中国科学院计算机网络信息中心
Computer Network Information Center,
Chinese Academy of Sciences

Fortran Programming

Presenter: Shaobo Tian

Date: 2026. 01. 27

CONTENTS

1. Introduction

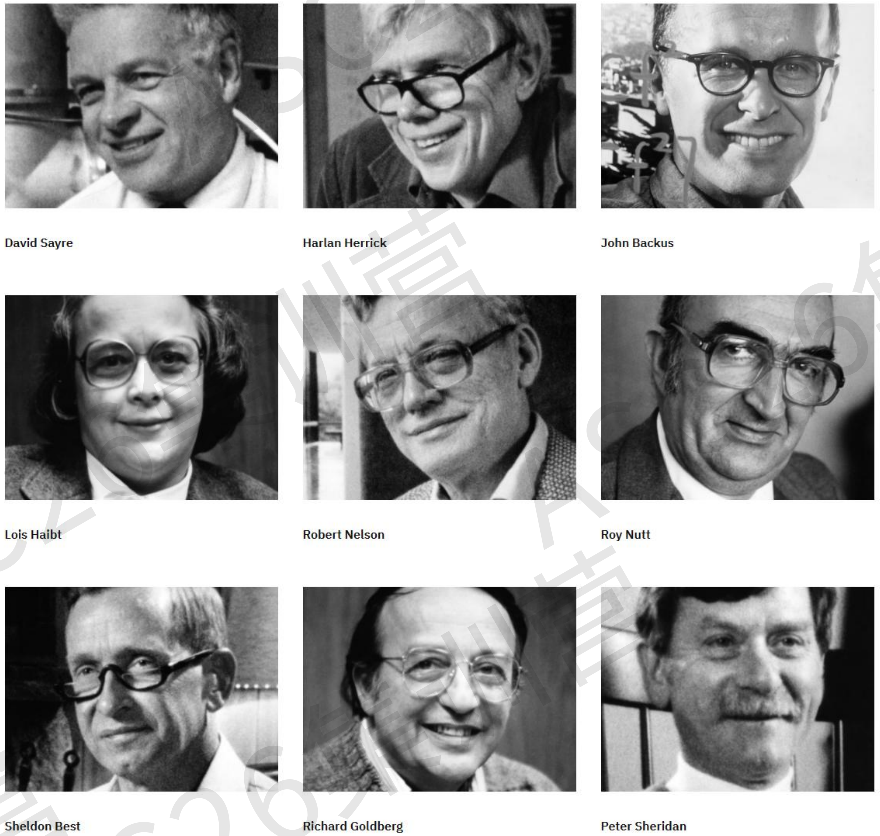
中国科学院计算机网络信息中心
Computer Network Information Center, Chinese Academy of Sciences

Fortran (Formula Translation)

In the early 1950s, computer programming was the exclusive domain of a small group of specialists who wrote code in machine language, a complex and cumbersome set of instructions. **Programming was for experts** only — outsiders need not apply. Then came Fortran.

From its creation in **1954** and its commercial release in 1957 as the progenitor of software, Fortran (short for *formula translation*) became the first computer language standard. It helped open the door to modern computing and ranks as one of the most influential software products in history. Fortran liberated computers from the exclusive realm of programmers and opened them to nearly everybody else. And it's still in use decades after its release [1].

Pioneers of Fortran



[1] https://www.ibm.com/history/fortran#Born+of+necessity

1. Introduction

中国科学院计算机网络信息中心
Computer Network Information Center, Chinese Academy of Sciences

Fortran (Formula Translation)

John Backus, the father of Fortran, released the first Fortran compiler at IBM, creating the world's first high-level programming language—predating the arrival of C in 1972.

Fortran was specifically designed for **computation-intensive applications** in science and engineering, and its strength lies in its ability to translate complex scientific formulas into computer code. The language was originally created to make scientific computing more efficient and straightforward.

The feature of Fortran language :

- Easy to learn, with rigorous grammar.
- It can directly perform operations on matrices and complex numbers.

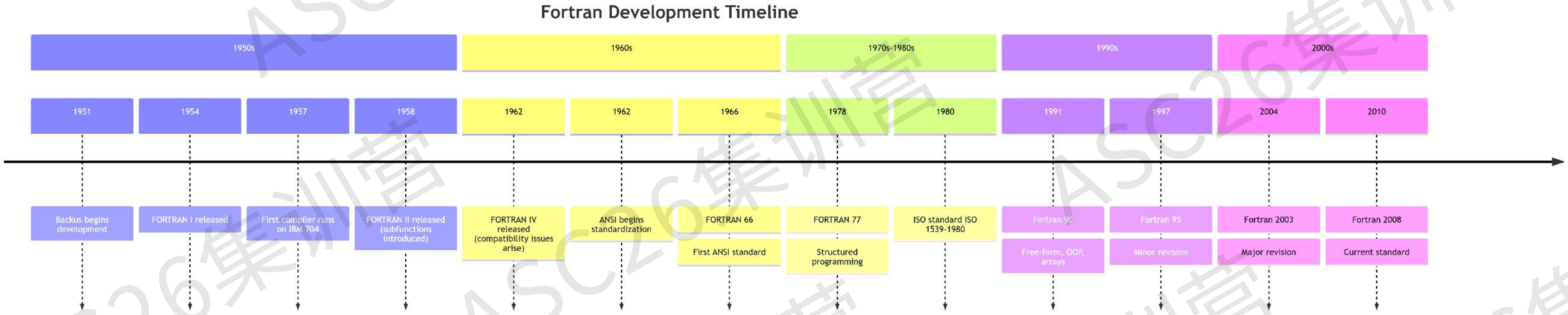
TIOBE index[1]

