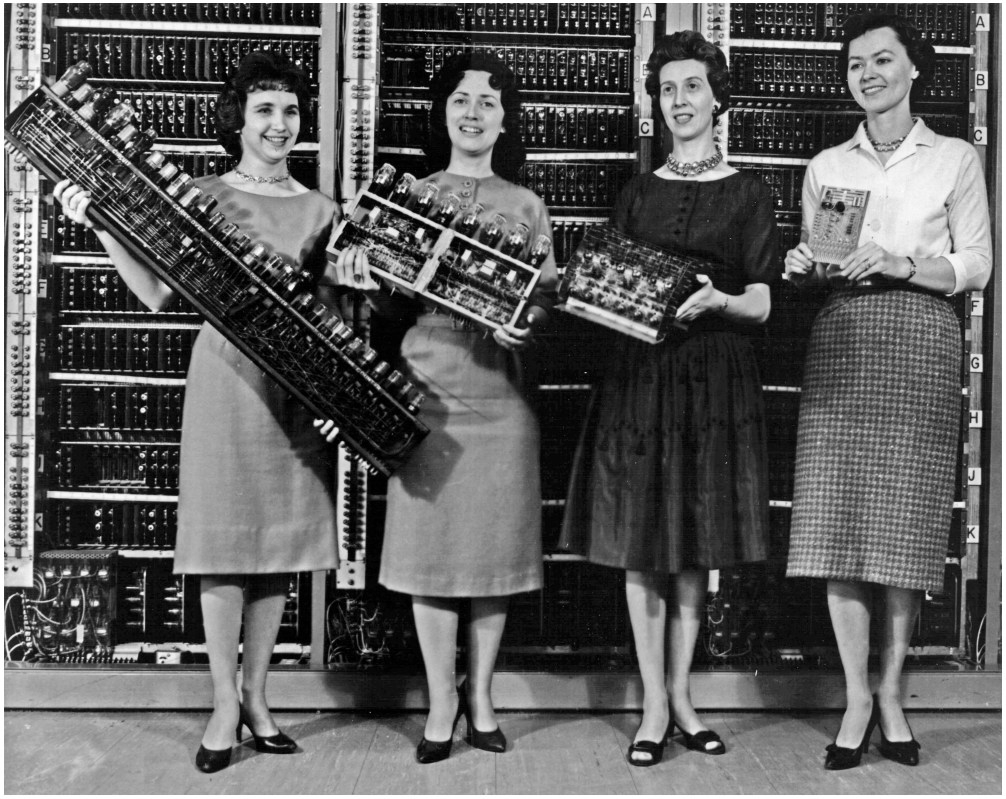
Programadoras do ENIAC



- Betty Snyder Holberton (1917-2001)
- Betty Jean Jennings Bartik (1924-2011)
- Ruth Lichterman Teitelbaum (1924-1986)
- Kathleen McNulty (1921-2006)
- Frances Bilas Spence (1922-2012)
- Marlyn Wescoff Meltzer (1922-2008).

