Kurs FORTRANU

Filip Hroch

Tento dokument je psaná verze přednášek o fortranu konaná v podzimním semestru akademického roku 2001-2002.

Základní prvky jazyka

Zápis čísel a textu

- Celá čísla se zapisují prostě jako 1,2,3,100,-25.
- Reálná čísla mají desetinou tečku a exponent: 1.0, 0.55, 1e6, 3.3e-66.
- Znakové informace se uzavírají mezi apostrofy nebo uvozovky: "ahoj",'ahoj'

Proměnné

Proměnné se zapisují jakýmkoli identifikátorem zacínajícím znakem jiným než je číslo. Například `X,Y,PIXEL,XY12`.

Aritmetické výrazy

Pro elementární operace sčítání, odčítaní, násobení, dělení a umocňování se používají operátory `+ - * / **`.

Funkce

Fotran obsahuje řadu matematických i jiných funkcí. Používají se následovně `sqrt(1.0 + d)`, `sin(x)`, `exp(1.0)`.

Struktura programu

Každý program ve Fortranu začíná příkazem `Program` a končí příkazem `end`. Mezi těmito slovy jsou popisy a příkazy. Kazdý popis nebo příkaz se píší na samostatný řádek. Mezery jsou nevýznamé. Vše následující za znakem `!` je komentář.Zde je přehled základních prvků:

- Definice typu proměnné pro celočíselné proměnné `integer :: i`
- Definice typu proměnné pro reálné proměnné `real :: x`
- Definice typu proměnné pro znakové proměnné `character :: s`
- Příkaz na vypsání obsahu proměnné `write(*,*) x`

Jednoduchý program

S těmito znalostmi už můžeme napsat jednoduchý program. Jakou řeší úlohu?

```
program p1
! prvni program
real :: a,b,c,d,e,x1,x2

a = 1.0
b = 2.0
c = -3.0
d = b**2 - 4.0*a*c
e = sqrt(d)
x1 = (-b + e)/2.0/a
xy = (-b - e)/2.0/a
write(*,*) "Hledane reseni: ",x1,x2
end
```

Řízení toku programu

Příkazy se ve Fortranu provádějí postupně od začátku zdrojového textu do konce, není-li řečeno jinak.

Zápis podmínek

Rozvětvení řídícího toku programu lze realizovat třemi typy podmínek:

- ```
  if( podmínka ) příkaz 1
  ```

- ```
  if( podmínka ) then
    příkaz 1
    příkaz 2
  endif
  ```

- ```
  if( podmínka ) then
    příkaz 1
  else
    příkaz 2
  endif
  ```

Tyto zápisy říkají počítači, že má vyhodnotit podmínku a je-li splněna pokračovat prvním příkazem. V opačném případě buď pokračuje druhým příkazem nebo neudělá nic.

Cykly

Opakované provádění příkazů zajišťují cykly.

- ```
  do
    příkaz
  endo
  ```

- ```
  do i = 1,n,krok
  ```
  ```

```
  příkaz
enddo
```

-

```
do while( podmínka )
  příkaz
enddo
```

První cyklus je nekonečný cyklus, opakuje se donekonečna pokud není přerušen například příkazem `exit.` který vyskočí z cyklu. Tato konstrukce umožňuje psát cykly s podmínkou na začátku, na konci i kdekoli jinde.

V dalších dvou případech cykly fungují tak, že říkají počítači aby vyhodnotit podmínku a je-li splněna vstoupit do cyklu, vykonat v něm příkazy, opět testovat podmínku, pokud je splněna vykonat příkazy, atd. While cykus má za podmínku logický výraz, prostřední aritmetický cyklus testuje hodnotu řídící proměnné kterou po každém vyhodnocení těla cyklu zvětší o hodnotu krok.

Ukončení programu

Normálně končí program v okamžiku, když už nemá na vykonání žádné příkazy. Jiná možnost bezpodmínečného ikončení programu je příkaz `stop 'text'` ktery ukončí program a na standardní výstup vypíše 'text'.

Skok a prázný příkaz

Příkaz skoku přenese další provádění programu na zvolený řádek. Příkaz `goto 666` skočí na řádek označený návestím 666. Obyčejně je na řádku prázdný příkaz `continue` který nedělá nic. Dvojice těchto příkazů má především význam při ukončování komplikovaných vnořených cyklů s podmínkami a dále pak při ošetření havarie.

Řešení nelineárních rovnic

Dvě nejpoužívanější jednoduché metody řešení rovnic jsou robustní ale pomalé půlenní intervalu a rychlá ale méně stabilní Newtonova metoda. Na těchto jednoduchých metodách budeme demonstrovat základní použití podmínek a cyklů.

Půlení intervalu

Tahle metoda hledá řešení rovnice v předem zadaném intervalu na jednom jejímž okraji je funkce kladná na na druhém záporná. Program pak dělí zadanný interval na polovinu tak dlouho, dokud není jeho délka menší než předem zadaná hodnota. Jediným problémem je rozhodnout, ve kterém ze dvou vytvořených intervalů je skutečný kořen rovnice. Ten je ovšem opět v takovém intervalu, kde mají hodnoty opačná znaménka a tedy jejich součin je záporný.

```
program pulky
! program resi rovnici cos(x) = 0 metodou puleni intervalu
integer :: i,maxn
real :: a,b,x,fa,fb,fx,tol

! zadani intervalu
a = 1
b = 2

! maximalni pocet opakovani
maxn = 100

! hledana presnost vypoctu
tol = 1e-2

! vypocet funkce na krajich intervalu
fa = cos(a)
fb = cos(b)

i = 1
do

   ! vypocet pulky intervalu a hodnoty v nem
   x = (a + b)/2.0
   fx = cos(x)

   ! vypis aktualniho korene
   write(*,*) "iterace = ",i," hodnota funkce = ",fx," odhad korene = ",x

   ! testy ukonceni
   if( fx == 0.0 ) exit
   if( abs(b - a)/2.0 < tol ) exit
   if( i >= maxn ) exit
   i = i + 1

   ! urceni intervalu ve kterem je koren
   if( fx*fb < 0 ) then
     ! v intervalu x..b je koren
```

```
     a = x
     fa = fx
   else
     ! v intervalu a..x neni koren
     b = x
     fb = fx
   endif
enddo

! vypis vysledku
write(*,*) "pocet iteraci = ",i
write(*,*) "hledany koren = ",x
write(*,*) "hodnota funkce = ",fx

end
```

Newtonova metoda

Tato metoda vychazí z myšlenky obráceneho Taylorova rozvoje funkce:

kterou lze do převést na tvar

Implementace je jednodušší než u metody půlení.

```
program Newton
! program resi rovnici cos(x) = 0 newtonovou metodou
integer :: i,maxn
real :: x,fx,f1,tol

! zadani pocatecniho odhadu minima, napr. z pulek
x = 1.5

! maximalni pocet opakovani
maxn = 100

! hledana presnost vypoctu
tol = 1e-5

! vypocet funkce
f = cos(x)
```

```
do i = 1, maxn

   ! vypocet derivace
   f1 = -sin(x)

   ! vypocet noveho odhadu korene
   if( f1 /= 0.0 ) then
      x = x - f/f1
   else
      stop 'Nulova derivace'
   endif

   ! vypis aktualniho korene
   write(*,*) "iterace = ",i," funkce = ",fx," derivace = ", f1, &
              " odhad korene = ",x,