Jan 2026	Jan 2025	Change	Programming Language	Ratings	Change
1	1		Python	22.61%	-0.68%
2	4	↑	C	10.99%	+2.13%
3	3		Java	8.71%	-1.44%
4	2	↓	C++	8.67%	-1.62%
5	5		C#	7.39%	+2.94%
6	6		JavaScript	3.03%	-1.17%
7	9	↑	Visual Basic	2.41%	+0.04%
8	8		SQL	2.27%	-0.14%
9	11	↑	Delphi/Object Pascal	1.98%	+0.19%
10	18	⇈	R	1.82%	+0.81%
11	32	⇈	Perl	1.63%	+1.14%
12	10	↓	Fortran	1.61%	-0.42%
13	14	↑	Rust	1.51%	+0.34%
14	15	↑	MATLAB	1.40%	+0.34%
15	13	↓	PHP	1.38%	-0.00%
16	7	⇊	Go	1.24%	-1.37%
17	12	⇊	Scratch	1.24%	-0.31%
18	26	⇈	Ada	1.19%	+0.54%
19	17	↓	Assembly language	1.07%	+0.05%
20	25	⇈	Kotlin	0.97%	+0.23%

[1] https://www.tiobe.com/tiobe-index/
[2] https://amturing.acm.org/award_winners/backus_0703524.cfm

1. Introduction

中国科学院计算机网络信息中心
Computer Network Information Center,
Chinese Academy of Sciences



- Original versions, Fortran I, II and III are considered obsolete now.
- Oldest version still in use is Fortran IV, and Fortran 66.
- Most commonly used versions today are : Fortran 77, Fortran 90, and Fortran 95.
- Fortran 77 added strings as a distinct type.
- Fortran 90 added various sorts of threading, and direct array processing.
- Fortran 2003,Fortran 2008, Fortran 2023

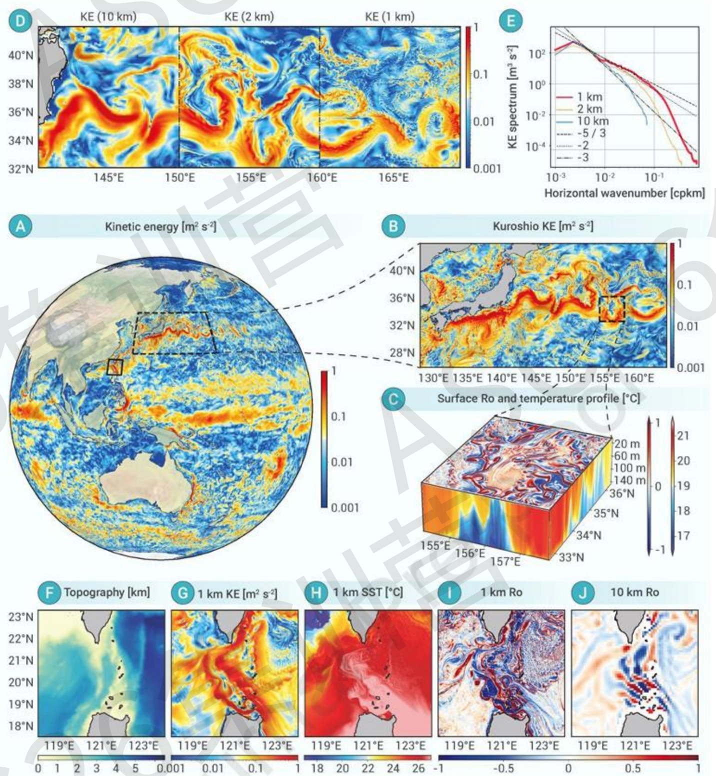
[1] extension://nhppiemcomgngbgdeffdgkhnkjlgpcdi/data/pdf.js/web/viewer.html?file=https://math.ecnu.edu.cn/~jypan/Teaching/Fortran/Fortran95pjy.pdf
[2] https://gcc.gnu.org/fortran/
[3] https://www.intel.cn/content/www/cn/zh/developer/tools/oneapi/fortran-compiler.html
[4] https://www.nvidia.cn/about-nvidia/press-releases/2014/nvidia-pgi-ibm-power-systems-11202014/
[5] https://www.tutorialspoint.com/fortran/fortran_overview.htm
[6] https://www.intel.cn/content/www/cn/zh/developer/articles/release-notes/fortran-compiler/2023.html

1. Introduction

Why Fortran?

Application

- Numerical analysis and scientific computation
- Structured programming
- Array programming
- Modular programming
- Generic programming
- High performance computing on supercomputers
- Object oriented programming
- Concurrent programming
- Reasonable degree of portability between computer systems



[1] https://www.tutorialspoint.com/fortran/fortran_overview.htm
[2] Xie, J., Yu, J., Zhou, Y., Liu, H., Wei, J., Han, X., Xu, K., Yu, M., Yu, Z., Lin, P., Jiang, J., Zheng, W., Zhang, T., Wang, R., Jing, Z., Wu, L., A 1-km resolution global ocean simulation promises to unveil oceanic multi-scale dynamics and climate impacts, The Innovation (2025), doi: https://doi.org/10.1016/j.xinn.2025.100843

2. Example

中国科学院计算机网络信息中心
Computer Network Information Center, Chinese Academy of Sciences

Case One

```
program hello
    implicit none
    print *, "Hello, Fortran World!"
end program hello
```

Compilation: **gfortran hello.f90 -o hello**
Run: **./hello**

```
Hello, Fortran World!
```

program： Tells the compiler that the main program unit starts here and is named "hello"; it plays the same role as int main() in C.

hello： The Program Name

implicit none: Switches off Fortran's default "implicit typing" rule; every variable must be explicitly declared, or the code will not compile.

print * : The command displays data on the screen.

end program: Marks the end of the main program and pairs with the opening program hello.

