Implicitní typové konverze: datové typy veličin jsou statické, v přiřazeních se provádějí implicitní konverze. Např. při integer n; real x; complex c se v přiřazeních x=n; n=x; c=x; x=c provede konverze integer na real, real na integer (uříznutím desetinné části, ne zaokrouhlením), real na complex a complex na real (přenos reálné části). Konverze operandů na vyšší typ se provádějí automaticky i ve výrazech jako x+n (n na real), x*c (x na complex); naopak neprovádějí se konverze argumentů, takže např. sqrt(n) nelze, neboť sqrt je definováno jen pro typ real a complex. Pro explicitní konverze slouží funkce int, nint (nearest int, zaokrouhlení), real, cmplx, ichar, char aj. Př.: x=0.5; n=x; print *,n,int(x),nint(x) pro 0 0 1, x=real(1); c=cmplx(0,x); x=c; print *,x pro 0.0, print *,ichar('A'),char(66) pro 65 B.

►Popis implicit none se používá pro obvyklé vynucení explicitního popisování datových typů veličin zrušením pradávné implicitní jmenné konvence, přidělující nepopsaným jménům začínajícím na I až N typ integer a ostatním typ real. Př.: i=1; end lze přeložit, ale implicit none; i=1; end se nepřeloží s chybou „i has no IMPLICIT type.

Příkazy

Výbavu pro imperativní strukturovaný procedurální styl tvoří přiřazovací příkaz, podmíněné příkazy, cykly, skoky v cyklech, volání podprogramu a návratový příkaz; k základům patří také příkazy vstupu a výstupu.

►(=, =>) Přiřazení variable=expression lze použít pro skaláry, pole a sekce polí a struktury. Př. integer n,ia(2); n=0; ia=[1,2]. K přiřazení pro ukazatele pointer=>target a možnosti redefinice přiřazení pro struktury se dostaneme níže.

►(if) Podmíněný příkaz s logickou podmínkou má plnou podobu if (logExpr1) then; cmdsTrue1; elseif (logExpr2) then; cmdsTrue2; …; else; cmdsFalse; endif. Při splnění logické podmínky logExpr1 se vykonají příkazy cmdsTrue1, jinak se testuje další podmínka atd.; není-li splněna žádná z podmínek logExpr, vykonají se příkazy cmdsFalse. Větve elseif i else jsou nepovinné. V nejkratší podobě má příkaz jen pozitivní větev s jediným příkazem, if (logExpr) cmdTrue. Podmíněný příkaz s přiřazením do téže proměnné v obou větcích, zde pro výpočet absolutní hodnoty, if (x>0) then; absx=x; else; absx=-x; endif; lze zapsat pomocí přiřazení s podmíněným výrazem v podobě funkce merge, absx=merge(x,-x,x>0).

►(select case) Podmíněný příkaz s přepínačem je ve tvaru select case (intExpr); case (list1); cmds1; …; case default; cmdsDefault; end select. K provedení se vybere větev, která je uvozena seznamem list, v němž se nachází aktuální hodnota přepínače (často integer výrazu) intExpr; není-li aktuální hodnota přepínače v žádném seznamu, vybere se větev case default. Seznamy mohou mít tvar výčtu a/nebo rozsahu, case (1,2,4:5). Př. Výpočet znaménka n do z: select case (n); case (:-1); z=-1; case (0); z=0; case (1:); z=1; end select.

►(do while, cycle/exit) Cyklus s prováděcí podmínkou má syntaxi do while (logExprRun); cmds; enddo. Příkazy cmds se opakovaně provádějí tak dlouho, dokud je splněn logický výraz logExprRun. Vykonávání příkazů cmds lze předčasně ukončit příkazem cycle pro skok zpět na vyhodnocení logExprRun nebo příkazem exit pro skok za enddo. Je-li vhodné místo pro podmínku jinde než na začátku, lze užít nekonečný cyklus do; cmds; enddo s příkazem if a skokem cycle nebo exit; např. cyklus s ukončovací podmínkou na konci je do; cmds; if (logExprBreak) exit; enddo. Cykly mohou být pojmenované, loop1: do; enddo loop1, což se může hodit při specifikaci cíle skoku ve vnořených cyklech: exit loop1. Př. Výpočet počítačového epsilon, tedy nejmenší kladné reálné číslo eps, pro které je 1+eps>1: eps=1; do while (1+eps*0.5>1); eps=eps*0.5; enddo; print *,eps,epsilon(1.). Na výpisu bude hledané číslo pro default real získané jednak výpočtem, jednak voláním fortranské funkce.