   ! testy na ukonceni
   if( abs(x) < tol ) exit

enddo

! vypis vysledku
write(*,*) "pocet iteraci = ",i
write(*,*) "hledany koren = ",x
write(*,*) "hodnota funkce = ",fx

end
```

Tento poslední program obsahuje jednu nebo více chyb.

Programové jednotky

Řešení komplikovaného problému lze často zjednodužšit jeho rozdělením na několik podproblémů, které lze vyřešit snáze. V programovacích jazycích tomu odpovídá dělení celého programu na podprogramy které jsou víceméně nezávislé na sobě navzájem. Složením několika podprogramů z určité oblasti pak můžeme konstruovat celé programové knihovny.

Podprogramy a funkce

Ve fortranu existují tři typy programových jednotek:

Hlavní program

Je nejvyšší v hierarchii, začíná slovem `program`, v programu může být jen jediný, všechny proměnné které program obsahuje jsou lokální, tj. nejsou známy žádné jiné programové jednotce

Podprogram

Podprogram je volán hlavním programem nebo jinou funkcí či podprogramem, případně sebou samým. Proměné jsou lokální. Podprogram nevrací žádnou návratovou hodnotu a se svým okolím může komunikovat výhradne prostřednictvím parametrů.

Funkce

Funkce je volána hlavním programem, jinou funkci nebo prodprogramem nebo sebou sama. Proměnné jsou lokální. Funkce musí vracet návratovou hodnotu ale může měnit i svoje parametry.

Obecně je vhodné psát samostatné programové jednotky do samostatných souborů na disku a případě i samostatně překládat. Proto je nevhodné používat něco jako globální proměnné, které navíc komplikují práci ve složitých programech.

Struktura podprogramu nebo funkce je podobná jako u hlavního programu, začíná deklaracemi lokalních proměnných po kterých následuje vlastní kód.

Podprogramy

Podprogram začíná slovem `subroutine` za nímž následuje jeho název a seznam parametrů. Končí slovem `end subroutine`. Parametry slouží na učení typu proměnných a na jejich případné přejmenování. Extrémě důležité je přesně dodržovat pořadí a typy parametrů jinak dostaneme zcela nesmyslné výsledky. Podprogram se z jiné programové jednotky volá příkazem `call` Typický příklad zápisu podprogramu je zde:

```
subroutine podprogram(a,b,x,i)

  real :: a,b,x
  integer :: i

  if( i == 666 ) return
```

```
  write(*,*) "Predane parametry:",a,b,x,i
  x = x + 1

end subroutine
```

Funkce

Funkce je uvozena slovem `function` před kterým ovšem může stát definice typu. Pokud ne, mělo by být někde v definicích funkce řečeno jakého typu je návratová hodnota funkce. Dále následují parametry podobně jako v prípadě podprogramu. Někde ve těle funkce by měla být do funce přiřazena návratová hodnota. Příklad:

```
function funkce(i,x) ! result(y)

  real :: x,funkce
  integer :: i
! real :: y

  if( i == 666 ) then
    funkce = 0.0
!   y = x + 1.0
    return
  endif

  write(*,*) "Predane parametry:",x,i

  funkce = x + 1.0
! y = x + 1.0

end function
```

V příkladu je v komentářích uvedena alternativní metoda návratové hodnoty. Je zapoznámkována nová lokální proměnná jejích přiřazená hodnota se předa volajícímu programu uvedením slova result. V takovém případě ovšem nesmí být název funkce natypován.

Použití

Jako příklad užití spolupráce funkcí a programu zde máme

```
Program Main

  real :: x,funkce
  integer :: i

! volani podprogramu
  call podprogram(2.0,3.0,x,0)
  write(*,*) "x po navratu z podprogramu = ",x

! volani funkce
  write(*,*) "vysledek funkce = ",funkce(1,x)

end
```

Rekurse

Rekurse, tj, volání podprogramu samo sebou se musí překladači explicitně sdělit uvedením slova `recursive` před názvem (`recursive subroutine`, `recursive function`). K použití rekurse se (možná) dostaneme později.

Použití

Jak již bylo řečeno programové jednotky jsou samostatné, takže se dají například samostaně překládat `f90 -c podprogram.f90` s parametrem -c značícím pouze překlad. Tento příkaz vytvoří soubor `podprogram.o`. S ten pak můžeme dak se spustitelnému souboru a vytvořit program např. `f90 podprogram.o program.f90`nebo knihovnu (viz dále).

Moduly

Protože je předávání parametrů mezi programovými jednotkami dost obtížná tak i poměrně omezená věc, byly zavedny tzv. moduly. To jsou programové jednotky vyššího řádu sloužící k větší intergraci samostatných jednotek. Mají následující strukturu:

```
Module modul

! definice typu, promennych

contains
```

```
! funkce, podprogramy, bud uvedene jejich kody nebo inkludnute

end module
```

V první části, která je veřejná, můžeme definovat vlastní typy, nastavit konstatny, proměnné pro celý program a pod. Vše co nemá atribut private je veřejné.

V druhé části jsou pak uvedeny zdrojové kódy procedur a programů. Příkazem `include 'funkce.f90'` můžeme vložit zdrojový text z jiného souboru.

Tato konstrukce slouží jednak ke sdílení společných veřejných proměnných různým jednotkam a jednak ke kontrole předávaných parametrů procedur již překladačem. Sdílení proměnných se budeme věnovat příšte.

Použití modulů je jednoduché, hned na začátku jednotky která jej bude používat se dá příkaz `use 'nejakymodul'` a program se přeloží následujícím způsobem: `f90 modul.f90 program.f90`

Hledání kořenů rovnic

Teď se zaměříme na zcela funkční a elegantní přepsání procedur na hledání kořenů rovnic, kterým jsme se už věnovali. Předním jsme totiž museli hledat kořen pouze určité funkce a ne takové, kterou by jsme si definovali sami. Použití funcí ovšem tento problém zcela elegantně řeší. Uvedeme dvě řešení, obě mají své výhody a nevýhody (jaké?).

Newton v samostatné funkci

Na ukázku si vezmeme newtonovu metodu (je kratší). Napíšeme funkci do samostaného souboru. V tomto případě musíme vytvořit podprogram který ma jako parametry počáteční odhad, funkci a konvergenční parametry. Zde je.

```
subroutine newton(f,df,x,fx,maxn,tol)

! poprogram resi rovnici f(x) = 0 newtonovou metodou,
!
! na vstupu ma funkci f(x), funci df(x) pocitajici jeji derivace,
! pocatecni odhad, povoleny pocet vycisleni funce a prenost
!
! na vystupu je v promenne x ulozen hledany koren


integer :: i,maxn
```

```
real :: x,f1,tol,x0

! vnejsi funkce musi byt oznaceny specifikatorem external aby prekladac
! poznal, ze jde o funkci
real, external :: f,df

x0 = x + 2.0*tol

do i = 1, maxn

   ! vypocet funkce
   fx  = f(x)

   ! vypocet derivace
   f1 = df(x)

   ! vypocet noveho odhadu korene
   if( f1 /= 0.0 ) then
      x = x - f/f1
   else
      stop 'Nulova derivace'
   endif

   ! vypis aktualniho korene
   write(*,*) "iterace = ",i," funkce = ",fx," derivace = ", f1, &
              " odhad korene = ",x,

   ! testy na ukonceni
   if( abs(x-x0) < tol ) exit

   ! uchovani korene
   x0 = x

enddo

end subroutine
```

Newton v modulu

Zde je příklad v modulu. Předávané funkce jsou zde navíc přesněji specifikovány, jak procedura předpokládá, že budou volané.

```
Module Rovnice

contains

  subroutine newton(f,df,x,fx,maxn,tol)

  ! poprogram resi rovnici f(x) = 0 newtonovou metodou,
  !
  ! na vstupu ma funkci f(x), funci df(x) pocitajici jeji derivace,
  ! pocatecni odhad, povoleny pocet vycisleni funce a prenost
  !
  ! na vystupu je v promenne x ulozen hledany koren

    interface
       function f(x)
         real :: f,x
       end function f

       function df(x)
         real :: df,x
       end function df
    end interface

    integer :: i,maxn
    real :: x,f1,tol,x0

    x0 = x + 2.0*tol

    do i = 1, maxn

      ! vypocet funkce
      fx  = f(x)

      ! vypocet derivace
      f1 = df(x)

      ! vypocet noveho odhadu korene
      if( f1 /= 0.0 ) then
        x = x - f/f1
      else
        stop 'Nulova derivace'
      endif

      ! vypis aktualniho korene
      write(*,*) "iterace = ",i," funkce = ",fx," derivace = ", f1, &
```

```
              " odhad korene = ",x,

      ! testy na ukonceni
      if( abs(x-x0) < tol ) exit

      ! uchovani korene
      x0 = x

    enddo

  end subroutine
end module
```

Tento modul pak z hlavního programu použijeme následovně.