[1] https://fortran-lang.org/zh_CN/learn/quickstart/derived_types/
[2] https://www.w3ccoo.com/fortran/fortran_basic_input_output.html

2. Example

- **Case Two**

```
program calculate
  implicit none
  real :: a, b, sum, product
  a = 5.0
  b = 3.0
  sum = a + b
  product = a * b
  print *, 'a + b = ', sum
  print *, 'a * b = ', product
end program calculate
```

2. Example

中国科学院计算机网络信息中心
Computer Network Information Center, Chinese Academy of Sciences

Case Three

```
program character
  implicit none
  character(len=4) :: s1, s2
  s1 = "Fort"
  s2 = "fort"
  print *, 's1 == s2 ', s1 == s2   !  False
  print *, 's1 /= s2 ', s1 /=  s2   !  True
end program character
```

```
s1 == s2   F
s1 /= s2   T
```

"!" represents the comment code

Use the relational operator "**/=** " between the operands, e.g. a /= b means "a is not equal to b".

"== " means "equal".

Fortran allows both uppercase and lowercase letters.

It is case-insensitive except for string literals.

2. Example

➢ Detail

```
program calculate
  implicit none
  real :: a, b, sum, product
  a = 5.0
  b = 3.0
  sum = a + b
  product = a * b
  print *, 'a + b = ", sum
  print *, 'a * b = ", product
end program calculate
```

- `program calculate` → **Program header:** program name
- `real :: a, b, sum, product` → **Declaration section**: variables, arrays, parameter declarations
- `a = 5.0` … `print *, 'a * b = ", product` → **Execution section**: computation, control, input/output
- `end program calculate` → **Program end:** end program

3. Variable

Basic Concepts & Naming Rules

Definition:

A variable is the name of a storage area that a program can manipulate, usually declared with its type and name before use.

Naming Rules:

- Maximum length: 31 characters（**Fortran 90/95**，**Recommended**）、63 characters (**Fortran 2003**)
- May contain only alphanumeric characters (A~Z, a~z, 0~9) and the underscore (_)
- First character must be a letter
- Letters are case-insensitive, except when used as strings

3. Variable

Basic Data type

中国科学院计算机网络信息中心
Computer Network Information Center,
Chinese Academy of Sciences

Data Types

The basic data types mainly include: INTEGER, REAL, COMPLEX, CHARACTER, and LOGICAL.

Data Type	Optional	Kind Selector	Example & Byte Size	
Integer	INTEGER	NO	INTEGER :: m	4 bytes
	INTEGER(k)	k = { 1, 2, 4, 8 }	INTEGER(2) :: n	Short integer
Real	REAL	NO	REAL :: m	4 bytes
	REAL(k)	k = { 4, 8, 16 }	REAL(8) ::n	8 bytes
	DOUBLE PRECISION	NO	DOUBLE PRECISION ::x	8 bytes
Character	CHARACTER	NO	CHARACTER :: str	1 byte
Logical	LOGICAL	NO	LOGICAL :: flag	4 bytes
	LOGICAL(k)	k = { 1, 2, 4, 8 }	LOGICAL(1) :: flag	1 byte
Complex	COMPLEX	NO	COMPLEX :: m	8 bytes
	COMPLEX(k)	k = { 4, 8, 16 }	COMPLEX(8) :: n	16bytes
	DOUBLE COMPLEX	NO	double complex :: x	16 bytes

3. Variable

Variable Declaration & Initialization

Declaration Syntax: data type [[, attribute]...] :: variable_list

Example1:

```
integer :: counter
real :: velocity_x
character(len=10) :: student_name
counter=1
velocity=1.0
student_name = "ZhangSan"
```

Example2:

```
integer :: i=0, j=1, k=2
real :: x = 0.0, y = 1.0
complex :: z = (0.0, 0.0)
logical :: debug = .false.
character(len=20) :: msg = "Initial value"
```

Example3:

```
real, parameter :: PI = 3.141592653589793d0
real, parameter :: G = 9.81
integer, parameter :: MAX_ITER = 1000
```

→

Values unchanged during program execution

Advantage

- Improve code readability
- Avoid magic numbers
- Improve maintenance and modification

3. Variable

➢ Array Concept

Definition:

An array is a collection of elements of the same type stored in sequence with contiguous memory addresses.

Array: A(7)

A(1)	A (2)	A(3)	A(4)	A(5)	A(6)	A(7)

Array: B(3,7)

B(1,1)	B (1,2)	B(1,3)	B(1,4)	B(1,5)	B(1,6)	B(1,7)
B(2,1)	B (2,2)	B(2,3)	B(2,4)	B(2,5)	B(2,6)	B(2,7)
B(3,1)	B (3,2)	B(3,3)	B(3,4)	B(3,5)	B(3,6)	B(3,7)

↓ Direction of data contiguity

Note: In Fortran, the elements are stored **column-contiguous**, i.e., the leftmost subscript varies first in memory.

3. Variable

Array Variables Declaration

Static array:

- Size is fixed at compile time and cannot be changed while the program runs
- Declaration: dimensions are written as constants or parameters

Example:

```
real, dimension(10) :: vector
integer, dimension(5,5) :: matrix
integer, parameter :: n=6
real    :: b(n,n)        ! parameter constant is also allowed
```

Fast access

Dynamic array:

- Size is determined at run time and can be allocated, released, or resized

Example:

```
integer, allocatable :: c(:,:)
allocate(c(1000,1000))   ! allocate at run time
deallocate(c)            ! manual release
```