►(sekvenční do) Sekvenční indexovaný cyklus se píše jako do n=first,last,step; cmds; enddo a provádí opakovaně příkazy cmds pro integer (s výstrahou i real) index n počínaje hodnotou first, konče hodnotou last včetně a nenulovým krokem step. Vykonávání příkazů cmds lze předčasně ukončit skokem cycle na další iteraci nebo skokem exit za enddo. Př. Sumace prvků aritmetické posloupnosti: nmax=10; s=0; do n=1,nmax; s=s+n; enddo; print *,nmax,s.

►(do concurrent) Paralelizovatelný indexovaný cyklus do concurrent (n=first:last:step); cmds; enddo provádí opakovaně příkazy cmds pro hodnoty integer indexu n z aritmetické posloupnosti od first do last s krokem step, ovšem v nedefinovaném pořadí. Překladače mohou na výzvu cyklus vláknově paralelizovat, tedy vytvořit vhodný počet vláken a každému přidělit část cyklu k paralelnímu provedení. Př. Sekvenční a paralelizovaný cyklus s Intel a Nvidia překladači (gfortran je pozadu):

```
do n=1,16; print *,n; enddo;   do concurrent (n=1:16); print *,n; enddo
```

Překlad a paralelizovaný běh s 8 vlákny (lze pozorovat pomíchané výpisy) v Linuxu:

```
ifort -qopenmp a.f90; OMP_NUM_THREADS=8 ./a.out          # počet vláken se nastavuje jako v OpenMP
nvfortran -stdpar=multicore a.f90; ACC_NUM_CORES=8 ./a.out                    # zde jako v OpenACC
```

a ve Windows: ifort -Qopenmp a.f90 && set OMP_NUM_THREADS=8 && a.

Klauzule cyklu shared, local a local_init přispívají k upřesnění lokality proměnných ve vláknech. Výsledek redukčních operací, zde sumace do s, je dosud nejistý (nvfortran je umí, ifort je kazí):

```
integer,allocatable :: a(:); nmax=16; a=[(n,n=1,nmax)]
do concurrent (n=1:nmax) shared (a) local (m); m=n; print *,m,a(n); enddo
s=0; do concurrent (n=1:nmax); s=s+n; enddo; print *,nmax,s ! nvfortran ok, ifort 2021.7: chybné s
```

Tradiční cestou k vláknové paralelizaci jsou OpenMP direktivy aplikované na sekvenční indexovaný cyklus.

►(call/return, stop/pause) Podprogramy se volají příkazem call subroutine(arguments), návrat na místo volání se provede po dosažení konce podprogramu nebo příkazem return. Program se ukončí po dosažení jeho konce nebo příkazem stop, kterým lze nastavit výstupní zprávu, stop 'Hotovo.', nebo chybový kód, stop 2, testovatelný v operačním systému (Windows: if errorlevel 2 echo Problem 2, Linux: (($?==2)) && echo Problem 2). Příkaz pause může program pozastavit; z normy jazyka byl sice vyškrtnut, ale překladače ho nadále tolerují.

►(I/O) Pro výpis na obrazovku (standardní výstup, stdout) se používají příkazy print *,list nebo write (*,*) list. Pro načtení dat z klávesnice (standardní vstup, stdin) jsou určeny příkazy read *,list nebo read (*,*) list. Hvězdička v kratších a druhá hvězdička v delších variantách dává překladači volnost k formátování, nahradit ji lze řetězcem s formátovou specifikací, např. print '(i0,x,f0.2,x,a)',1,1.,'1' pro výpis 1 1.00 1 (podrobněji později). První hvězdička v delších variantách může být nahrazena celočíselným deskriptorem souboru. Tím může být 0 pro standardní chybový výstup (stderr): write (0,*) 'Problem', nebo kladné číslo přiřazené programátorem v příkazu open pro otevření souboru: id=1; open (id,file='filename'); write (id,*) x; close (id), nebo záporné číslo přiřazené automaticky pomocí klauzule newunit: open (newunit=id,file='filename'). Uvedené dvě varianty příkazu open otevřely textový