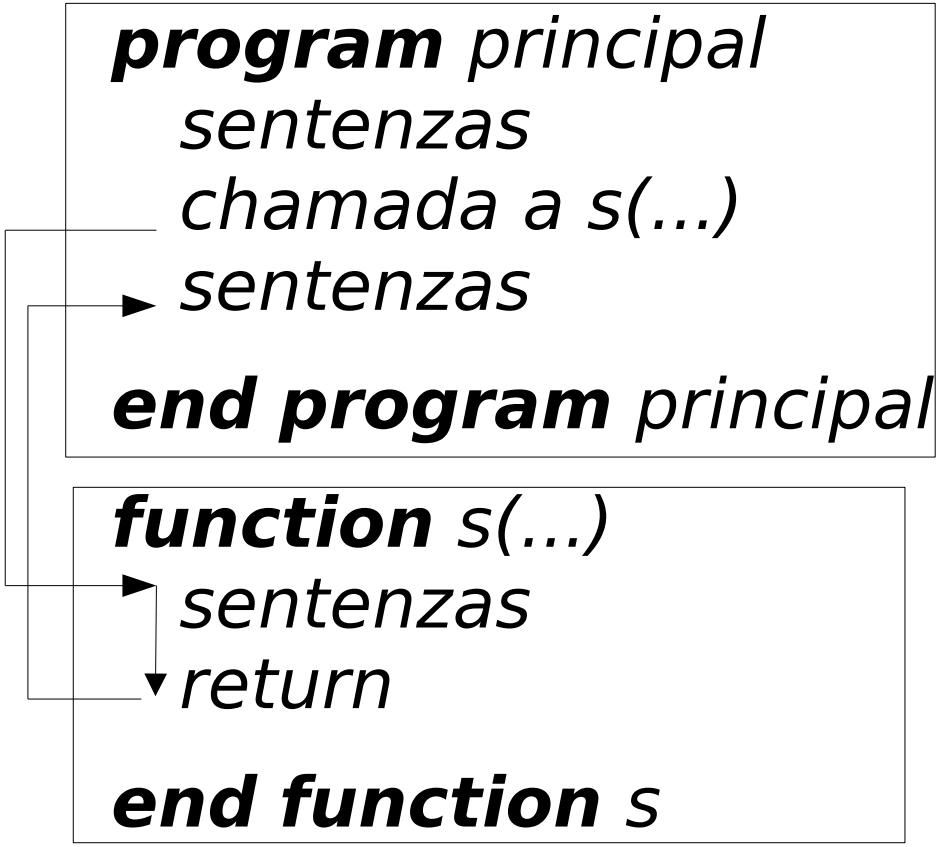
- ENIAC: Electronic Numerical Integrator And Computer
- Primeiro ordenador de propósito xeral (1946-1955)
- Elas desenvolveron as bases da programación de ordenadores

Subprogramas (I)

- Subprograma: conxunto de sentenzas que realizan unha tarefa clara, cunhas entradas (datos) e saídas (resultados) ben definidas. Exemplos:
  - subprograma que recibe un vector e calcula a súa media aritmética
  - subprograma que recibe unha matriz e calcula o seu determinante
- Hai dúas cousas:
  - Chamada ao subprograma dende o programa principal (*program*)
  - Corpo do subprograma: sentenzas do mesmo
- Argumentos: entradas e saídas do subprograma
- Poden estar no mesmo arquivo *.f90* ou en arquivos distintos

Subprogramas (II)

- O corpo do subprograma debe estar fóra do programa principal
- A chamada estará no programa principal (subprograma chamador)
- O subprograma ten as súas propias variábeis, inaccesíbeis dende o programa principal (so pode acceder aos argumentos)
- O programa principal tampouco non pode acceder ás variábeis do subprograma
- Dous tipos principais de subprogramas: **subrutinas** (non retornan valores) e **funcións externas** (retornan valores)



Sentenza *return* (opcional): retorna ao programa chamador (pode haber varias *return* no mesmo subprograma)

Tipos de subprogramas

Principais

- **Subrutina**: non retorna ningún valor, so ten argumentos *in-out-inout*
- **Función externa**: retorna un único valor (*integer*, *real*, *complex*, *double*, *character*, vector ou matriz), ten argumentos *in-out-inout*
- **Subprograma interno** (función ou subrutina): en programa principal, logo de *contains*. So se pode chamar dende o subprograma onde se define
- **Función de sentenza**: ten unha única sentenza, so se pode chamar dende o subprograma onde se define.

Cando usar funcións e cando subrutinas?

- **Función**: cando o subprograma so ten que calcular un único resultado (enteiro, real, complexo, lóxico ou carácter). A función retorna este resultado, almacenado na variábel co nome da función ou na variábel indicada no *result(...)*.

- **Subrutina**: cando o subprograma ten que calcular máis dun resultado, ou un único resultado pero éste é vector ou matriz. A subrutina debe ter algún argumento de saída ou entrada/saída (ver páxina seguinte) no que almacenar os resultados.