Save memory & Flexible size

3. Variable

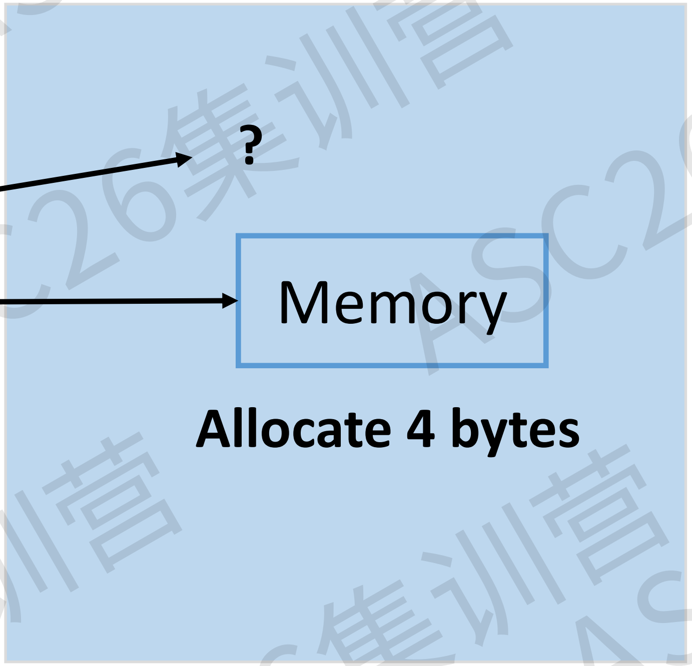
Array Variables Declaration

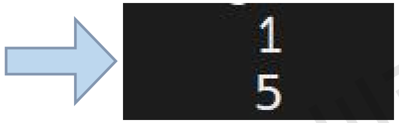
Defeinition:

- A pointer is a type of data object that not only stores memory addresses, but also contains more information about a specific object, such as type, level and range.
- A pointer is associated with a target through allocation or pointer assignment.

Example:

```
program pointerExample
implicit none
  integer, pointer :: p1
  allocate(p1)
  p1 = 1
  Print *, p1
  p1 = p1 + 4
  Print *, p1
end program pointerExample
```





3. Variable

Array Variables Declaration

Target And association:

- The target is another normal variable, leaving space for it. The target variable must use the **target attribute**.
- Use the association operator (=>) to associate pointer variables with target variables.

Example:

```
program pointerExample
implicit none
  integer, pointer :: p1
  integer, target :: t1
   Print *, p1
   Print *, t1
   p1 = p1 + 4
  Print *, p1
  Print *, t1
end program pointerExample
```

```
 -2110419640
           0

Program received signal SIGSEGV: Segmentation fault - invalid memory reference

Backtrace for this error:
#0  0x2b1ff2c09bda
#1  0x2b1ff2c08dc3
#2  0x2b1ff375527f
#3  0x4007c3
#4  0x4008bf
#5  0x2b1ff37413d4
#6  0x400638
#7  0xffffffffffffffff
Segmentation fault
```

3. Variable

中国科学院计算机网络信息中心
Computer Network Information Center, Chinese Academy of Sciences

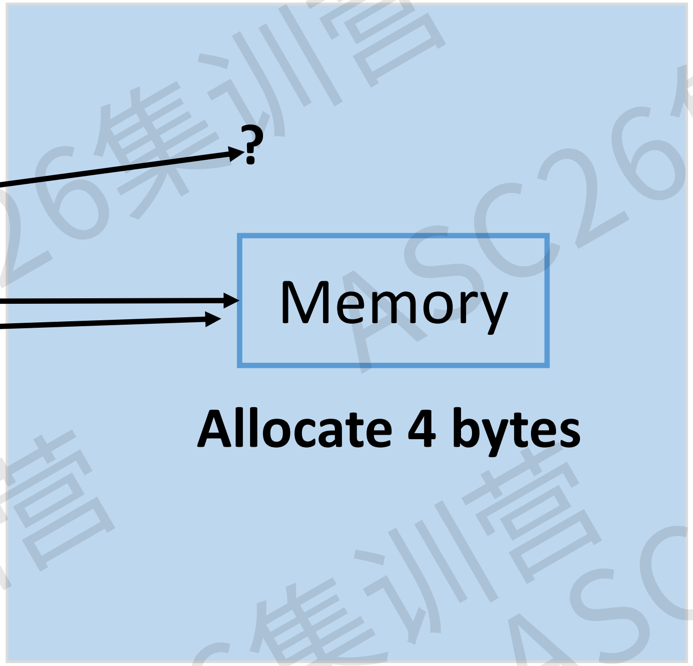
➢ Array Variables Declaration

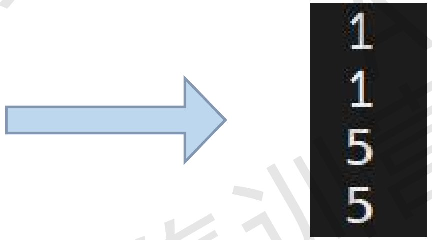
Target And association:

- The target is another normal variable, leaving space for it. The target variable must use the target attribute.
- **Use the association operator** (=>) to associate pointer variables with target variables.

Example:

```
program pointerExample
implicit none
  integer, pointer :: p1
  integer, target :: t1
  p1=>t1
  p1 = 1
  Print *, p1;   Print *, t1
  p1 = p1 + 4
  Print *, p1;   Print *, t1
end program pointerExample
```





3. Variable

➢ Derived Type (Struct)

Definition:

The derived type is a special form of data type that can encapsulate other intrinsic types as well as other derived types. It can be regarded as equivalent to the **struct** in C and C++ programming languages.

Example:

```
type :: person
  character(len=20) :: name
  integer :: age
  real :: height
end type person
```

An element

```
type(person) :: student
student%name = "ZhangSan"
student%age = 20
student%height = 1.75
```

Array

```
type(person), dimension(50) :: class
class(1)%name = "ZhangSan"
.......
class(2)%name = "LiSi"
class(1)%name = "ZhaoWu"
```

3. Variable

Local Variable and Global Variable

Local Variable:

- Declared inside a **program**, **function**, **subroutine**.
- Lifetime: created on entry to the procedure and destroyed on exit.
- Visible only within their own program unit; Same variable name in other units do not conflict.

Example:

```
! Inside a program
program main
  implicit none
  real :: a      ! visible only in main
  a = 5.0
print *, 'a = ", a
end program main
```

```
! Inside a function
real function square(a)
  real, intent(in) :: a
  real :: tmp      ! visible only in square
  tmp = a*a
  square = tmp
end function square
```

```
! Inside a subroutine
subroutine Show()
  character(len=5) :: msg =
"Hello"  ! visible only in Show
  print *, msg
end subroutine Show
```

3. Variable

➢ Local Variable and Global Variable

Global Variable:

- Declare the variables inside a **module** and mark them as **public**
- Any unit (module/function/subroutine) that uses the module can access them, and their lifetime equals that of the program.