(formátovaný) soubor pro čtení nebo zápis, příkaz close soubor uzavřel (ukončil zápis na disk). Binární soubor (neformátovaný proud bytů) se otevře příkazem open (newunit=id,file='filename', form='unformatted', access='stream', status='replace') a zapisuje se do něj bez uvedení formátové specifikace: write (id) x; close (id). Pro čtení je běžné užívání hvězdičky místo formátových specifikací, sloupce dat oddělené běžnými oddělovači se načítají bez potíží. Soubor s obsahem 1 1.234 "1234 5678" lze načíst sekvencí integer n; real(8) x; character(10) s; open (newunit=id,file='a.dat'); read (id,*) n,x,s; close (id).
►(etc) O příkazech forall a where pro práci s poli a o alokačních příkazech allocate/deallocate se zmíníme později. Příkazy skoku na návěští v různých variantách (goto, if) jsou zastaralé, prázdný příkaz continue také ztratil na významu. Příkaz include 'filename' se ve zdrojovém souboru nahradí obsahem udaného souboru; např. při dvou zdrojových souborech module.f90 a main.f90, překládaných příkazem gfortran -o main module.f90 main.f90, lze po přidání řádku include 'module.f90' na začátek main.f90 překládat jen pomocí gfortran -o main main.f90.

Popisy

Popisy (deklarace) mají v novodobém Fortranu podobu popisů datového typu s volitelnými atributy. Standardní datové typy byly shrnuty výše, tvoření odvozených datových typů následuje zde. Pomocí atributů se popisují konstanty, pole, ukazatele, formální argumenty procedur aj. Cestu k objektovému programování otevírají odvozené datové typy s vázanými (vnořenými) procedurami. Popisy se uvádějí zkraje programových jednotek, před příkazy, a jejich platnost zahrnuje programovou jednotku; v omezené míře se popisy mohou vyskytnout i jako součást některých konstrukcí, především konstrukce block.
►Popisy typu v minimalistické podobě se skládají ze jména typu, případně s podtypem (kind) a pro řetězce s délkou (len) v závorkách, a seznamu jmen, tedy: type(kind) name,... Např.:

```
integer i,j; real(8) x,y; complex(8) c; logical(1) L; character ch; character(10) s
```

pro popis proměnných v typech default (4bytový) integer, 8bytový real (též double precision), (8+8)bytový complex, 1bytový logical (default je 4bytový) a 1- a 10-znakový character (o 1bytových znacích).
►Popisy typu obsahující atributy nebo inicializace mají syntaxi: type(kind),attributes :: name=value,..., pro řetězce: character(len),attributes :: name=value,... Atributy lze jménům přiřazovat i pomocí samostatných popisů.
►Atribut parameter slouží k popisu a inicializaci symbolických konstant. Symbolické konstanty pak lze hned použít např. na místě podtypů a pro řetězce jejich délek v následných popisech:

```
integer,parameter :: wp=8,len=10        ! working precision pro real a complex data, statická délka pro řetězce
real(wp),parameter :: pi=atan(1._wp)*4.         ! symbolická reálná konstanta
complex(wp) :: c1,c2=(0,1)                      ! komplexní proměnné, druhá z nich inicializovaná
character(len) :: s='ABC'                       ! řetězec statické délky
```

Pro zjištění vhodného podtypu integer a real typů pro udržení až p platných desítkových číslic a desítkového exponentu až r lze užít funkce selected_int_kind(r) a selected_real_kind(p,r), např. integer,parameter :: ip= selected_int_kind(9), rp=selected_real_kind(15,300) pro hodnoty 4 a 8, použitelné v popisech integer(ip) a real(rp).
►Ukazatele se popisují atributem pointer, jejich cíle (nejsou-li samy ukazatelem) povinným atributem target. Ukazatel získá adresu cíle alokací (příkaz allocate), přiřazením jiného ukazatele nebo přiřazením cíle deklarovaného s atributem target (přiřazení pro ukazatele: =>). Přiřazením pomocí = se nemění adresa v ukazateli, ale ukládá se nová hodnota do cíle ukazatele. Dereference neboli přístup k cíli ukazatele je implicitní, pouhým jménem ukazatele:

```
real,pointer :: p,r,s; real,target :: t=1               ! popis tří ukazatelů a jedné proměnné jako možného cíle
allocate (p); p=1; r=>p; s=>t; print *,p,r,s,t          ! alokace cíle s inicializací, přiřazení cílů pro r a s, výpis 4x 1.0
p=2; s=2; print *,p,r,s,t                               ! update obou cílů (p&r, s&t), výpis 4x 2.0
```