Argumentos (I)

- Tipo dos argumentos: *integer*, *real,* vector, matriz...
  - *intent(in)*: só lectura, o subprograma non pode modificalo (entrada): hai que inicializalos antes da chamada ao subprograma
  - *intent(out)*: non se pode ler (o chamador non lle dou ningún valor), só escribir nel (saída)
    - Non se pode inicializar antes da chamada
    - Logo de chamar ao subprograma hai que usa-lo seu valor
  - *intent(inout)*: o subprograma pode ler o seu valor e tamén modificalo (entrada/saída)

Argumentos (II)

- Os argumentos poden ser constantes, variábeis ou expresións
- Cando se pasa un argumento constante, variábel, ou expresión, hai que ter en conta o que espera o subprograma (*intent in*, *out* ou *inout*):
  - Se espera un argumento *out* ou *inout*, na chamada hai que pasarlle unha **variábel**: non constante ou expresión (daría un erro de compilación), porque vai ser modificado
  - Se o subprograma espera un argumento *in*, pódeselle pasar unha **constante, variábel ou expresión** (porque non vai modificarse)
- Os argumentos deben coincidir en número, tipo e orde na **chamada** e no **corpo do subprograma**, pero poden ter nomes distintos nos dous sitios
- Os argumentos *intent*(*in*) son constantes dentro do subprograma, e poden ser usados como dimensión de arrais estáticos.

Subrutina

- Non retorna ningún valor ao chamador: as entradas e saídas son a través dos argumentos. Axeitadas cando hai múltiples saídas

```
subroutine nome(arg1, ..., argN)
   tipo1, intent(...) :: arg1
   tipoN, intent(...) :: argN
   declaracións
   sentenzas executábeis

end subroutine nome
```

Variábeis locais do subprograma

- Chamada á subrutina: ***call*** *nome(arg1, ..., argN)*

Función externa

- Retorna un único valor, do *tipo* indicado:

```
tipo function nome(arg1, ..., argN) result(var)
tipo1, intent(...) :: arg1
tipoN, intent(...) :: argN
declaracións
sentenzas executábeis
var=expresión    !valor retornado: var
end function nome
```

Variábeis locais do subprograma

- A función chámase dende unha sentenza de asignación:
  tipo :: nome
  var=nome(arg1, ..., argN)
- Se non leva *tipo*, retorna un valor do tipo implícito de *nome*
- Se falta ***result***(*var*), a función retorna a variábel *nome*

Subprogramas e programa principal en arquivos distintos

- Os subprogramas e programa principal poden ir no mesmo arquivo *.f90*, ou en arquivos distintos, normalmente un arquivo distinto para cada subprograma
- Podes compilar co comando: *f95 *.f90 -o executabel*
- Tamén podes compilar cada arquivo coa opción -*c*:

f95 -c principal.f90
f95 -c subprograma1.f90
...
f95 -c subprogramaN.f90

- Crea un arquivo .*o* para cada arquivo *.f90*: non se executa
- Para crear o executábel:

*f95 *.o -o executabel*

Paso de vectores como argumentos

- **Neste curso**: pasar o nome do arrai e as súas dimensións (*explicit-shape arrays*)
- Cursos posteirores: pasar o nome do arrai (*assumed-shape arrays*) sen as súas dimensións: 3 posibilidades:
  - Subprograma interno
  - Módulo
  - Interface

```
program proba
integer :: x(10)
call sub(x,10)
end program proba
```

```
subroutine sub(x,n)
integer,intent(out) :: x(n)
integer,intent(in) :: n
...
end subroutine sub
```

```
program proba
integer,allocatable :: x(:)
...
call sub(x)
...
contains
    subroutine sub(x)
    integer,intent(out) :: x(:)
    n=size(x)
    ...
    end subroutine sub
end program proba
```

```
module aux
contains
    subroutine sub(x)
    integer,intent(out) :: x(:)
    n=size(x)
    ...
    end subroutine sub
end module
```

```
program proba
use aux
integer,allocatable :: x(:)
...
call sub(x)
...
end program proba
```

```
program proba
interface
    subroutine sub(x)
        integer,intent(out) :: x(:)
    end subroutine
end interface
integer,allocatable :: x(:)
call sub(x)
end program proba
```

```
subroutine sub(x)
integer,intent(out) :: x(:)
n=size(x)
...
end subroutine sub
```