Example:

```
module math_const
implicit none
real, parameter :: PI  =  3.14159265
real, parameter :: E    = 2.71828182
module math_const
```

```
module timers
implicit none
integer :: count = 0
contains
  subroutine tick()
    count = count + 1
    print *, "Timer tick =", count
  end subroutine tick
end module timers
```

```
program main
use timers ! Import  count and tick
use math_const, only: PI   ! Import only the constants needed
implicit none
print *, "PI  =", PI
call tick()
call tick()
print *, "Final count =", count
end program main
```

4. Operator

Concept

Definition:

"Operator" is a symbol that tells the compiler to perform a specific **mathematical or logical** operation.

Three fundamental control structure:

- Arithmetic operator
- Relational operator
- Logical operator

4. Operator

Arithmetic operator

Assume variable A is 5 and variable B is 2.

Operator	Description	Example
+	**Addition**: adds the two operands.	A + B = 7
–	**Subtraction**: subtracts the second operand from the first.	A – B = 3
*	**Multiplication**: multiplies the two operands.	A * B = 10
/	**Division**: divides the numerator by the denominator.	A / B = 2
**	**Exponentiation**: raises the first operand to the power of the second.	A ** B = 25

4. Operator

➢ Relational operator

Assume variable A is 5 and variable B is 2.

Operator	equivalent	Description	Result
==	.eq.	Tests equality	(A == B) → false
/=	.ne.	Tests inequality	(A /= B) → true
>	.gt.	Tests "greater than"	(A > B) → true
<	.lt.	Tests "less than"	(A < B) → false
>=	.ge.	Tests "greater than or equal"	(A >= B) → true
<=	.le.	Tests "less than or equal"	(A <= B) → false

4. Operator

中国科学院计算机网络信息中心
Computer Network Information Center, Chinese Academy of Sciences

➢ Logical operator

Assume variable A is true and variable B is false.

Operator	Description	Result
.and.	Logical AND: true only if both operands are non-zero.	(A .and. B) → false
.or.	Logical OR: true if either operand is non-zero.	(A .or. B) → true
.not.	Logical NOT: reverses the logical state.	.not.(A .and. B) → true
.eqv.	Logical equivalence: true if both values are equal.	(A .eqv. B) → false
.neqv.	Logical non-equivalence: true if values differ.	(A .neqv. B) → true

4. Operator

Operator Precedence

Operator precedence determines how terms are grouped in an expression. This affects the way the expression is evaluated. Some operators have higher precedence than others. For example, the multiplication operator has higher precedence than the addition operator.

Type	Operators	Associativity
Exponentiation	**	left → right
Multiplication & Division	* /	left → right
Addition & Subtraction	+ -	left → right
Relational	< <= > >=	left → right
Equality	== /=	left → right
Logical NOT /unary minus	.not. (-)	left → right
Logical AND	.and.	left → right
Logical OR	.or.	left → right
Assignment	=	right → left

5. Control Structure

中国科学院计算机网络信息中心
Computer Network Information Center, Chinese Academy of Sciences

Concept

Definition:

Control flow is the order that instructions are executed in a program. A control statement is a statement that determines control flow of a set of instructions.

Three fundamental control structure:

- Sequential control
- Selection control
- Iterative control

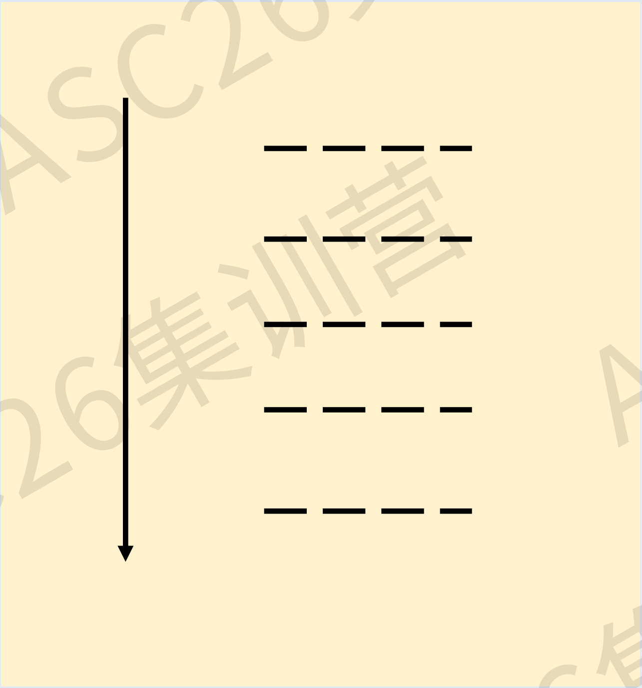
These three basic structures can be combined into complex programs to solve various problems.

5. Control Structure

Sequential control

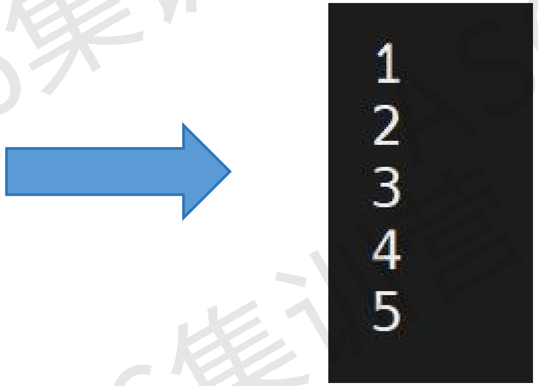
Definition:

Sequential control is the simplest control structure; statements are executed one after another in the order they are written. After each statement finishes, the program automatically proceeds to the next.