```
program Newton
! program resi rovnici cos(x) = 0 newtonovou metodou

use Rovnice

integer :: i,maxn
real :: x,fx,f1,tol

! zadani pocatecniho odhadu minima, napr. z pulek
x = 1.5

! maximalni pocet opakovani
maxn = 100

! hledana presnost vypoctu
tol = 1e-5

call newton(f,df,x,fx,maxn,tol)

! vypis vysledku
write(*,*) "pocet iteraci = ",i
write(*,*) "hledany koren = ",x
write(*,*) "hodnota funkce = ",fx

end

function f(x)
  real :: f,x
```

```
  f = cos(x)
end function

function df(x)
  real :: df,x

  df = -sin(x)
end function
```

Vstup/Vystup

Prostřednictvím vstupů a výstupů komunikuje program s okolím. Tato část jazyka je mimořádně hluboce propracovaná a tím pádem i komplikovaná. Obecně lze říci, že se při každém I/O příkazu přečte/vypíše jeden řádek nebo záznam. Každý příkaz je též spojen s tzv. jednotkou (povinný parametr) která identifikuje jednotlivé připojené soubory. Standardní jednotka pro I/O se identifikuje hvězdičkou. Každý příkaz má též kontrolní parametry umožňující ošetřit chybové stavy.

Příkazy na manipulaci se soubory

Open

Otevření souboru a spojení s jednotkou. Příkaz má tvar:

```
open([unit=]číslo,file='nazev',status='new/old/scratch',
            access='sequential/direct/append',form='formated/unformated',
            action='read/write/readwrite',recl=číslo,...)
```

parameter status udává stav souboru při jeho otevření. Pokud soubor neexistuje a je otevřen s parametrem old, dostaneme chybu. Paremeter scratch vytváří dočasný soubor. Přístup k souboru může být buď sekvenční, přímý nebo můžeme do již existujícího souboru dopisovat. Záznamy v něm jsou buď formatované (obyčejně u souborů typu počítač-člověk) nebo neformatované (soubory typu počítač-počítač).Můžeme též předem specifikovat akci kterouy chceme se souborem dělat a parametrem recl definovat délku záznamů v souboru.

Close

Uzavření souboru. Příkaz má tvar:

```
close([unit=]číslo,status='keep/delete',...)
```

parameter status udává jestli má po uzavření soubor existovat nebo ne.

Inquire

Zjištění informací o souboru. Příkaz má tvar:

```
inquire(file='název',exist=.true./.false.,...)
```

tento příkaz má velké množství nejrůznejších parametrů a lze s ním zjistit o souboru kde co. Nejčastější použití je zjištění, jestli daný soubor existuje nebo ne. Například:

```
logical :: existuje
inquire(file='data.dat',exist=existuje)
write(*,*) existuje
```

Endfile

Zapsání značky konec souboru do souboru. Příkaz má tvar:

```
endfile([unit=]číslo,...)
```

paraemtry má pouze chybové.

Backspace

Vrácení o jeden záznam vzad. Příkaz má tvar:

```
backspace([unit=]číslo,...)
```

paraemtry má pouze chybové.

Rewind

Přetočení souboru na začátek. Příkaz má tvar:

```
rewind([unit=]číslo,...)
```

paraemtry má pouze chybové.

Čtení a psaní do souborů

Čtení

Na čtení záznamů nebo řádků ze souboru používáme příkaz read. Každé jeho zavolání způsobí načtení jednoho záznamu nebo řádku z daného souboru. To znamená, že opakované čtení jednoho řádku lze procés jedině ve spolupráci s příkazem backspace nebo vhodným formátováním. Příkaz přiřadí hodnoty do proměnný uvedených za příkazem.

```
read([unit=]číslo,[fmt=]formát,[end=návěší],..) proměnné
```

jako jednotka může být uvedeno číslo předím otevřeneho souboru, hvězdička udávající standardní vstup nebo znaková konstanta ze které se čte. Formát je udání formátové specifikace nebo hvězdička udávající volný formát. Je-li uveden parameter end a dočteme se na konec souboru, skočí příkaz na předem dané návěští.

Zápis

Na zápis záznamů nebo řádků do souboru používáme příkaz write. Každé jeho zavolání způsobí zapsání jednoho záznamu nebo řádku do daného souboru. Ve fortranu je to obvyklý způsob zpracování. Není zvykem vypisovat na jeden řádek pomocí více příkazů, i když to lze udělat. Příkaz vypíše hodnoty proměnných uvedených za příkazem.

```
write([unit=]číslo,[fmt=]formát,[advance='no'],..) proměnné
```

jako jednotka může být uvedeno číslo předtím otevřeného souboru, hvězdička udávající standardní výstup nebo znaková konstanta do které se zapisuje. Formát je udání formátové specifikace nebo hvězdička udávající volný formát. Je-li uveden parameter advance='no' pak se neprovádí odřádkování a následující příkaz write začne vypisovat na tentýž řádek.

Formát

Formátování je ve fortranu mimořádně bohaté. Zde jsou uvedeny jen nejvíce používané způsoby. Postup při použití formátu je takový, že se postupně berou proměnné které se mají vypsat a asociují se s příslušným formátem. Pokud si neodpovídají typově dojde k chybě. Pokud neodpovídá počet formátových specifikací a počet veličin je v případě, že je formát delší než výspuní veličiny zbytek ignorován, pokud je naopak více výstupních veličin než formátů dojde k odřádkování. Všechny formáty odpovídající veličinám lze opakovat. Při formátovacím řízení se lze předbíhat i vracet.

Volný formát

Nejjednodužší a taktéž nejvíce používaný formát je volný formát uvozený hvězdičkou. V takovém případě se formát přizpůsobuje příslušným I/O veličinám které jsou oddělené mezerou nebo čárkou. Znakové veličiny musí být při vstupu uvedeny v apostrofech nebo uvozovkách.

```
Program degrees
! ukazka vstupu a vystupu volnym formatem
real :: ra1,ra2,dec1,dec2,ra,dec
character(len=80) :: objname

write(*,*) 'Degrees: program na prevod souradnic z hodin na stupne'
write(*,*)

! hlavni cyklus
do
   ! precteni objektu z klavesnice
   write(*,*) 'zadej nazev objektu v apostrofech, hodiny, min-
uty v RA a dec:', &
             '    konec=^D|^Z'
   read(*,*,end=666) objname,ra1,ra2,dec1,dec2
   ra = 15.0*(ra1 + ra2/60.0)
   dec = dec1 + dec2/60.0
   write(*,*) objname,ra,dec
enddo
666 continue

end
```

Formátové konverze

Uživatel může přesně předepsat tvar vypisovaných dat pomocí formátových konverzí. V tomto případě je přesně zadaná celková šířka vypisovaných informací a ty tím pádem nemusí být oddělené nijak.

Formátování pro Integer

V tomto případě má formát tvar

```
Iw[.m]
```

kde w udává celkovou šířku čteného čísla. V případě, že je uvedeno m pak se při výstupu dané číslo doplnuje z leva nulami tak aby mělo číslo minimálně m znaků.

Tabulka 1. Vstup I edicí

Vstupní pole	Edice	Údaj v počítači
_1234	I5	1234
_1234	I5.1	1234
_-_34	I5	-34
_	I1	0
3456	I2	34

Tabulka 2. Výstup I edicí

Údaj v počítači	Edice	Výstupní pole
1234	I5	_1234
12	I5.4	_0012
-34	I2	**

Formátování pro Real

V tomto případě má formát tvary

```
Fw.d      nebo      Ew.d[Ee]    nebo Gw.d[Ee]
```

kde w udává celkovou šířku čteného čísla. V případě, že je uvedeno d pak při výstupu má dané číslo d desetinných míst. Edice F vypisuje čísla v normálním a edice E v exponenciálním tvaru. e pak udává počet míst v exponentu. Na vstupu jsou E a F ekvivalentní. Edice G pak slouží na vhodné zkloubení E a F edice tak aby měl údaj co nejmenší počet znaků. Edice G se uplatňuje opět pouze na výstupu.

Tabulka 3. Vstup E nebo F edicí

Vstupní pole	Edice	Údaj v počítači
_1234.567	F10.5, E10.5	1234.567
1.2e3	F10.0	1.2e3
1. 3	F10.0	1e3

Tabulka 4. Výstup F nebo E edicí

Údaj v počítači	Edice	Výstupní pole
1.234	F5.2	_1.23
0.255	F5.3	-.255
1.234	E8.2	_0.12e+1

Tabulka 5. Výstup G edicí

Údaj v počítači	Výstup pro F10.4	Výstup pro E10.4	Výstup pro G10.4
0.01234567	____0.0123	0.1235e-01	0.1235e-01
-0.1234567	____-0.123	-0.1235e+0	-0.1235___
1.234567	____1.2346	0.1235e+01	_1.235____

Formátování pro Character

V tomto případě má formát tvar