►Pole se deklarují atributem dimension s popisem mezí nebo popisem mezí u svého jména. Statická pole musí mít jako meze uvedeny literály nebo výše definované symbolické konstanty a může jim být popisem přidělena počáteční hodnota. Alokovatelná, resp. ukazatelová pole mají atribut allocatable, resp. pointer, a meze se v popisu neuvádějí, jen dvojtečky naznačují počet dimenzí. Alokovatelná a ukazatelová pole pak mohou získat paměťové místo příkazem allocate nebo přiřazením (alokovatelná pole) cílové hodnoty pomocí =, resp. (ukazatelová pole) cílové adresy pomocí =>. Cíl ukazatelového pole musí být ukazatelem nebo mít atribut target. Pole mohou mít libovolnou dolní mez, implicitně je 1. Statická i alokovatelná pole jsou v paměti uložena souvisle; matice se v paměti ukládají po sloupcích.

```
integer,dimension(3) :: v=[1,2,3],m(2,2)=2      ! statický vektor inicializovaný konstruktorem pole, matice
real,allocatable,target :: a(:,:)               ! alokovatelná matice, možný cíl ukazatelového pole
real,pointer :: p(:,:)                          ! ukazatelové pole
allocate (a(-1:1,0:2)); p=>a                    ! alokace pole, nastavení cíle pole p
allocate (p(2,2)); p=m                          ! realokace cíle pole p a přiřazení hodnot z matice m
a=m                                             ! automatická realokace pole a a přiřazení hodnot z matice m
```

►Odvozený typ sdružuje položky standardních a dříve definovaných odvozených typů. Definuje se deklarační konstrukcí type tName; declaration_of_components; end type, kde deklarace položek mají tvar popisů typů jako výše, např. type tBod; integer n; real M,x(3); end type shrnující integer číslo a real hmotnost a polohový vektor hmotného bodu. Proměnné odvozených typů, tzv. struktury, se vytvářejí popisem type(tName),attributes :: name,... a inicializují

se přiřazením konstruktoru struktury tName(…) nebo jiné struktury téhož typu. Vypisovat struktury lze po položkách nebo s jistými omezeními celé. Přístup k jednotlivým položkám struktury umožňuje selektor %:

```
type tBod; integer n; real M,x(3); end type        ! odvozený typ s položkami n, M, x(1:3)
type(tBod) :: bod=tBod(1,1.,[1.,2.,3.]),pole(10) ! popis struktury a pole struktur, inicializace konstruktorem
pole=bod                                           ! inicializace pole struktur přiřazením skalární struktury
print *,bod,pole(1)                                ! výpis skalární struktury a prvního prvku pole struktur
print *,bod%n,bod%M,bod%x,pole(1)%n,pole(1)%M,pole(1)%x(1:3)     ! výpis téhož po položkách
```

Vytvoříme-li strukturu trojbod typu type tTrojbod; type(tBod) A,B,C; end type, k jejím položkám můžeme přistupovat pomocí např. trojbod%A%n, trojbod%B%M, trojbod%C%x(3).

►Později rozebereme atributy intent, optional a value formálních argumentů procedur, save lokálních proměnných procedur, public, private a protected modulových proměnných a další atributy související s objektovým programováním a interoperabilitou s C.