Example:

```
program sequential
    implicit none
    print *, '1'
    print *, '2'
    print *, '3'
    print *, '4'
    print *, ‘5’
 end program
```



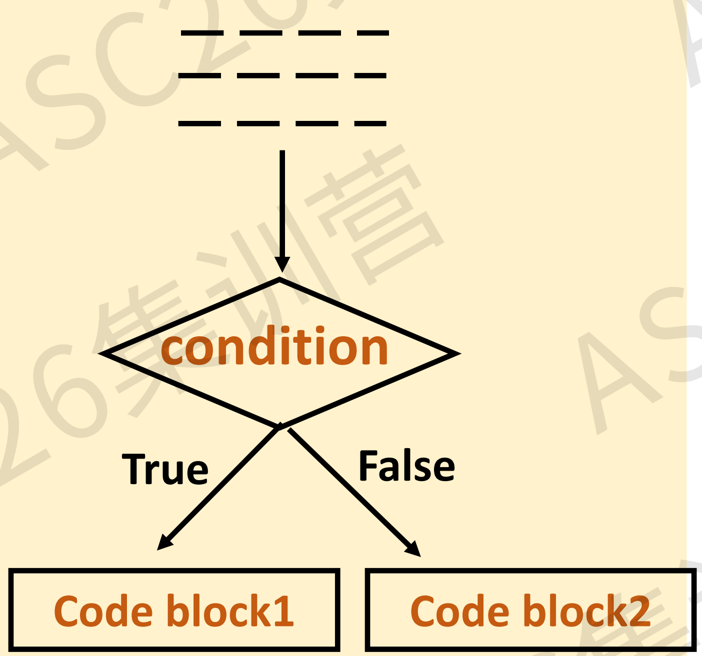
5. Control Structure

中国科学院计算机网络信息中心
Computer Network Information Center, Chinese Academy of Sciences

➢ Selection control

Definition:

Selection control decides which code segment to execute by evaluating whether a condition is true or false. It mainly includes if statements, if-then-else, if-else-if-else and select statements.



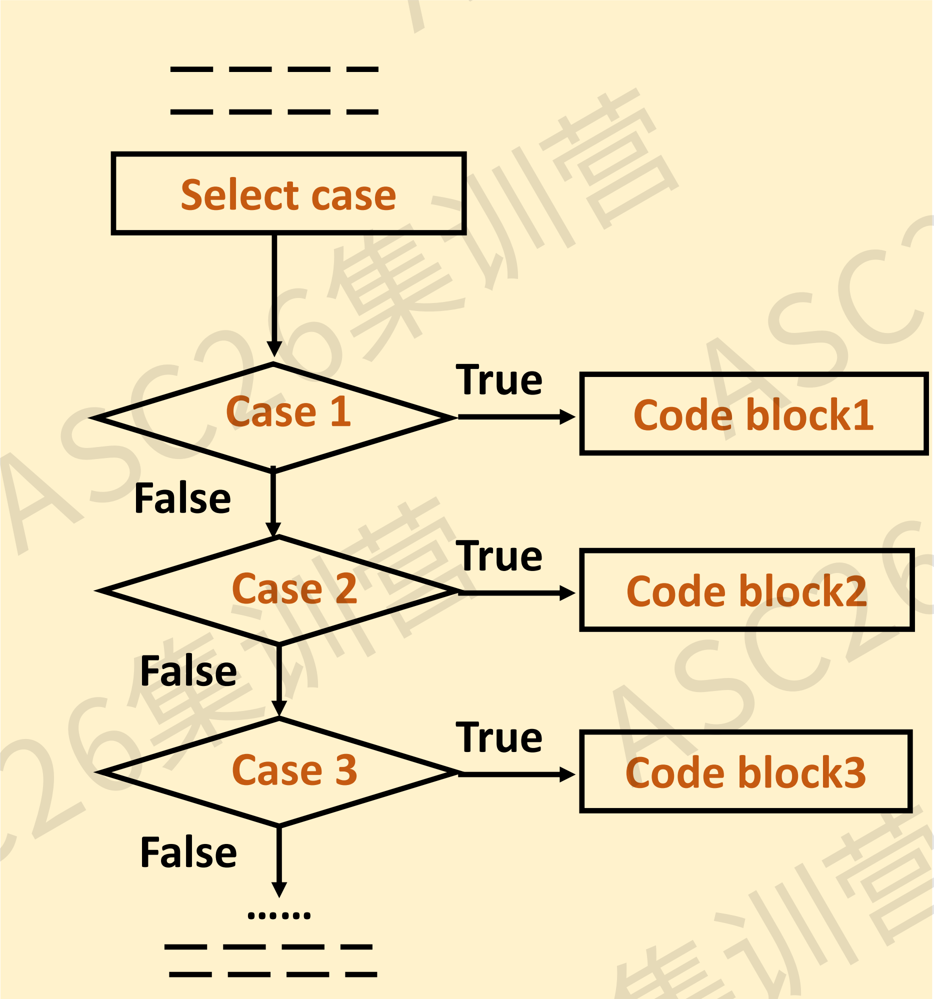
Example:

```
program ifProg
implicit none
  integer :: a = 10
  if (a < 20 ) then
    print*, "a is less than 20"
  end if
  if (a > 15 ) then
    print*, "a is more than 15"
  else
    print*, "a is less than 15"
  end if
  print*, "value of a is",a
end program ifProg
```

```
a is less than 20
a is less than 15
value of a is            10
```

5. Control Structure

➢ Selection control



Example:

```
program selectCaseProg
implicit none
 character :: grade = 'B'
   select case (grade)
    case ('A')
      print*, "Excellent!"
    case ('B')
       print*, "Well done"
    case ('C')
       print*, "You passed"
    case default
       print*, "Invalid grade"
  end select
  print*, "Your grade is ", grade
end program selectCaseProg
```