```
A[w]
```

kde w udává celkovou šířku řetezce. V případě, že není w uvedeno pak se odvodí délka dat ze skutečné délky řetezce. Proto se může stát, že řetezec není načten celý. Tento formát je nezbytně třeba, pokud chceme číst zznakové údaje které nejsou ohraničené apostrofy.

Tabulka 6. Vstup A edicí

Vstupní pole	Edice	Údaj v počítači
Fortran 2000	A	Fortran 2000
Fortran 2000	A7	Fortran

Tabulka 7. Výstup A edicí

Údaj v počítači	Edice	Výstupní pole
Fortran 2000	A	Fortran 2000
Fortran 2000	A9	Fortran 2
short	A10	short_____

Další formátovací možnosti

Tzv změna měřítka se provede pomocí konverze

```
kP
```

kde k udává měřítko. Tento možnost je v podstatě užitečná jen v případě, že chceme vypsat data edicí E tak aby na prvním místě nebyla nula. Například formát 1P,E8.3 pro číslo 1.2 vypíše

```
write(*,'(1P,E8.3)') 1.2  ->   1.2e+00
```

místo 0.12e+01.

Na jakoukoli pozici v záznamu se lze dostat přes formátovací značky

```
T, TR, TL
```

přičemž T10 posune následující čtení na znak 10 od začátku záznamu, TR10 o 10 znaků doprava a TL10 o 10 znaků doleva.

Pole

Pole jsou nejzákladnější datová struktura. Protože jsou nesmírně často používanou praktikou při programování matematických algoritmů, ma pro ně Fortran velmi silnou jazykovou podporu. Použití polí je často nepřirozenější způsob zápisu a v případě, že máme patřičnou hardware podporu, vede i k podstatnému zrychlení programů.

Deklarace pole

Pole se deklaruje uvedením typu prvků pole a dimenzí.

Statická pole

Definice statických polí je prostá uvede se dimenze pole prostřednictvím celočíslené konstanty nebo výrazu, který může být vyhodnocen již při kompilaci.

```
real :: pole(100)
```

tento příkaz deklaruje reálné pole o 100 prvcích. První index má hodnotu 1 a poslední 100. K jednotlivým prvkům přistupujeme pomocí indexu zapsaného za jméno pole do závorek. Například v následujícím fragmentu programu přiřadíme do prvku 24 hodnotu -1.0 a pak ji vytiskneme.

```
 pole(24) = -1.0
 write(*,*) pole(24)
```

Jako příklad složitejší deklarace pole ovšem s naprosto stejným výsledkem uvedeme:

```
 integer, parameter :: ndim = 50
 real :: pole(2*ndim)
```

Dynamická pole

Často ovšem neznáme velikost pole v okamžiku kompilace programu. To se často řeší nadimenzováním pole na mnohem větší velikost. Zřejme to ovšem nelze považovat za optimální řešení. Mnohem lépe je deklarovat pole v průběhu běhu programu. V tomto případě uvedene při deklaraci pole do jako mez dvojtečku.

Na vytvoření pole v průbehu programu pak slouži funkce `allocate` s argumentem pole a jeho dimenze. V příkladu je opět uvedno vytváření 100 prvkového pole.

```
 real :: pole(:)

 allocate(pole(100))
```

Práce i chování takto vytvořeného pole ja naprosto stejná jako u pole statického. Navíc jej lze příkazem `deallocate` kdykoli zrušit. Neexistuje ovšem, vzhledem k náročnosti implementace, funkce na změnu velikosti pole.

Automatická pole

Často je třeba v podprogramu vytvořit pole a po jeho skončení toto pole opět zrušit. Ručně to lze provést prostřednictvím dynamických polí, ale ve Fortranu exsituje ještě jednodužší možnost. V případě, že v podprogramu (!) dáme jako dimenzi některý z celočíselných parametrů podprogramu, uděla za nás postřebné alokace a dealokace překladač. Je to stejně náročná metoda na systémové prostředky jako udělat to ručně, ovšem značně to zjednodušší text programu a umožní oddělit vlastní algoritmus od počítačově závyslých prostředků.

```
subroutine demo(ndim)

real :: pole(ndim)

pole(ndim) = -1.0
write(*,*) pole(ndim)
end subroutine
```

V tomto příkladě jsme jako parametr procedury předali dimenzi pole a s ním pak udělali obvyklé procedury. Číslo ndim musí být větší než nula.

Opičárny s polama

Dimensování

Dimensi můžeme také popsat v atributu dimension:

```
real, dimension(3) :: pole
```

nebo můžeme uvést indexování jinak než od 1:

```
real :: pole1 (0:99)     ! C-like
real :: pole2 (-10:10)
```

případně můžeme použít více dimenzí

```
real :: pole0 (100,100)
real :: pole1 (0:99,0:99)     ! C-like
real :: pole2 (-10:10,-10:10)
```

Initializace

Krátké pole lze initialozovat prvkek po prvku prostřednictvím tzv. konstruktoru. V tomto případě máme pole s třemi prvky a inicialuzujeme jejich jednotlivé hodnoty.

```
pole = (/ 0.0, 1.0, 2.0 /)
```

Podpole

Z polí můžeme dělat výřezy. Například rozsekat matici na vektory.

```
real :: a(3,3), b(3), c(2,2)

a =  1.0
c = a(1:2,1:2)

b = a(1,:)
! totozne s
b = a(1,1:3)

! nebo dokonce
b = a(2,3:1:-1)        ! priradi zinvertovany vektor
```

Základní operace s poli

Všechny základní operace které můžeme provádět s čísli můžeme provádět i s poli. V tomto případě se provádějí operace postupně nad jednotlivými prvky. V příkladu je uvedeno maticové sčítání. Složkový a maticový součin.

```
real :: a(3,3), b(3,3), c(3,3)

a =  1.0
b = -1.0

c = a + b
c = a*b
c = matmul(a,b)
```

S poli tedy jdou dělat všechny elementární operace jako s čísly a navíc můžeme použít řadu funkcí:

```
real :: a(3,3), b(3,3), c(3), d(3)

a =  1.0
b =  0.0
c = (/ 1.0, 0.0, 0.0 /)
d = (/ 0.0, 1.0, 0.0 /)

! soucet prvku
write(*,*) sum(c*d)
! totozné s
!     s = 0.0
```

```
 !     do i = 1,3
 !        s = s + c(i)*d(i)
 !     enddo
 !     write(*,*) s


 ! nejvetsi hodnota
 write(*,*) maxval(c)

 ! vsechny hodnoty mensi nez 1
 write(*,*) all(c < 1.0)

 ! dimense pole
 write(*,*) count(c)

 ! skalarni soucet
 write(*,*) dot_produkt(c,d)

 ! vektorovy soucin
 write(*,*) matmul(a,b)

 ! trasnponovani
 write(*,*) transpose(a), transpose(c)
```

Maticové if

je příkaz where.

```
real :: a(100), b(100)

a = 0.0
a(1:10) = 1.0

where( a > 0.0 )
  b = 1.0
elsewhere
  b = -1.0
endwhere

! toto je ekvivalentni s
do i = 1,100
  if( i ≤ 10 )then
    a(i) = 1.0
```

```
  else
    a(i) = 0.0
  endif
enddo

do i = 1, 100
  if( a(i) > 0.0 )then
    b(i) = 1.0
  else
    b(i) = -1.0
  endif
enddo
```

Předávání polí

V principu lze předávat pole dvojím způsobem tak aby volané procedura věděla o jeho dimenzích. Pokud dimenze neuvedeme pak musí (!) mít procedura informace o polích buď uvedením interface nebo uvedením v modulu (pak dělá interface samotný kompiler). Pokud nejsme proti uvádění dimensí nemusí být procedura ani v modulu anu nemusí mít uvedený interface. Pokud tyhle zasady nedodržujeme překlad proběhne v pořádku, ale problémy se projeví při běhu programu.

Klasický styl

V tomto případě předává nadřazená procedura dimense poli ručně. Ve volané proceduře musí být dimenzované předávané pole s těmihle dimensemi.

```
Program vrsek

integer, paramerter :: ndim = 10
real :: a(ndim), b(ndim, ndim)

call spodek(ndim,a,ndim,ndim,b)
end program

subroutine spodek(adim, a, bdim1, bdim2, b)

integer :: adim, bdim1, bdim2
real :: a(adim), b(bdim1,bdim2)

a = 0
b = 0
end subroutine
```

Moduly

V tomto případě předává nadřazená procedura dimense poli automaticky. Ve nemusi být dimense uvedeny. Případně můžeme dimese zistit pomocí standarních funkcí.