Paso de matrices como argumentos

- Tamano explícito (**neste curso**):
- Tamano asumido (cursos posteriores):
  - Subprograma interno
  - Módulo
  - Interface

```
program proba
integer,allocatable :: a(:,:)
y=fun(a,n,m)
end program proba
```

```
function fun(a,n,m)
integer,intent(...) :: a(n,m)
integer,intent(in) :: n,m
...
end function fun
```

```
program proba
integer,allocatable :: a(:,:)
...
y=fun(a)
...
contains
    function fun(a)
    integer,intent(...) :: a(:,:)
    nf=size(a,1);nc=size(a,2)
    ...
    end function fun
end program proba
```

```
program proba
interface
    function fun(a)
        integer,intent(...) :: a(:,:)
    end function
end interface
integer,allocatable :: a(:,:)
y=fun(a)
end program proba
```

```
function fun(a)
integer,intent(...) :: a(:,:)
nf=size(a,1);nc=size(a,2)
...
end function fun
```

```
module aux
contains
    function fun(a)
    integer,intent(...) :: a(:,:)
    nf=size(a,1);nc=size(a,2)
    ...
    end function fun
end module
```

```
program proba
use aux
integer,allocatable :: a(:,:)
...
y=fun(a)
...
end program proba
```

Interface

- Bloque no que se indican subprogramas (tipo, nome, argumentos, valores retornados).

interface
bloque subprograma1
...
bloque subprograma N
end ***interface***

```
interface
    function fun(x,y,n) result(z)
        integer,intent(in) :: x(n),y(n)
        real :: z
    end function fun
    subroutine sub(x,y)
        real,intent(in) :: x
        real,intent(out) :: x
    end subroutine sub
end interface
```

- Non son necesarias, agás que os subprogramas teñan cousas especiais:

  Retornar arrais dinámicos
  Argumentos arrai sen a súa dimensión
  Argumentos nomeados ou opcionais

Módulo

- Bloque de código que contén datos e subprogramas: ***use*** *modulo*

```
module nome
    tipo :: var
    interface
        tipo subprograma(...)
        ...
        end fipo
    end interface
contains:
    tipo subprograma(...)
    ...
    end tipo
end module
```

```
module proba
      integer :: x
contains
      subroutine sub(y)
            integer,intent(in) :: y
            print *,y
      end subroutine sub
end module
```

```
program modulo
use proba
call sub(5)
end program modulo
```

- Pode estar en arquivo distinto do programa principal
- Se está en arquivo distinto: *f95 modulo.f90 principal.f90*

antes do arquivo que o usa!

Exemplo de **subrutina**: ordeamento dun vector de números:

```
subroutine ordea_seleccion(x,n)
real, intent(inout) :: x(n)
integer,intent(in) :: n
do i = 1, n
    aux = x(i); k = i
    do j = i + 1, n
        if(x(j) < aux) then
            aux = x(j); k = j
        end if
    end do
    x(k) = x(i); x(i) = aux
end do
return
end subroutine ordea_seleccion
```

vector: argumento *inout* xa que se os seus elementos lense e tamén se modifican

```
real :: x(5) = (/5,2,4,3,1/)
call ordea_seleccion(x,5)
print *, “ordeado=”, x
```

chamada ao subprograma

Exemplo de **función externa** sen cambiar o tipo do valor devolto

```
program proba
real :: x(5)=(/1,2,3,4,5/)
t=f(x,5)
print *,t
end program proba
```

Programa principal

Chamada á función

```
 function f(x,n)
real,intent(in) :: x(n)
integer,intent(in) :: n
f=0
do i = 1, n
      f=f+i*x(i)
end do
end function f
```