►Výše stojí, že popisy se uvádějí zkraje programových jednotek a jejich platnost zahrnuje programovou jednotku. Moderní Fortran přinesl několik výjimek: konstrukci block/end block pro zapouzdření popisů a zapouzdření indexů v konstruktorech polí a cyklech do concurrent a forall:

```
integer :: n=0,a(3)              ! inicializace n=0
a=[(n,n=1,3)]; print *,n         ! index n zapouzdřen v konstruktoru pole, na výpisu původní 0
do concurrent (n=1:3); a(n)=n; enddo; print *,n          ! index n zapouzdřen v do concurrent, na výpisu 0
block; integer :: n=1; print *,n; end block       ! n zapozdřeno v konstrukci block, na výpisu jeho hodnota 1
print *,n                        ! výpis původního n, tedy 0
```

►Ve Fortranu 77 frekventované, dnes však beze zbytku nahrazené a proto zastaralé jsou popisy common společných bloků a equivalence pro překrývání dat v paměti. Př. Přenos dat mezi jednotkami, překryv proměnných v paměti:

```
subroutine s; common /cb/ a,b; a=1.; b=2.; end subroutine  ! podprogram bez rozhraní sdílí data v bloku /cb/
program p; common /cb/ a,b; equivalence (c,d); call s; c=3.; print *,a,b,c,d; end program          ! 1. 2. 3. 3.
```

Dnes:

```
module m; real a,b; end module                              ! modulové proměnné sdílitelné mezi jednotkami
subroutine s; use m; a=1.; b=2.; end subroutine             ! podprogram bez rozhraní sdílí modulová data
program p; use m; real,target :: c; real,pointer :: d=>c; call s; c=3.; print *,a,b,c,d; end program ! 1. 2. 3. 3.
```

Výrazy a operátory

Výrazy zahrnují běžné aritmetické, relační a logické operace a řetězení znakových výrazů. Lze definovat vlastní operátory spřažením s funkcemi o jednom či dvou argumentech standardního nebo odvozeného typu. Výrazy se vyčíslují v pořadí podle daných priorit operací, dílčí výrazy v závorkách se vyhodnocují přednostně. Operandy mohou být skaláry i pole; operace s poli se vyhodnocují prvek po prvku.

►Aritmetické operátory jsou aditivní +, − (nejnižší priorita), multiplikativní *, / (vyšší priorita) a umocnění ** (nejvyšší priorita). Operace o stejné prioritě se provádějí zleva doprava s výjimkou umocnění, které naopak: 1+2*3 je 7, 2**1**2 jsou 2, 1e30*1e30/1e30 je Inf, zatímco 1e30*(1e30/1e30) je 1e30. Operátor dělení / je kontextový, s dvěma integer operandy provádí celočíselné dělení (vrací celou část podílu, zbytek lze zjistit funkcí mod), jinak reálné dělení: 1/2 je 0, mod(1,2) je 1, 1./2. stejně jako 1./2 a 1./2+1/2 vrátí 0.5; 1./2+1./2 vrací 1.0. Mocnina 2**(-1) je totéž co 1/2**1, tedy 0, na rozdíl od 2.**(-1). Operace s poli: [1,2]+[3,4] vrátí [4,6] a [(n,n=1,10)]**2 vektor kvadrátů.

►Pro typ character je definován operátor řetězení //: 'a'//'bc' pro 'abc'. S řetězci statické délky bývá vhodné řezat koncové mezery: trim(s1)//trim(s2). Pro předběžnou ukázku definice operátoru + pro řetězce (o modulech více níže):

```
module mPlus; interface operator (+); module procedure plus; end interface; contains
function plus(a,b); character(*),intent(in) :: a,b; character(:),allocatable :: plus; plus=trim(a)//trim(b)
end function; end module
use mPlus; print *,'a'+'bc'; end
```

►Relační operátory vyhodnocují relace mezi aritmetickými a znakovými výrazy: rovnost ==, nerovnost /= a porovnání <, <=, >, >=. Existují i jejich starší ekvivalenty: .eq., .ne., .lt., .le., .gt., .ge.. Návratová hodnota je logického typu (T=.true., F=.false.). Ukázky: 1+1==2, .1+.1==.2, .1+.1+.1==.3 pro 3krát T, ale .1_8+.1_8+.1_8==.3_8 pro F. Poslední relace dokládá rizikovost testování na přesnou rovnost v (nepřesné) reálné aritmetice; vhodnější je předepsat požadovanou přesnost eps a testovat dosažení abs(x-y)<eps. Relace na polích vytvářejí pole logických hodnot, tzv. masky: mod([(n,n=1,10)],2)==0 vrátí logické pole s T v sudých prvcích.