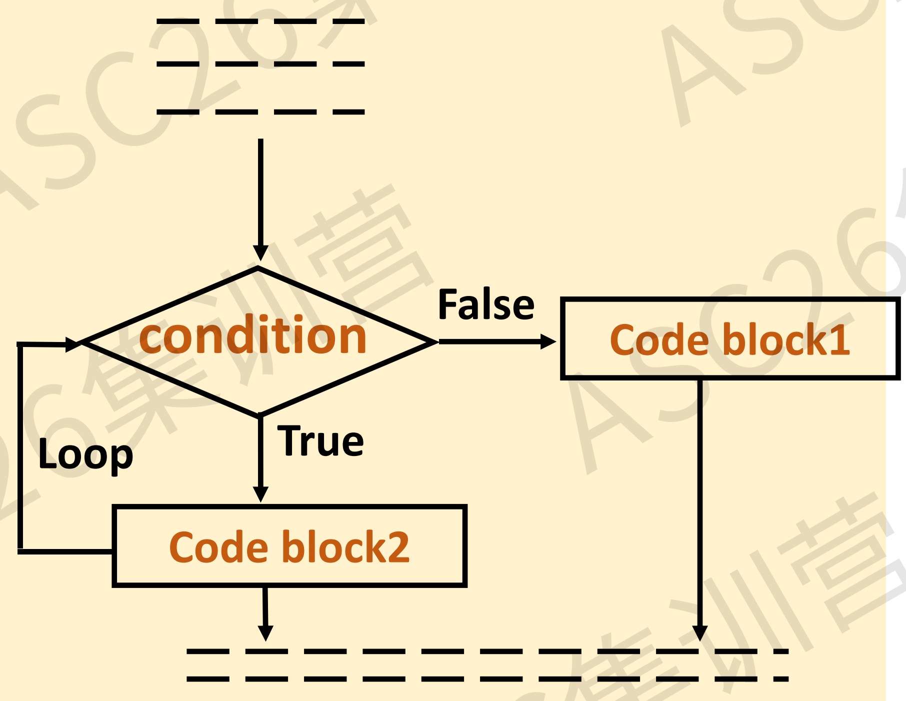
```
Well done
Your grade is B
```

5. Control Structure

➢ Iterative control

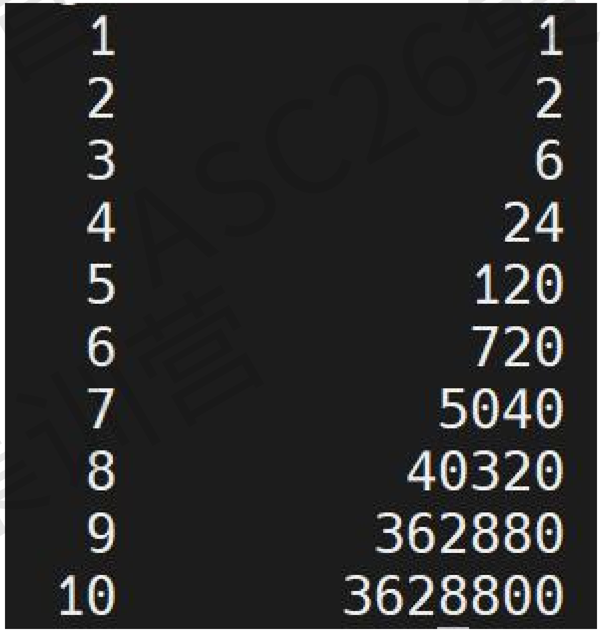
Definition:

Iterative control repeatedly executes a segment of code until the condition becomes false and the loop exits. It mainly includes for **do loops**, and do-while loops.



Example:

```
program factorial
 implicit none
 integer :: nfact = 1
  integer :: n
  ! compute factorials
  do n = 1, 10
     nfact = nfact * n
      print*,  n, " ", nfact
  end do
end program factorial
```

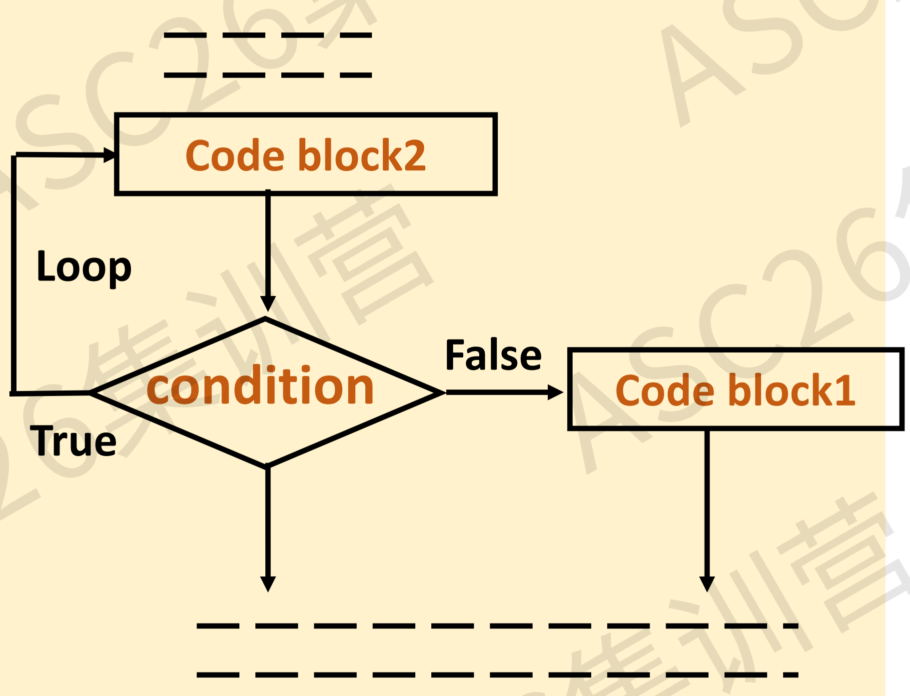


5. Control Structure

Iterative control

Definition:

Iterative control repeatedly executes a segment of code until the condition becomes false and the loop exits. It mainly includes for do loops, and **do-while** loops.