```
Program vrsek

use spodky

integer, paramerter :: ndim = 10
real :: a(ndim), b(ndim, ndim)

call spodek(a,b)
end program

Module spodky

contains

subroutine spodek(a, b)

integer :: adim, bdim1, bdim2
real :: a(:), b(:,:)

adim = size(a)
bdim1 = size(b,1)
bdim2 = size(b,2)

a = 0
b = 0
end subroutine
end module
```

Příklad

Operace s maticemi jsou základní operace prováděné s poli. Asi jedna z nejzákladnějších věcí, kterou lze s maticemi provádět je řešit soustavy rovnic použitím Gaussovi eliminační metody. Předvedeme nejjednodužší verzi, bez pivotování.

Vzorce i algoritmus jsou všeobecně známy. My použijeme upravený algoritmus s Hřebíčka. Doporucuju srovnat s puvodnim algoritmem bez pouziti maticoveho zapisu. Je vhodné připomenout, že celý postup by šlo zapsat pomocí maticových operací, ovšem vzhledem k úspoře místa a prostředků to není vhodné.

```
subroutine Gauel(n,a,b)
! n je dimense matice, b je vektor pravych stran, na vystupu
! obsahuje reseni

integer :: n
real :: a(n,n),b(n)

do k = 1, n

  ! vydeleni prvku vpravo od diagonaly diagonalnim prvkem
  a(k,k+1:n) =  a(k,k+1:n)/a(k,k)
  b(k) = b(k)/a(k,k)

  ! eliminace prvku pod diagonalou
  do i = k + 1, n
     do j = k + 1, n
       a(i,j) = a(i,j) - a(i,k)*a(k,j)
       b(i)   = b(i)   - a(i,k)*b(k)
     enddo
  enddo

enddo

! zpetna substituce
  do i = n - 1, 1, -1
     do k = i + 1, n
       b(i) = b(i) - a(i,k)*b(k)
     enddo
  enddo
end subroutine

Program tgauel

real :: a(3,3),b(3)

a = 1
b = 0

call gauel(3,a,b)

write(*,*) b
end
```

LU Rozklad

Pole jsou neuvěřitelně silný programovací prostředek. Umožnují jako jediné objekty provádět rozsáhle operace nad jejími prvky najednou a tak umožňují využití počítačů naplno. Je to samozřejmě společně s jiným velmi silným prostředkem cykly. Fortran ovšem umožňuje zacházet s poli mnohem elegantněj.

LU rozklad je určený na řešení soustav lineárních rovnic. Na rozdíl od Gaussovy eliminace je ovšem přímo vyvinuta na použití maticových operací. Uvedeme jeho modul Croutuv rozklad.

Zajimavé je použití proměnné ix. Ta není viditelná pro vnejší svět mimo modulu. Slouží na uložení mezivýsledků indexů při přehazování řádků během pivotování. Pro většinu použití není potřeba a proto je viditelná pouze uvnitř modulu. Pokud je z nějakých důvodů potřeba jej přece jen použít pak je na její získání třeba uvedena subroutina getix. Důležité je, správně alokovat a dealokovat pamět tak aby v průběhu běhu programu nezůstávaly v paměti její nepoužitelné zbytečné kusy.

```
Module Crout

  integer, allocatable, private :: ix(:)

contains

subroutine lu (n,a,d)

  implicit none
  integer :: n, i, k, k1
  real :: d,a(:,:),s(n),piv

  deallocate(ix)
  allocate(ix(n))
  d = 0.0
  do k = 1, n
    piv = 0.0

! Vypocet L
    do i = k, n
      a(i,k) = a(i,k) - sum(a(i,1:k-1)*a(1:k-1,k))
      if( abs( a(i,k) ) > piv )then
         k1 = i
         piv = abs( a(i,k) )
      endif
    enddo
! Singularni matice
    if( piv == 0.0 ) return
! Vymena radku
    if( k /= k1 )then
```

```
        s = a(k,:)
        a(k,:) = a(k1,:)
        a(k1,:) = s
    endif
    ix(k) = k1

! Vypocet U
    do j = k + 1, n
        a(k,j) = (a(k,j) - sum(a(k,1:k-1)*a(1:k-1,j)))/a(k,k)
    enddo
enddo

d = 1.0
end subroutine

subroutine lusol(n,a,b)

Integer :: n, i, k
real :: a(:,:),b(:),s

if( .not. allocated(ix) ) stop 'Matrix not factorized.'

! prehozeni indexu
do i = 1, n - 1
   if( ix(i) /= i )then
      s = b(i)
      b(i) = b(ix(i))
      b(ix(i)) = s
   endif
enddo
! zpetna substitice do spodni matice
do i = 1, n
  b(i) = (b(i) - sum(a(i,1:i-1)*b(1:i-1)) / a(i,i)
enddo
do i = n - 1, 1, -1
  b(i) = b(i) - sum(a(i,i+1:n)*b(i+1:n))
enddo
end subroutine

subroutine getix(n,ixx)

real :: ixx(:)

ixx(1:n) = ix(1:n)
end subroutine
```

```
! jaka bude inverze?

end module
```

Čtení tabulek a obrázků

Zajímavá aplikace modulů a polí je při práci s tabulkami a obrázky. Problém je podobně řešitelný v různých jazicích. Zde si ukážeme čtení datového souboru (databáze) ve fortranu. Cvičně si zvolíme soubor s následující strukturou:

Tabulka 8. Struktura databazové tabulky

Část souboru	Údaj	délka
Hlavička	délka v x	10 znaků
Hlavička	délka v y	10 znaků
Hlavička	kometář	40 znaků
Data	záznamy	10 čísel na radek

každý záznam v tomhle souboru obsahuje 10 čísel na řádek. To znamená, že bez ohledu na délku v X nebo Y čteme pořád 10 číselné záznamy.

```
program pokus1

! mozne naivni reseni

integer :: nx,ny
character(len=40) :: com
real,allocatable :: data(:,:)

read(*,"(3I10,I40)") nx,ny,com
allocate(data(nx,ny))

do
  read(*,*,end=220) data(?????)
enddo
220 continue
end
```

Problem je s indexováním pole data. Jistě by šlo nejakým strašně složitým způsobem přetrasfomovávat indexy. Jiné řešení by bylo pomocí standardní funkce pack namapovat matici do vektoru a pak číst z něj. Ovšem existuje podstatně elegantnejší řešení.

```
Module databaze

character(len=40), private :: com = ' '
integer, private :: nx=0,ny=0
real, private, allocatable :: data(:,:)
logical :: databuf
private :: readdata

contains

subroutine opendb(unit,name)

integer :: unit
character(len=*) :: name

open(unit,file=name,status='old')
read(*,"(3I10,I40)") nx,ny,com
allocate(data(nx,ny))
databuf = .false.

end subroutine

subroutine getxy(nx1,nx2)

integer :: nx1,ny1

nx1 = nx
ny1 = ny
end subroutine

function readdb(unit,i,j)
integer :: unit,i,j
real :: readdb

if( .not. databuf ) then
  call readdata(unit,i,j)
  databuf = .true.
endif
readdb = data(i,j)
end function
```