►Logické operátory skládají operandy logického typu (obvykle relace) podle pravidel Booleovy algebry. K dispozici je unární operátor .not. (nejvyšší priorita) a binární operátory konjunkce (průnik, logické násobení) .and. (nižší priorita), disjunkce (sjednocení, logický součet) .or. (nižší priorita) a ekvivalence .eqv., .neqv. (nejnižší priorita). Ve složených výrazech se nejprve vyhodnocují aritmetické, resp. znakové dílčí výrazy, pak relace a nakonec logické výrazy. Příklad složené zastavovací podmínky: abs(x-xfinal)<eps .or. n>nmax. Tyto a další operace na bitech integer výrazů konají funkce not, iand, ior aj.: not(0), iand(1,3), ior(1,3) pro -1, 1 a 3.

►Definované operátory mohou přetížit (dodefinovat, nikoliv předefinovat) standardní operátory pro dosud nevyužité kombinace operandů nebo získat nové jméno v podobě .name. (písmena obklopená tečkami), např. .plus., .sum..

Struktura programu

Fortranský program se skládá z programových jednotek: jednoho hlavního programu (program) a libovolného počtu procedur (podprogramy subroutine a funkce function). Jednotky mohou být řazeny sekvenčně, bez dalšího strukturování (vnější procedury), mocnější je však procedury vnořit (contains) do hlavního programu nebo ještě lépe sdružovat do modulů (module) a moduly podle potřeby k hlavnímu programu, procedurám nebo jiným modulům připojovat (use). Každá programová jednotka končí popisem end. Zrušení implicitní jmenné konvence (implicit none) platí v celé jednotce, včetně vnořených jednotek.

►Hlavní program s vnořeným podprogramem a funkcí a jeden vnější podprogram:

```
subroutine s(); print *,'external s'; end subroutine        ! vnější (external) podprogram
program p; implicit none; call s(); print *,f(1.)           ! program upřednostní vnitřní podprogram
contains
   subroutine s(); print *,'internal s'; end subroutine     ! vnitřní (internal) podprogram
   function f(x); real f,x; f=x; end function
end program
```

Vnořené procedury sdílejí data svého hostitele. Lokální deklarací ve vnořené proceduře se stejnojmenná data hostitele zakryjí: i=1; j=1; call s(); print *,i,j; contains; subroutine s(); integer i; i=2; j=2; end subroutine; end vypíše 1 2.

►Modul s vnořenými procedurami a hlavní program připojující modul:

```
module mProcedures; implicit none                           ! implicit none platí v celém modulu
contains                                                    ! modulové procedury
   subroutine s(); print *,'module procedure s'; end subroutine
   function f(x); real f,x; f=x; end function
end module
program p; use mProcedures; implicit none; call s(); print *,f(1.); end program
```

Modul může obsahovat deklarace modulových proměnných, konstant, odvozených typů aj. Data a procedury v modulech jsou implicitně veřejná (public), vnější přístup k nim lze omezit na čtení (protected) nebo zcela potlačit (private). Data zpřístupňovaná připojením modulu lze vybírat vyjmenováním (only) a případně přejmenovat (=>):

```
module m1; real,parameter :: e=exp(1.),pi=atan(1.)*4; end module                ! vše implicitně public
module m2; private; public pi; real,parameter :: pi=atan(1.)*4,phi=(1+sqrt(5.))/2; end module   ! phi private
use m1, only : e,pi1=>pi; use m2, pi2=>pi; print *,e,pi1,pi2,phi; end          ! neinicializované phi
```

Moduly často obsahují také bloky rozhraní (interface), pomocí nichž lze popisovat rozhraní vnějších procedur, přetěžovat procedury a definovat operátory a přiřazení (více níže).

Vnější procedury

Procedury (podprogramy a funkce) člení program do menších úseků, které plní čitelný cíl, mohou obsahovat z vnějšku nedostupná lokální data a mohou být samostatně překládány. Je definováno jejich rozhraní pomocí formálních argumentů, kterým jsou při volání procedury přiřazeny (odkazem nebo hodnotou) skutečné argumenty; procedury mohou též sdílet s okolím (globální) modulová data. Fortran neprovádí typové konverze argumentů, o to potřebnější je dbát o explicitní zpřístupnění rozhraní procedur volajícím jednotkám. Procedury mohou být rekurzivní, přetížené, čisté a prvkové.