Subprograma

$$f = \sum_{i=1}^{n} i\, x_i$$

o valor calculado debe retornarse almacenándoo na variábel co mesmo nome ca función

Exemplo de **función externa** cambiando o tipo e nome do valor devolto

Cambiando so o tipo do valor devolto

```
program proba
real :: x(5)=(/1,2,3,4,5/)
integer :: f  ! valor devolto enteiro
n=f(x,5)
print *,n
end program proba

integer function f(x,n)
real,intent(in) :: x(n)
integer,intent(in) :: n
f=0
do i = 1, n
     f=f+i*x(i)
end do
end function f
```

$$f = \sum_{i=1}^{n} i\, x_i$$

Cambiando o tipo e nome do valor devolto

```
program proba
real :: x(5)=(/1,2,3,4,5/)
integer :: f  ! valor devolto
enteiro
n=f(x,5)
print *,n
end program proba

integer function f(x,n) result(y)
real,intent(in) :: x(n)
integer,intent(in) :: n
y=sum([(i,i=1,n)]*x) !vectorizado
end function f
```

Subprograma que ten que calcular un vector ou matriz

- Pódese facer cunha subrutina ou cunha función, pero é máis fácil cunha subrutina, porque coa función necesitas unha **interface**
- Polo tanto, usa unha **subrutina** cun argumento *out* (ou *inout*) para o vector ou matriz

```
program exemplo_vector
integer,allocatable :: x(:)
read *,n
allocate(x(n))
call sub(x,n)
print *,'x=',x
deallocate(x)
end program exemplo_vector
!----------------------------------------
subroutine sub(x,n)
integer,intent(out) :: x(n)
integer,intent(in) :: n
do i=1,n
      x(i)=i**2
end do
end subroutine sub
```

```
program exemplo_matriz
integer :: a(2,3)
call sub(a,2,3)
print *,'a='
do i=1,2
      print *,(a(i,j),j=1,3)
end do
end program exemplo_matriz
!----------------------------------------
subroutine sub(a,n,m)
integer,intent(out) :: a(n,m)
integer,intent(in) :: n,m
forall(i=1:2;j=1:3) a(i,j)=i**2+j**3
end subroutine sub
```

Subrutina que reserva un vector ou matriz *intent*(*out*) cunha interface

Con vector

```
program exemplo
interface
      subroutine sub(x)
         integer,allocatable,intent(out) :: x(:)
      end subroutine sub
end interface
integer,allocatable :: x(:)
call sub(x)
print *,'x=',(x(i),i=1,size(x))
deallocate(x)
end program exemplo
!-----------------------------------------
subroutine sub(x)
integer,allocatable,intent(out) :: x(:)
allocate(x(5))
do i=1,5
      x(i)=i*i
end do
return
end subroutine sub
```

Con matriz

```
program exemplo
interface
      subroutine sub(a)
            integer,allocatable,intent(out) :: a(:,:)
      end subroutine sub
end interface
integer,allocatable :: a(:,:)
call sub(a)
do i=1,size(a,1)
      print *,(a(i,j),j=1,size(a,2))
end do
deallocate(a)
end program exemplo
!-----------------------------------------
subroutine sub(a)
integer,allocatable,intent(out) :: a(:,:)
allocate(a(3,3))
do i=1,3
      do j=1,3
            a(i,j)=i*j+i-j
      end do
end do
return
end subroutine sub
```

Función externa que retorna un vector ou matriz reservado dinámicamente cunha interface

Con vector

```
program exemplo
interface
      function func(x) result(y)
         integer,intent(in) :: x(:)
         integer,allocatable :: y(:)
      end function func
end interface
integer :: x(5)=(/1,2,3,4,5/)
integer,allocatable :: y(:)
print *,'x=',x
y=func(x)
print *,'y=',y
deallocate(y)
end program exemplo
!-----------------------------------------
function func(x) result(y)
integer,intent(in) :: x(:)
integer,allocatable :: y(:)
n=size(x);allocate(y(n))
do i=1,n
      y(i)=x(n-i+1)
end do
end function func
```