```
subroutine readdata(unit,x,y,data)

integer :: unit, i, x, y
real :: data(:)

i = 1
do while ( i ≤ (nx*ny)/10 + 1)
  read(unit,*) data(i:i+80)
  i = i + 1
enddo

end subroutine

subroutine closedb(unit)

integer ::: unit

close(unit)
nx = 0
ny = 0
com = ' '
deallocate(data)
end subroutine

end module

Program pokus2

use database

! lepsi reseni nacitani

integer nx,ny

call opendb(1,"databaze.dat")
call getxy(nx,ny)
write(*,*) "posledni prvek =",readdb(1,nx,ny)
if( status /= 0 ) stop 'Error while data read.'
closedb(1)
end program
```

Vypadá to uměle a složitě, ale v případě použití složitho souboru to velmi usnadní práci a čtení. Kromě toho je přesně s tímhle stylem definován astronomický FITS formát a programováno čtení a psaní do souboru v Unixu. Je totiž méně HW náročné načíst najednou 1kB z disku a pak z něj postupně číst, když je potřeba než provádět operace po bytech.

Datové typy

Pamět je z hlediska procesoru jen řada bytů a na to aby se stala pamět srozumitelná pro programy musí její obsah interpretovat. Pokud jde o data programu na ty se program dívá prostřednictvím datových typů.

Základní typy dat jsou: `integer`, `real`, `character`, `logical`, `complex`.

Celá čísla jsou reprezentována na intelech 32bitovým číslem s rozsahem -2**31..2*31, všechny ostatní procesory už dnes používají 64bitovou representaci celočíslenýn s rozsahem -2**63..2**63. Často jde překladačí sdělit jestli chceme používat 32 nebo 64 bitovou přesnost jako základní.

Logická data jsou reprezetována stejným typem jako celá čísla. Je věcí překladače jak zakoduje logickou hodnotu `.true.` a `.false.`

Specifikace přesnosti reálných a komplexních dat

V numerické praxi je často potřeba specifikovat numerickou přesnost s jakou mají vystupovat čísla ve vyhodnocovaných výrazech. Tu lze zadat jak na základě znalosti vnitřního zobrazení dat na daném počítači tak na základě skutečně udané přesnosti pomocí počtu desetiných míst nebo čísel v exponentu. Obyčejně je délka reálného čísla stejná jako u celého čísla, tj 32bit na intelech. U reálných čísel je ovšem možné definováním typu `double precision` specifikovat číslo s 2x větším rozsahem. Přehled je vidět v tabulce.

Tabulka 9. Datové typy na Intelech

typ	rozsah	přesnost
32bit integer	-2147483648..2147483648	
32bit reál	-1e38..1e38	6 desetiných míst
64bit reál	-1e308..1e308	16 desetiných míst

Jak je vidět je při použití různých čísel programů a dat značný zmatek v délkách reprezentujících to či ono. Proto došlo k zavedení tzv. numerických typů (kind) pomocí kterých můžeme udat kind daného čísla. Tyto kindy jsou ovšem implementačně závyslé a je tedy třeba nějak zajistit jejich systémovou nezávyslost. Proto byly zavedeny standarní funkce umožňující přesně specifikovat co chce. Jejich přehled je uveden v tabulce.

Tabulka 10. Funkce typů dat

funkce	popis
selected_int_kind(exp)	vybere takový kind který umožní používat celá čísla v rozsahu -10**exp..10**exp
selected_real_kind(digit,exp)	vybere takový kind který umožní používat reálná čísla v rozsahu -10**exp..10**exp s přesností na digit míst, exp je nepovinné

Je ovšem třeba zdůraznit, že si nelze vybírat jakoukoli kombinaci parametrů. Pro daný typ je třeba mít HW podporu.

Pomocí parametrů je možné specifikovat typ a ten používat dále v proramu. To bude vidět v ukázce. Snadno se tak dají programy modifikovat při přenosu na jiný počítač případně z jednoduché do dvojnásobné přesnosti a podobně.

```
program kindy

! specifikujeme presnost realnych cisel
integer, parameter :: Q = selected_real_kind(16,100)

real(Q) :: x

x = 2.0_Q*atan2(1.0_Q,0.0_Q)       ! = 2*atan(1/0) = pi
write(*,*) "real kind = ",Q," pi = ",x

end program kindy
```

Ovšem někdy je třeba vědět další informace o datových typech v průběhu výpočtu. Například u numerické derivace je nejpřesnejší volit krok jako druhou odmocninu z počtu platných desetiných míst. Pak se nám mohou hodit fuknce z následující tabulky. Argumenty funkce může být jak konstanta tak proměnná.

Tabulka 11. Specifikcé funkce typů

funkce	popis
digits(x)	počet platných míst
epsilon(x)	strojové epsilon
huge(x)	strojové nekonečno
[min,max]-exponent(x)	exponent
precision(x)	přesnost

funkce	popis
radix(x)	báze x
range(x)	rozsah exponentů
tiny(x)	nejmenší kladné číslo

Tabulka 12. Manipulace s čísly

funkce	popis
exponent(x)	vraci exponent čísla x
fraction(x)	desetinná část x
nearest(x)	nejbližší zobrazitelné číslo u čísla x
set_exponent(x)	nastaví exponent u čísla x
spacing(x)	rozdíl mezi nejbližšími čísly

Příkladem použití těchto funkcí může být například test na rovnost dvou reálných čísel.

```
program test

real :: a,b

a = 1.0
b = 2.0**0.0

if( abs(a - b) < epsilon(a) )then
  write(*,*) "cisla jsou stejna: ",a,b
else
  write(*,*) "cisla nejsou stejna: ",a,b
endif
end program
```

Odvozené typy

Pomocí základních typů umožňuje fortran též definovat odvozené typy. To umožňuje konstrukci vlastních programových knihoven, použití pointerů a podobně.

Vlastní typy se definují pomocí popisu `typedef`. Slouží k definování strukturovaných dat. Takto nově definovaný typ pak použijeme příkazem `type`. Na složky typu se dostaneme prostřednictvím operátoru `%` Celé si to ukážeme v praxi, kdy se pokusíme nadefinovat operace s quaterniony což jsou čtveřice čísel analogických komplexním číslům. Dnes jsou ve fyzice nahrazeny spinory. Napíšeme modul, který definuje jak typ tak základní operace s nimi. Přitom provedeme takzvané přetížení operátorů. Pomocí definice typu můžeme též rozšířit množinu základních operací běžných operátorů. Zároveň můžeme vytvořit vlastní operátor. Překladač pak na základě proměnných typů, které do daného výrazu vstupují určí jaký je skutečný význam daného operátoru.

```
module quater
  ! modul umoznujici pouziti kvaternionu v programech

  ! definice typu
  type quaternion
     real :: x1, x2, x3, x4
  end type quaternion

  ! scitani
  interface operator(+)
     module procedure addq
  end interface

  ! odcitani
  interface operator(-)
     module procedure subq
  end interface

  ! nasobeni
  interface operator(*)
     module procedure mulq
  end interface

contains

  function addq ( a,b )        ! addq = a + b
    type (quaternion) :: addq
    type (quaternion) :: a,b

    addq = quaternion( a%x1 + b%x1, a%x2 + b%x2, a%x3 + b%x3, a%x4 + b%x4)

  end function addq

  function subq ( a,b )        ! subq = a - b
    type (quaternion) :: subq
    type (quaternion) :: a,b

    subq = quaternion( a%x1 - b%x1, a%x2 - b%x2, a%x3 - b%x3, a%x4 - b%x4)

  end function subq
```

```
  function mulq ( a,b )        ! mulq = a * b
    type (quaternion) :: mulq
    type (quaternion) :: a,b

    mulq = quaternion( a%x1 * b%x4 + a%x4 * b%x1 + a%x2 * b%x3 -
a%x3 * b%x2, &
                       a%x2 * b%x4 + a%x4 * b%x2 + a%x3 * b%x1 -
a%x1 * b%x3, &
                       a%x3 * b%x4 + a%x4 * b%x3 + a%x1 * b%x2 -
a%x2 * b%x1, &
                      -a%x1 * b%x1 - a%x2 * b%x2 -
a%x3 * b%x2 + a%x4 * b%x4  )

  end function mulq

end module

program qua

  use quater

  type(quaternion) :: a,b,c

  a = quaternion( 1.0, 1.0, 1.0, 1.0)
  b = quaternion( 2.0, 3.0, 4.0, 5.0)

  c = a + b
  write(*,*) c

  c = a - b
  write(*,*) c

  c = a * b
  write(*,*) c

end
```

Jinou zajímavou možností je definovat generické funkce. Například potřebujeme mít v knihovně funkci fungující poněkud jinak v závislosti od typu vstupních parametrů.