Volání: volání funkce (výraz): name(actual_arguments), př. sqrt(2.), sqrt(2._8)
volání podprogramu (příkaz): CALL name(actual_arguments), př. call random_number(x)

Specifikace: funkce: FUNCTION name(dummy_arguments) s volitelným RESULT (resultname)
a specifikace typu pro name, resp. resultname
podprogram: SUBROUTINE name(dummy_arguments)

Přiřazení mezi skutečnými a formálními argumenty je poziční nebo pomocí formálních jmen,
př. CMPLX(x[,y][,kind]): print *,cmplx(1.,0.,8),cmplx(x=1.,kind=8).

Předávání argumentů: skutečné argumenty jsou předávány do procedury přednostně odkazem (předáním adresy), nelze-li, pak hodnotou (kopírováním hodnoty do lokálního paměťového místa). Odkazem lze předat jen veličinu s adresou, tj. proměnnou, prvek pole, položku struktury. Konverze typu a podtypu se neprovádějí v žádném případě, takže např. sqrt(1) nelze ani přeložit a volání f(1) pro function f(x); real f,x; f=sqrt(x); end; vrátí nesprávnou hodnotu.

Rozhraní implicitní a explicitní: shoda formálních a skutečných argumentů při volání samostatných jednotek se nekontroluje (překladač ji odvozuje implicitně). Vhodnější a často nezbytné je poskytnutí explicitního rozhraní volající jednotce, a to pomocí INTERFACE bloků nebo vnořením procedury do připojeného modulu nebo volající jednotky.

Pole předpokládaného tvaru: formální pole lze specifikovat bez uvedených mezí, které jsou při volání převzaty od skutečných argumentů a zpřístupněny funkcemi SHAPE, SIZE, LBOUND a UBOUND; př. REAL a(:),b(0:,0:).

Sdílení dat: sdílení globálních dat lze dosáhnout připojením modulu s takovými daty ve volající jednotce i volané proceduře. Vnořené proceduře jsou dostupná data jejího hostitele. Lokální data procedury jsou vně procedury nedostupná.

Rekurzivní procedury: procedury odvolávající se na sebe, buď přímo nebo nepřímo (prostřednictvím jiné procedury). Popisují se klíčovým slovem RECURSIVE, např. recursive function name(dummies) result (resultname). Rekurzivní funkce se volají pomocí name a hodnotu vracejí pomocí resultname.
Přetížené, čisté a prvkové podprogramy: přetížení je zavedení generického jména pro specifické procedury rozlišené různým počtem, typem a podtypem argumentů, obvykle pomocí INTERFACE bloku pro modulové procedury. Čisté procedury nemají vedlejší efekty (nemění hodnoty argumentů, nevypisují apod.). Prvkové procedury se definují pro skalární argumenty a volány mohou být s poli jako skutečnými argumenty.

Vstup a výstup

Příkazy pro standardní V/V:	READ fmt,list ; READ (*,fmt,attribs) list; PRINT fmt,list ; WRITE (*,fmt,attribs) list	
formátování V/V:	atribut FMT=* nebo řetězec s formátovou specifikací	
odřádkování:	atribut ADVANCE='YES' | 'NO' (default 'YES')	
ošetření V/V chyb:	atribut IOSTAT=ierr (ierr=0 bez chyby, <0 end-of-file, >0 jiná chyba)	
Formátové soubory:	OPEN (id,FILE='filename') ; READ (id,form) ; WRITE (id,form), CLOSE (id)	
Formátová specifikace:		
datové deskriptory	pro INTEGER	Iw, Iw.m, a BOZ ekvivalenty pro 2, 8, 16kovou soustavu
	pro REAL a COMPLEX	Fw.d, Ew.d, Ew.dEe, ENw.d, ESw.d
	pro LOGICAL	Lw
	pro CHARACTER	A, Aw
	obecně	Gw.d, Gw.dEe
	jsou opakovatelné	př. nIw
	Iw, Fw.d připouštějí nulové w pro výstup v minimální potřebné šířce	
řídicí deskriptory	nX pro mezery, / nebo n/ pro odřádkování, nestd. $ pro ADVANCE='NO', aj.	
vnořování	n(…), př. (2147483647(2147483647E10.3))	