Con matriz

```
program exemplo
interface
      function func(a) result(b)
            integer,intent(in) :: a(:,:)
            integer,allocatable :: b(:,:)
      end function func
end interface
integer :: a(5,5)
integer,allocatable :: b(:,:)
b=func(a)
deallocate(b)
end program exemplo
!-----------------------------------------
function func(a) result(b)
integer,intent(in) :: a(:,:)
integer,allocatable :: b(:,:)
nf=size(a,1);nc=size(a,2)
allocate(b(nf,nc))
do i=1,nf
      do j=1,nc
            b(i,j)=a(nf-i+1,nc-j+1)
      end do
end do
end function func
```

Exemplo de función que retorna un vector dinámico: *find*

- Atopa os índices dos elementos dun vector que cumpren unha condición (expresión lóxica)
- Moi útil para vectorizar expresións
- Retorna un vector reservado dinámicamente na función

```
function find(x) result(y)
logical,intent(in) :: x(:)
integer,allocatable :: y(:)
n=count(x);m=size(x)
allocate(y(n));j=1
do i=1,m
      if(x(i)) then
            y(j)=i;j=j+1
      end if
end do
end function find
```

```
program proba
interface
      function find(x) result(y)
          logical,intent(in) :: x(:)
           integer,allocatable :: y(:)
      end function find
end interface
integer :: x(5)=(/1,2,3,4,5/)
print *,count(x>2) !nº elementos >2
print *,find(x>2)    ! índices de elementos >2
print *,pack([(i=1,5)],x>2) !alternativa
print *,x(find(x>2))  ! valores de elementos >2
print *,pack(x,x>2) !alternativa
end program proba
```

Paso de subprogramas como argumentos

- Subprograma *external*: pode pasarse como argumento doutro subprograma. Ex: integral definida

```
program integral_definida
f(x)=sin(x); h(x)=cos(x)
real :: integral
external :: f, h
print *, integral(f, 0., 1.)
print *, integral(h, -1., 1.)
end program integral_definida
```

$$\int_0^1 \sin x \, dx$$

$$\int_{-1}^1 \cos x \, dx$$

```
real function integral(g,a,b)
real, intent(in) :: a, b
integral=0;x=a; h = 0.001
do
    integral=integral+g(x)
    x = x + h
    if(x > b) exit
end do
integral=integral*h
end function integral
```

Variábeis estáticas

- Son variábeis locais dos subprogramas que conservan o seu valor entre chamadas sucesivas a un subprograma
- Son estáticas porque se almacenan na mesma posición de memoria en tódalas chamadas ao subprograma
- As variábeis locais por defecto non son estáticas
- Para ser estática, dúas opcións alternativas:
  - Inicialización na declaración: *real :: x = 5*
  - Atributo *save: real, save :: x*
- Na 1ª chamada ao subprograma, o seu valor é 0 agás que se inicialice na declaración: *real :: x = 1*

Exemplo de uso das variábeis estáticas

```
program exemplo
do i=1,10
    call s()
end do
end program exemplo
```

```
subroutine s()
integer :: n=0
print *, “n=”, n
n=n+1
end subroutine s
```

Subprograma que mostra por pantalla o nº de veces que se leva executado usando unha variábel estática

Variábel local estática por inicializarse na declaración

Execución: imprime por pantalla os números do 0 ao 9 dende o subprograma (sen que se lle pasen argumentos)

Atención: se inicializamos na declaración unha variábel local dun subprograma:
• Pasa a ser **estática** (e conservar o seu valor entre chamadas a subprograma)
• A inicialización só vale para a 1ª chamada ao subprograma

Función de sentenza

- É unha función que so ten unha sentenza cunha expresión matemática.
- Sintaxe: *nome(arg1,...argN)=expresion*
- Exemplo:

```
f(x)=x**2+x-2
print *,x,y,f(x),f(y)
```

```
integer :: f
f(x)=x**2+x-2
print *,x,f(x)
```