```
Module kvadry

  interface qudra
    subroutine qudrar(a,b,c,x1,x2)
      real :: a,b,c,x1,x2
```

```
    end subroutine

    subroutine qudrac(a,b,c,x1,x2)
      complex :: a,b,c,x1,x2
    end subroutine
  end interface quadra

contains

subroutine qudrar(a,b,c,x1,x2)
 real :: a,b,c,x1,x2,d

 d = b**2 - 4.0*a*b
 if( d > 0 ) then
   x1 = (-b + sqrt(d))/2.0/a
   x2 = (-b - sqrt(d))/2.0/a
 else
   stop 'nelze resit v oboru realnych cisel'
 endif

end subroutine

subroutine qudrac(a,b,c,x1,x2)
 complex :: a,b,c,x1,x2,d

 d = b**2 - 4.0*a*b
 x1 = (-b + sqrt(d))/2.0/a
 x2 = (-b - sqrt(d))/2.0/a

end subroutine

end module
```

Dynamické datové typy

Konstrukce odvozených typů umožňuje rovněž použití dynamickách datových stuktur profilujících současné programovací jazyky. pomocí atributů `pointer` vytvoříme proměnnou která může ukazovat na nějaké místo v paměti a toto místo interpretuje jako specifiovaný datový typ. Na začátku programu ovšem není definováno a tak je jeho hodnota zcela deurčitá. Lze ji definovat pomocí proměnné s atributem `target` což je uplně normální proměnná která ovšem může sloužit jako cíl pro pointer. Přiřazení se dělá prostřednictvím operátoru =>. Ukážeme si to v praxi. Následující program vytvoří linkovaný seznam. (tohle chce asi trochu víc popsat).

```
program LinkedList

type(node)
  real :: data
  type(node), pointer :: next
end type node

type(node), pointer :: list, current

nullify(list)                        ! initializace listu na nullu
allocate(list)                       ! vytvori pamet pro promenou
list%data = 0.0
allocate(list%next)                  ! alokuje dalsi prvek seznamu
list%next%data = list%data + 1.0
nuliffy( list%next%next)

current => list
do while( associated(current) )
   write(*,*) current%data
   current => current%next
enddo
end program
```

Rozsáhlé projekty: CVS,make,knihovny

Na udržování rozsáhlých projektů byla vyvynuta řada nástrojů. V podstatě se dělí do několika kategorií. Jednak jsou to nástroje na udržování konzistentního projektu při spolupráci řady lidí (CVS, RCS). Dále pak nástroje na kompilaci (udržování informací o tom co je preložene zmeněné a pod). Tyto naástroje mohou být například aplikovány na vývoj nějaké knihovny.

Na internetu existuje rozsáhlá literatura k tomu to tématu. Nejdůležitejší asi jsou: Správa projektů pomoci CVS [*http://www.linux.cz/noviny/2001-04/clanek08.html* (http://www.linux.cz/noviny/2001-04/clanek08.html)], Program-Library-HOWTO [*http://www.linux.cz/linuxdoc/HOWTO/Program-Library-HOWTO/index.html* (http://www.linux.cz/linuxdoc/HOWTO/Program-Library-HOWTO/index.html)].

Dále se budeme věnovat vývoji nějaké programové knihovny. Pod tímto názvem se myslí několik procedur a funkcí řešící problém s určité oblasti. Knihovny které se používají na počítači se tradičně dělí na několik skupin podle způsobu, jakým interagují s programem, který používá jejich funkce. Základní dělení je na statické, které se staticky pridají (přilinkují) k programu, takže ten se může odnést na disketě jinam, spustit a tam pojede (tj program i knihovna jsou v jednom souboru na disku). Dále pak na sdílené, které se k programu nijak nepřičelnují a ten pouze ví kde je hledat a pokud potřebuje nějakou funkci z knihovny, tak si ji najde a vyvola z jiného souboru na disku. Extrémním případem techto shared knihoven jsou tzv. dynamické knihovny (pluginy) které si program sám vyhledáva na disku při běhu, takže uživatel může například měnit knihovnu a tím i metody výpočtu apod.

Bylo by velmi težkopádné dokola překládat všechny funkce knihovny při malé změně jedné funkce a proto je třeba mít nástroj, který bude schpoen rozpoznat co se změnilo a co nikoli a podle toho daný soubor přeložit nebo jej nechat. Tím je program

```
make
```

. Je řízen pomocí

```
Makefile
```

souboru který obsahuje pravidla a akce které mají udělatr to či ono.

Vytvoření knihovny

Použití všemožných nástrojů si ukážeme na fiktivním případě tvorby programové knihovny.

Naším úkolem bude vytvořit statickou knihovnu pod Unixovým systémem se dvěma funkcema na jednoduché statistické funkce průměr a jeho chybu.

Nejprve napíšeme do souboru tyto dvě funkce. V souboru prumer.f90 bude:

```
function prumer(n,x)
integer :: n
real :: prumer, x(n)

prumer = sum(x(1:n))/n
end function
```

Soubor chyba.f90 pak bude obsahovat:

```
function chyba(n,x,prumer)
integer :: n
real :: chyba, x(n), prumer

chyba = sum((x(1:n)-prumer)**2)/(n - 1)
end function
```

Sestavení Makefile je jednoduché:

```
# Makefile pro std

libstd.a: chyba.o prumer.o
	ar rv libstd.a prumer.o chyba.o; \
	ranlib libstd.a

prumer.o: prumer.f90
	f90 -c prumer.f90

chyba.o: chyba.f90
	f90 -c chyba.f90
```

Poznámky. Nejdůležitější je pamatovat si, že mezery před odsazenými příkazu jsou dělané tabelátorem TAB. Příkaz ar vytváří archiv z několika souboru a ranlib pak jeho index. Parametr -c říká překladači, aby se soubor pouze přeložil a nevyvářel se program.

Takto vytvořenou knihovnu pak můžeme v nějakém našem programu použít zcela přirozeně:

```
program zk

real :: x(3) = (/ 1.0, 2.0, 3.0 /)

write(*,*) prumer(3,x)
end program
```

Který přeložíme pomocí příkazu:

```
f90 zk.f90 -L. -lstd
```

Parametr -L označuje adresář ve kterém se má hledat knihovna libstd.a která je specifikována jako druhý parametr -l bez předpony lib.

Grafika

Žádný počítačový jazyk si nemůže z důvodů přenositelnosti dovolit přímo do své definice zahrnout grafické operace, které jsou silne závyslé od HW a navíc dochází v této oblasti k masivnímu rozvoji technologií a žádné všeobecně uznávané grafické rozhraní neexistuje. Proto pokud nějaký program chce používat operace s grafikou musí použít nějakou grafickou knihovnu. Z hlediska programování je to jakakoli knihovna a proto s implementací nebývají žádné problémy. Ukážeme použití nejrozšířenejší nízkoůrovňové knihovny OpenGL ve fortranu. Není to ovšm jediná použitelná knihovna.

PgPlot

Nejsofistikovanější fortranovskou knihovnou je PgPlot *[http://www.astro.caltech.edu/~tjp/pgplot/]* (http://www.astro.caltech.edu/~tjp/pgplot/) od Tima Pearsona která umí všechny možné grafické operace včetne zobrazování obrázků, kreslení grafů a podobně. Všetně rozsáhlého manuálu je volně k dispozici.

DISLIN

Knihovna DISLIN *[http://www.linmpi.mpg.de/dislin/]* (http://www.linmpi.mpg.de/dislin/) je o poznání jednodužší ale zase se může hodit pro jednoduché projekty. Volně k dispozici je ovšem jen verze pro f90.

OpenGL+GLUT

Obě uvedené knihovny jsou určené spíše pro neinteraktivní práci, vykreslují ale neumožňují modifikovat data a podobně. Naproti tomu OpenGL je nízkourovňová knihovna umožňující programovat jakékoli grafické aplikace bez omezení. Existují porty pro všechny možné jazyky. OpenGL i ostatní grafické knihovny pracuje na poněkud jiném principu, než jsme při obyčejném programování zvykli. Obyčejně všechny programy vykonávají příkazy od začátku do konce a pak skončí, maximálně se občas uživatele na něco zeptají. Ovšem tenhle model ovládání programů je zcela nepoužitelný v případě, ze máme aplikaci s menu a podobně, kde je běh programu různě měněn v závislosti například na uživatelově rozhodnutí.

Proto OpenGL pracuje jinak. Při startu se nejprve vytvoří procedury, které obhospodařují jednotlivé akce uživatele případě dalších zařízení. Na začátku se též obyčejně vykreslí do paměti grafické karty obrázek a pokud chce uživatel například myší otáčet obrázek pak jen grafická karta přepočítává základní objekt tak ja by jej viděl uživatel a data pak posílá na obrazovku. Program pak je vlastně sada příkazů pro HW procesor.

Ukázka, kterou uvedeme bude vykreslovat torus (pneumatiku) podle toho v jaké poloze je zrovna ukazatel myši na obrazovce. Dále bude ukazovat jeho souřadnice. Levé tlačítko myši ukáže jednduché, menu například umožňující ukončit program. Zajímavé je experimentovat s polohou a barvou světelných zdrojů které torus osvětlují.

