7. Procedures and Structured Programming
external procedure:
separated small and reusable program units to conduct individual subtasks

Each program unit can be debugged and tested individually.

Two types of external procedure:
- subroutine
- function
specific tasks for this problem:
one main program or
several smaller sub-programs

loops to match
in time

input

initialization

update concentration

update temperature

update velocity

rhs of Poisson

solve Poisson eq.

1st diff.

filter

FFT

1st diff.

2nd diff.

FFT

1st diff.

2nd diff.

FFT

1st diff.

2nd diff.

FFT

1st diff.

2nd diff.

FFT

1st diff.

2nd diff.

FFT

matrix solver

1st diff.

2nd diff.

FFT

reuseable routines:
subrutines or
functions
Subroutines

Example:

```fortran
program abc
  implicit none
  real :: a, b, c, temp
  read(*,*) a, b
  temp = a**2 + b**2
  c = sqrt(temp)
  write(*,*) c
end program
```

```fortran
program abc
  implicit none
  real :: a, b, c
  read(*,*) a, b
  call hypotenuse(a, b, c)
  write(*,*) c
end program
```

```fortran
subroutine hypotenuse(a, b, c)
  real, intent(in) :: a, b
  real, intent(out) :: c
  real :: temp
  temp = a**2 + b**2
  c = sqrt(temp)
  return
end subroutine
```
• General form of a subroutine:

```plaintext
program main_program
...
call subroutine_name(argument_list)
...
end program
```

- **subroutine name**
  - ≤ 31 characters
  - to pass data between main program and subroutine

```plaintext
subroutine subroutine_name(argument_list)
description section
...
execution section
...
return
end subroutine [subroutine_name]
```
**INTENT attribute**

```fortran
subroutine hypotenuse(a,b,c,d)
real,intent(in) :: a, b
real,intent(out):: c
real,intent(inout) :: d

O c = sqrt(a**2 + b**2)
O c = a*b*c
X a = b
O d = a
O c = d
O d = c
```

Dummy argument *can* have an **INTENT** attribute associated with them to declare the type of the argument.

- **INTENT(in)**: argument is used only to pass data to subroutine, cannot be modified in subroutine
- **INTENT(out)**: argument is used only to return result to calling program
- **INTENT(inout)**: argument is used both to pass data to subroutine and return result to calling program
• Variable passing: pass-by-reference scheme

When the main program calls the subroutine, what is passed to the subroutine are the **pointers** to the memory locations containing the calling arguments. Not the actual variables.

```
program test
real :: a, b(4)
integer :: next
call sub1(a, b, next)
end program

subroutine sub1(x, y, i)
real, intent(out) :: x
real, dimension(4), intent(in) :: y
integer :: i
...
end subroutine
```

<table>
<thead>
<tr>
<th>main program</th>
<th>memory address</th>
<th>subroutine</th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
<td>001</td>
<td>x</td>
</tr>
<tr>
<td>b(1)</td>
<td>002</td>
<td>y(1)</td>
</tr>
<tr>
<td>b(2)</td>
<td>003</td>
<td>y(2)</td>
</tr>
<tr>
<td>b(3)</td>
<td>004</td>
<td>y(3)</td>
</tr>
<tr>
<td>b(4)</td>
<td>005</td>
<td>y(4)</td>
</tr>
<tr>
<td>next</td>
<td>006</td>
<td>i</td>
</tr>
</tbody>
</table>
• Array passing

**explicit-shape dummy array:**

```fortran
program test_array_pass
  implicit none
  real, allocatable :: data1(:), data2(:)
  integer :: n, nvals, i
!
read(*,*) n, nvals
allocate(data1(n), data2(n))
do i=1,nvals
  read(*,*) data1(i)
end do
!
call process(data1, data2, n, nvals)
...
end program

subroutine process(data1, data2, n, nvals)
  integer, intent(in) :: n, nvals
  real, intent(in) :: data1(n)
  real, intent(out), dimension(n) :: data2
!
do i=1, nvals
  data2(i)=3.*data1(i)
end do
end subroutine process
```
assumed-size dummy array (not recommended):

```fortran
subroutine process(data1, data2, nvals)
  real, intent(in) :: data1(*)
  real, intent(out), dimension(*) :: data2
  ...
```

Assumed-size dummy arrays are a holdover from earlier versions of Fortran. You should not use them in any new programs.
Example:
Read in a rank-2 array from a data file and calculate the sums of all the data in each row and of each column in the array. In the input data file, the first line contains two integers which specify the dimensions of the array. The elements in each row of the array appear on a line (i.e. a record) of the data file.