- O seu resultado é do tipo definido implícitamente polo seu nome. Para cambialo:
- So se pode usar no subprograma no que se declara.
- Unha función de sentenza pode ser usada na definición doutra función de sentenza:

```
length(r)=2*pi*r
area(r)=pi*r*r
key(r)=length(r)*area(r)
```

Recursividade (I)

- Subprograma recursivo: subprograma que se chama a si mesmo. Pode ser función ou subrutina
- Debe ter o atributo *recursive*: se non, erro de compilación
- **Funcións recursivas**:

```
recursive function nome(args) result(var)
   x = nome(...)   ! chamada recursiva
   var = ...       ! valor retornado
```

- O nome da función non pode ser retornado, porque debe ser usado para chamarse a si mesma
- A función debe chamarse a si mesma nalgún lugar do corpo

Recursividade (II)

- **Subrutinas recursivas**:

  recursive subroutine nome(args)

  call nome(...)

- Dentro da subrutina debe chamarse a si mesma
- Subprogramas recursivos: debe haber sentenza(s) de selección para distinguir entre:
  - Caso recursivo (hai chamada recursiva)
  - Caso non recursivo (non hai chamada)
- Se non hai caso non recursivo, secuencia infinita de chamadas: *stack overflow* (erro de execución)

Exemplo de función recursiva: factorial dun número enteiro

```
program exemplo
integer :: factorial
fn = factorial(5)
end program exemplo


recursive integer function factorial(n) result(fn)
integer, intent(in) :: n
if(n <= 1) then
    fn = 1
else
    fn = n*factorial(n-1)
end if
end function factorial
```

$$n! = n(n-1)\cdot(n-2)\ldots 2\cdot 1$$

$$n! = n(n-1)!$$

Caso non recursivo

Caso recursivo

Implementación iterativa:

```
fn=1
do i=2,n
    fn=fn*i
end do
```

Vectorizada:

```
fn=product([(i,i=2,n])
```

Versión con subrutina recursiva

```
program exemplo
call factorial(5,fn)
print *,fn
stop
end program exemplo
!-------------------------------------------------------
recursive subroutine factorial(n,fn)
integer, intent(in) :: n
integer, intent(out) :: fn
if(n <= 1) then
    fn=1
else
    call factorial(n-1,m)
    fn=n*m
end if
end subroutine factorial
```

Argumento *out* para almacenar o resultado

Caso non recursivo

Caso recursivo

$$n! = n(n - 1)!$$

Argumentos nomeados

- Na chamada ao subprograma pódese especificar os nomes dos argumentos.
- Se na chamada se nomea un argumento, hai que nomealos todos.
- Pode haber chamadas con argumentos nomeados e outras sen nomealos.
- O subprograma debe declararse nun bloque *interface* no programa principal.

```
program argumentos_nomeados
interface
   subroutine nomeados(x,y)
      real,intent(in) :: x,y
   end subroutine
end interface
call nomeados(y=3.2,x=2.1)
call nomeados(1.1,2.2)
end program
argumentos_nomeados
!-----------------------------------------
subroutine nomeados(x,y)
real,intent(in) :: x,y
print *,'x=',x,'y=',y
end subroutine nomeados
```

Argumentos opcionais

- Teñen o atributo *optional*, de modo que se pode chamar ao subprograma indicando este argumento ou non.
- Non pode haber ningún argumento obrigatorio logo dun argumento opcional.
- O subprograma debe declararse nun bloque *interface* no programa principal.
- Función intrínseca *present(x)*: retorna *.true.* se o argumento *x* está presente, e *.false.* en caso contrario.

```
program argumentos_opcionais
interface
   subroutine opcionais(x)
      real,intent(in),optional :: x
   end subroutine
end interface
call opcionais(2.1)
call opcionais()
end program argumentos_opcionais
!-----------------------------------------------
subroutine opcionais(x)
real,intent(in),optional :: x
if(.not.present(x)) then
   print *,'argumento y non presente'
else
   print *,'argumento y presente',y
end if
end subroutine opcionais
```