Různé

měření doby běhu	real tic,toc; call cpu_time(tic); …; call cpu_time(toc); print *,toc-tic	! čas v sekundách
argumenty programu	integer :: number=1; character(80) value	
	if (command_argument_count()>0) call get_command_argument(number,value)	
proměnná prostředí	character(80) :: name='username',value	! Windows: 'username', Linux: 'USER'
	call get_environment_variable(name,value)	
vnější příkaz	character(80) :: command='cmd'	! Windows: 'cmd', Linux: 'bash'
	call execute_command_line(command)	

Překladače

gfortran — GNU překladač, podpora OpenMP, MPI a GPU (direktivy OpenACC, OpenMP pro GPU)
ifort a ifx — Intel překladače (Classic a LLVM) z oneAPI Toolkits, knihovna MKL, podpora OpenMP a MPI
nvfortran — Nvidia překladač z HPC SDK, podpora OpenMP, MPI a GPU (OpenACC, OpenMP, CUDA Fortran)
flang — fortranský front-end pro LLVM back-end, součást AMD AOCC

Vybrané volby překladačů:

	gfortran	ifort a ifx	nvfortran	flang
nápověda:	man gfortran gcc	-help, man ifort	-help, man	
výstupní soubor:	-o outfile (default a.out)	-o outfile	-o outfile	
min. optimalizace:	-O0 (default)	-O0	-O0, -O1 (default)	
běžná optimalizace:	-O2	-O2 (default)	-O2	
max. optimalizace:	-Ofast -march=native	-Ofast -xHost, -fast	-fast, -O4	
jen překlad:	-c	-c	-c	
pro debugger:	-g, -ggdb	-g, -debug, -traceback	-g, -traceback	
OpenMP:	-fopenmp	-qopenmp	-mp	
OpenACC:	-fopenacc		-acc	
BLAS a LAPACK:	-lblas, -llapack	-mkl	-lblas, -llapack	
coarrays:	caf/cafrun,-fcoarray=single	-coarray, -coarray=single		
default real přesnost:	-fdefault-real-8 (10, 16)	-r8, -r16	-Mr8, -r8	
zachycení real výjimek:	-ffpe-trap=overflow,invalid,zero	-fpe0, -init=snan	-Ktrap=ovf,inv,divz|fp	
šetření zásobníku:	-fno-automatic	-heap-arrays, -save	-Mnostack_arrays, -Msave	
kontrola chyb za běhu:	-fcheck=all	-C, -check all		
kontrola mezí polí:	-fcheck=bounds	-CB, -check bounds	-C, -Mbounds	
kontrola neinic. prom.:	-Wuninitialized	-CU, -check uninit		
méně/více informací:	-w/-Wall, -pedantic	-w/-warn	-w/-Minform, -Minfo	
shoda s normou:	-std=f95|f2003|f2008	-stand f95|f03|f08	-Mstandard	

Knihovny numerických metod

NAG, IMSL — velké knihovny numerických metod
MKL, ACML — Intel Math Kernel Library, AMD Core Math Library, knihovny šířené s překladači (LA, FFT aj.)
LAPACK, BLAS — Linear Algebra Package, Basic LA Subprograms, volně šiřitelné, součást všech předchozích

Odkazy

Metcalf M., Reid J., Cohen M., Modern Fortran Explained, Incorporating Fortran 2018, Oxford Science, 2018
Brainerd W. S., Guide to Fortran 2008 Programming, Springer, 2015
Chapman S. J., Fortran for Scientists and Engineers, 4th Ed., McGraw-Hill, 2018
Chivers I., Sleightholme J., Introduction to Programming with Fortran, 4th Ed., Springer, 2018
Clerman N. S., Spector W., Modern Fortran, Style and Usage, Cambridge, 2012
Markus A., Modern Fortran in Practice, Cambridge, 2012
Ray S., Fortran 2018 with Parallel Programming, CRC Press, 2020
Fortranské normy: gcc.gnu.org/wiki/GFortranStandards
Dokumentace k překladačům: GNU gcc.gnu.org/onlinedocs, Nvidia docs.nvidia.com/hpc-sdk,
Intel www.intel.com/content/www/us/en/develop/documentation/fortran-compiler-oneapi-dev-guide-and-reference

L. H., 15. 11. 2011, částečná aktualizace 18. 11. 2022