Input file:

```
2 4
-24.0  -1121.  812.1  11.1
35.6   8.1E3  135.23 -17.3
```

Output:

```
Sum of row  1=   -321.80
Sum of row  2=    8253.53
Sum of column  1=     11.60
Sum of column  2=   6979.00
Sum of column  3=    947.33
Sum of column  4=    -6.20
```
Example:

```fortran
program chap7_example
implicit none
character(len=20) :: infile  ! Input data file name
character(len=20) :: outfile  ! Output data file name
real, allocatable :: a(:, :)  ! Data array to sort
real, allocatable :: sum_row(:)  ! Sum of elements in a row
real, allocatable :: sum_col(:)  ! Sum of elements in a column
integer :: m, n
integer :: i, j

! Get the names of the input and output files
write(*,11)
11 format(1X,'Enter the filename to read the data: ')
read(*,'(A20)') infile
write(*,12)
12 format(1X,'Enter the filename to write the results: ')
read(*,'(A20)') outfile

! Open output data file
open(unit=2, file=outfile, status='replace', action='write')

! Open input data file and read the dimensions of the array
open(unit=1, file=infile, status='old', action='read')
read(1,*) m, n
```

Continued on next page...
allocate memory for matrix and read in the data
allocate(a(m,n))
doi = 1,m
  read(1,*) (a(i,j), j = 1,n)
end

! Sum elements in a row and output the results
allocate(sum_row(m))
call row_summation(m,n,a,sum_row)
doi = 1,m
  write(2,120) i, sum_row(i)
  120 format('Sum of row',i4,'=',f12.2)
end do
deallocate(sum_row)

! Sum elements in a column and output the results
allocate(sum_col(n))
call column_summation(m,n,a,sum_col)
doj = 1,n
  write(2,110) j, sum_col(j)
  110 format('Sum of column',i4,'=',f12.2)
end do
deallocate(sum_col)
end program
Subroutine to sum elements in rows of a matrix

subroutine row_summation(m,n,a,x)
  integer, intent(in) :: m, n
  real, intent(in) :: a(m,n)
  real, intent(out):: x(m)
  integer :: i, j
  
  do i = 1, m
    x(i) = 0.0
    do j = 1,n
      x(i) = x(i) + a(i,j)
    end do
  end do
  
  end subroutine

This subroutine takes a matrix a with dimensions m x n and computes the sum of elements in each row. The result is stored in the vector x of length m.
! subroutine to sum elements in columns of a matrix
subroutine column_summation(m,n,a,y)
integer, intent(in) :: m, n
real, intent(in) :: a(m,n)
real, intent(out):: y(n)
integer :: i, j
!
do j = 1, n
  y(j) = 0.0
  do i = 1,m
    y(j) = y(j) + a(i,j)
  end do
end do
!
return
end subroutine
Automatic Arrays

To create temporary working arrays in a subroutine

The automatic array will be created when entering the subroutine, and destroyed automatically when exiting.

```fortran
subroutine sub1(x,y,n,m)
integer, intent(in) :: n, m
real, intent(in) :: x(n,m)
real, intent(out):: y(n,m)
real :: wrk(n,m)
...
```

cf: allocatable array

```fortran
subroutine sub2(x,y,n,m)
real, allocatable :: wrk(:,:)
...
allocate wrk(n,m)
...
deallocate (wrk)
...
CPU_TIME subroutine

Syntax: CALL CPU_TIME(TIME)

Argument: TIME  real with INTENT(OUT)

- CPU_TIME returns a REAL value representing the elapsed CPU time in seconds.
- This is useful for testing segments of code to determine execution time.
- If a time source is available, time will be reported with microsecond \(10^{-6}\) s) resolution. If no time source is available, TIME is set to -1.0.
- TIME may contain a system dependent, arbitrary offset and may not start with 0.0. So the absolute value of TIME is meaningless, only differences between subsequent calls to this subroutine, as shown in the example below, should be used.

```fortran
program test_cpu_time
  real :: t_start, t_end
  call cpu_time(t_start)
  !
  task to be timed
  !
  call cpu_time(t_end)
  write(*,'("CPU time = ",f6.3," seconds.")') t_end-t_start
end program test_cpu_time
```
Example:

```fortran
program test_cpu_time
  implicit none
  real :: start, finish, x
  integer :: i

  !
  call cpu_time(start)
  ! put code to test here
  do i = 1, 999999999
    x = 1.
  end do
  call cpu_time(finish)

  !
  write(*,*) "start at ", start, " second"
  write(*,*) "finish at ", finish, " second"
  write(*,'(" CPU time = ",f6.3," seconds")')
  finish - start
end program test_cpu_time
```

Output:

```
start at    0.0000000     second
finish at    2.4648149     second
CPU time =  2.465 seconds
```
Functions

- **intrinsic function:**
  
e.g. \( a = \sin(x) \), \( b = \text{abs}(y) \)

- **user-defined function (function subprogram)**

```plaintext
program main_program_name
declaration statements
  ...
variable = func_name(argument_list)
  ...
end program

function func_name(argument_list)
declaration statements
  ...
execution statements
  ...
func_name = expression
return
end function [func_name]
```
Example:

a user-defined function named \texttt{quadf} to evaluate $ax^2 + bx + c$

```plaintext
program test_quad_func
implicit none
real :: quad_func
real :: a, b, c, x, error
!
read(*,*) a, b, c
read(*,*), x
quad = a*x**2 + b*x + c
error = abs(quad - quad_func(x,a,b,c))
write(*,*), error
end program

function quad_func(x,a,b,c)
real :: quad_func
real, intent(in) :: x, a, b, c
!
quad_func = a*x**2 + b*x + c
return
end function
```
Exercise

Subroutine of matrix multiplication

Rewrite the program of previous exercise (Matrix Multiplication) by calling a subroutine which multiplies two matrices passing from the main program and return the resulting matrix to the main program.