```
! vzorova implmentace GL+GLUT

Module callback

use opengl_gl
use opengl_glu
use opengl_glut
```

```
implicit none

real :: px,py

contains

subroutine init

use opengl_gl
use opengl_glu
use opengl_glut
character(len=20) :: out = "ahoj svete"
integer(glint) :: i,image_data(3*40*40)
contains

subroutine init

use opengl_gl
use opengl_glu
use opengl_glut
character(len=20) :: out = "ahoj svete"
integer(glint) :: i,image_data(3*40*40)

call glclearcolor(0.0, 0.0, 1.0, 1.0)
call glmatrixmode(GL_PROJECTION)
call glloadidentity()
call gltranslatef(-0.1,0.1,0.0)
call glrotatef(30.0,0.0,0.0,10.0)

call glnewlist(1,GL_COMPILE)
call glutsolidtorus(0.275_gldouble,0.45_gldouble,50,55)
call glendlist

call glLightfv(GL_LIGHT0, GL_DIFFUSE, (/1.0, 1.0, 0.0, 0.0 /))
call glLightfv(GL_LIGHT0, GL_POSITION, (/0.0, 0.0, 10.0, 10.0 /) )
call glEnable(GL_LIGHT0)
call glEnable(GL_LIGHTING)

call glMaterialfv(GL_FRONT, GL_AMBIENT, (/0.4, 0.4, 0.4, 0.0/))

end subroutine init

subroutine display
```

```
use opengl_gl
use opengl_glu
use opengl_glut

integer :: i
character(len=80) :: data

call glclear(GL_COLOR_BUFFER_BIT)

write(data,*) px,py
call glrasterpos2f(-0.1,0.95)
do i = 1, len_trim(data)
   call glutbitmapcharacter(GLUT_BITMAP_TIMES_ROMAN_24,ichar(data(i:i)))
enddo

call glrotatef(px/100.0,00.0,10.0,0.0)
call glcalllist(1)
call glutswapbuffers
call glflush

end subroutine display

subroutine mouse(x,y)
integer(kind=glcint) , intent(in out) :: x,y
integer :: i

write(*,*) x,y
px = x+200!abs(x-150.0)+10.0
py = y+200!abs(y-150.0)+10.0

call display

end subroutine mouse

subroutine mainmenu(value)

integer, intent(inout) :: value

!write(*,*) "value = ",value

if( value == 1 ) then
   write(*,*) "help"
endif
if( value == 666 ) stop
```

```
end subroutine mainmenu

end Module callback

program zk1

use opengl_gl
use opengl_glu
use opengl_glut
use callback

integer(kind=glcint) :: iwin, imenu
integer(kind=glenum) :: itemp


call glutinit()
call glutinitdisplaymode(GLUT_DOUBLE + GLUT_RGB)

iwin = glutcreatewindow("zk1")

! menu
imenu = glutcreatemenu(mainmenu)
call glutaddmenuentry("Help",1)
call glutaddmenuentry("Quit",666)
call glutattachmenu(GLUT_RIGHT_BUTTON)

call init
call glutmotionfunc(mouse)
call glutdisplayfunc(display)

call glutmainloop()

end program zk1
```

Jak to celé funguje? Nejprve je třeba knihovnu i window system initializovat. To lze demonstrovat na jednoduchem programu:

```
Program InitGL

use opengl_gl
use opengl_glu
use opengl_glut

integer(kind=glcint) :: iwin, imenu
integer(kind=glenum) :: itemp
```

```
call glutinit()
call glutinitdisplaymode(GLUT_RGB)

iwin = glutcreatewindow("initgl")

call glutmainloop()

end program initgl
```

Tento program zinitializuje knihovnu v RGB modu a vytvoří prázdné okno se jménem initgl a spusti hlavní tzv. zpracovávač událostí. Ten je, zhruba řečeno, představován nekonečným cyklem ve kterém se testují stavy jednotlivých zařízení (klavesnice, myš) a v případě, že se například uživatel zmáčne klávesu tak vyvolá asociovanou funkci (viz dále). V tomto případě ovšem žádné nejsou a tak program nebude reagovat na nic a bude třeba jej nasilím sejmout.

Ovládání je realizováno pomocí tzv. callbacků což jsou funkce, které můžeme svázat s nejakou uživatelovou akcí. Přiřazení funkcí se děje pomocí `glutmotionfunc(), glutdisplayfunc()` funkcí které zaregistrují danou funci jako callback. Můžeme tak ošetřit jednotlivé akce uživatele. Například v subroutinw `mouse` kterým gl předává jako parametry polohu ukazatele myši v aktualním okně vypisujeme tyto souřadnice na stadradni vystup. Na základě těchto souřadnic pak můžeme otáčet s objektem.

Vlastní nadefinování a vytvoření objektu je v proceduře init, kde se v paměti GL vyrobí objekt a projekce ze které se uživatel dívá. Dále pak poloha světelného zdroje na osvětlení scény.

Většina počítačů ma nějakou podporu GL v sobě, většinou je to knihovna messa která umožňuje fungovat jako GL knihovna i na strojích kde GL neni k dispozici, dochází ovšem k podstatnému zpomalení. Fortran interface lze stahout z *[http://math.nist.gov/f90gl/]* (http://math.nist.gov/f90gl/), kde jsou zdrojáky které po přeložení vytvoří jak fortranovsky interface tak moduly s daty tak aby šlo normálně používat OpenGL v F90. Nakonec uvádíme Makefile potřebný pro přeložení ukázkových programů.

```
OGLLIBDIR = -L$(HOME)/f90gl/lib
#-L/usr/lib

# the directory containing the X11 libraries
X11LIBDIR = -L/usr/X11R6/lib/

# the fortran 90 libraries for OpenGL, including GLUT, GLU and OpenGL
F90GLUTLIB = -lf90glut -lf90GLU -lf90GL -lglut -lGLU -lGL

# the X11 libraries
X11LIB =   -lXt -lXmu -lXi -lXext -lX11 -lm
```

```
# the f90 compiler flag for specifying the location of MOD files
MODS = $(HOME)/f90gl/include/GL

# fortran 90 compiler and compiler flags
F90 = ifc
F90FLAGS =

# fortran 90 compiler flag for fixed source form
FFIXED = -FI

#----------- end of user configuration parameters ------------

all: glkinds.o interf.o intrfglt.o intrfglu.o fwrapglt.o fwrap.o fwrapglu.o \
zk zk1

zk: zk.f90
$(F90) $(F90FLAGS) -o zk zk.f90 \
$(OGLLIBDIR) $(F90GLUTLIB) $(X11LIBDIR) $(X11LIB)

zk1: zk1.f90
$(F90) $(F90FLAGS) -o zk1 zk1.f90 \
$(OGLLIBDIR) $(F90GLUTLIB) $(X11LIBDIR) $(X11LIB)

fwrapglt.o:
$(F90) $(F90FLAGS) -c $(MODS)/fwrapglt.f90 \
$(OGLLIBDIR) $(F90GLUTLIB) $(X11LIBDIR) $(X11LIB)

fwrap.o:
$(F90) $(F90FLAGS) -c $(MODS)/fwrap.f90 \
$(OGLLIBDIR) $(F90GLUTLIB) $(X11LIBDIR) $(X11LIB)

fwrapglu.o:
$(F90) $(F90FLAGS) -c $(MODS)/fwrapglu.f90 \
$(OGLLIBDIR) $(F90GLUTLIB) $(X11LIBDIR) $(X11LIB)

glkinds.o:
$(F90) $(F90FLAGS) -c $(MODS)/glkinds.f90 \
$(OGLLIBDIR) $(F90GLUTLIB) $(X11LIBDIR) $(X11LIB)

interf.o:
$(F90) $(F90FLAGS) -c $(MODS)/interf.f90 \
$(OGLLIBDIR) $(F90GLUTLIB) $(X11LIBDIR) $(X11LIB)

intrfglt.o:
$(F90) $(F90FLAGS) -c $(MODS)/intrfglt.f90 \
```

```
$(OGLLIBDIR) $(F90GLUTLIB) $(X11LIBDIR) $(X11LIB)

intrfglu.o:
$(F90) $(F90FLAGS) -c $(MODS)/intrfglu.f90 \
$(OGLLIBDIR) $(F90GLUTLIB) $(X11LIBDIR) $(X11LIB)

clean:
rm -f *.o *.d work.pc*
```