Example:

```
program factorial
 implicit none
 integer :: nfact = 1
  integer :: n = 1
  ! compute factorials
  do while (n <= 10)
     nfact = nfact * n
     print*,  n, " ", nfact
  end do
end program factorial
```

2	1
3	2
4	6
5	24
6	120
7	720
8	5040
9	40320
10	362880
11	3628800

5. Control Structure

➢ Iterative control

"**exit**": When "exit" is encountered, the current loop stops immediately and execution continues with the code that follows the loop. If the `exit` statement appears inside nested loops, it exits only the innermost loop.

"**cycle**" : The "cycle" keyword skips the remaining code in the current iteration and proceeds directly to the next loop condition.

Example:

```
program main
 implicit none
 integer :: i
  do i = 1, 100
    if (i > 50) exit
    if (mod(i,2) == 0) cycle
    print *, i
 end do
end program main
```

```
           1
           3
           5
           7
           9
          11
          13
          15
          17
          19
          21
          23
          25
          27
          29
          31
          33
          35
          37
          39
          41
          43
          45
          47
          49
```

Example:

```
program main
 implicit none
 integer :: i, n
 do n = 1, 3
  do i = 1, 10
    if (i > 8) exit
   print *, 'i=', i, 'n=', n
 end do
end do
end program main
```

```
i=           1 n=           1
i=           2 n=           1
i=           3 n=           1
i=           4 n=           1
i=           5 n=           1
i=           6 n=           1
i=           7 n=           1
i=           8 n=           1
i=           1 n=           2
i=           2 n=           2
i=           3 n=           2
i=           4 n=           2
i=           5 n=           2
i=           6 n=           2
i=           7 n=           2
i=           8 n=           2
i=           1 n=           3
i=           2 n=           3
i=           3 n=           3
i=           4 n=           3
i=           5 n=           3
i=           6 n=           3
i=           7 n=           3
i=           8 n=           3
```

5. Control Structure

Iterative control

Unconditional exit

EXIT [do-construct-name]

Rules

"do-construct-name" must be the exact name appearing on an enclosing "DO" statement.

Example:

```
program factorial
  implicit none
  integer :: i, n
  Outer: do n = 1, 3
    inner: do i = 1, 10
       if (i > 8) exit  Outer
     print *, 'i=', i, 'n=', n
   end do inner
  end do Outer
end program factorial
```

```
i =           1 n =           1
i =           2 n =           1
i =           3 n =           1
i =           4 n =           1
i =           5 n =           1
i =           6 n =           1
i =           7 n =           1
i =           8 n =           1
```

6. Procedure

Concept

Definition:

A procedure is a set of statements that performs a well-defined task and can be invoked from a program. Information (or data) is passed to the calling program as arguments to the procedure.

There are two types of procedure:

- Functions
- Subroutines

```
function name(arg1, arg2, ...)
    [declarations, including those for the arguments]
    [executable statements]
end function [name]
```

```
function name(arg1, arg2, ...) result (return_var_name)
    [declarations, including those for the arguments]
    [executable statements]
end function [name]
```

6. Procedure

➢ Example

```
Program calling_func
  implicit none
  real :: a
   a = area_of_circle(3.0)

   Print *, "The area of a circle with radius 3.0 is"
   Print *, a
end program calling_func
```

```
function area_of_circle (r)
implicit none
   ! dummy arguments
   real :: area_of_circle
   ! local variables
   real :: r
   real :: pi
   pi = 4 * atan (1.0)
   area_of_circle = pi * r**2
end function area_of_circle
```

```
The area of a circle with radius 3.0 is
  28.2743340
```

Note:

1.) You must specify implicit none in both the main program and the procedure.

2.) The argument r in the called function is referred to as a dummy argument.

6. Procedure

Intent Attribute

The **intent** attribute (intent(in), intent(out), intent(inout)) tells the compiler:

1. Whether the procedure will read, write, or both read and write the dummy argument.
2. It also makes the interface contract clear to the caller, improving readability and safety.

Attribute	Meaning	Caller restriction
intent(in)	read-only	may pass variable, constant, or expression
intent(out)	write-only	must pass a variable
intent(inout)	read-write	must pass a variable

6. Procedure

Intent Example

```
Program calling_func
 implicit none
 real :: x, y, z, disc

  x = 1.0
  y = 5.0
  z = 2.0

  call intent_example(x, y, z, disc)

  Print *, "The value of the discriminant is"
  Print *, disc

end program calling_func
```

```
subroutine intent_example (a, b, c, d)
  implicit none

   real, intent (in) :: a
   real, intent (in) :: b
   real, intent (in) :: c
   real, intent (out) :: d
   d = b * b - 4.0 * a * c
   ! c= 3
end subroutine intent_example
```

```
Before call intent_example
   1.00000000
   5.00000000
   2.00000000
   0.00000000
After call intent_example
   1.00000000
   5.00000000
   2.00000000
   17.0000000
```

```
     c = 3
    1
Error: Dummy argument ‘c’ with INTENT(IN) in variable definition context (assignment) at (1)
```

6. Procedure

Internal procedure

Definition:

When a procedure is contained within a program, it is called an internal procedure of that program.

Format style:

```
program name
 implicit none
 ! type declaration statements
 ! executable statements
. . .
contains
 ! internal procedures
. . .
end program name
```

```
Program main
 implicit none
  real :: a, b
  a = 2.0
  b = 3.0
  Print *, "Before calling swap"
  Print *, "a = ", a
   Print *, "b = ", b
  call swap(a, b)
  Print *, "After calling swap"
   Print *, "a = ", a
   Print *, "b = ", b
contains
  subroutine swap(x, y)
    real :: x, y, temp
    temp = x
    x = y
    y = temp
  end subroutine swap
end program main
```

```
Before calling swap
a =     2.00000000
b =     3.00000000
After calling swap
a =     3.00000000
b =     2.00000000
```

7. Module

Concept

Definition:

A module is like a package where you can **store functions and subroutines**—especially useful when you're writing a very large program, or when your functions/subroutines need to be reused across several programs.

Modules are used to:

- Encapsulate subprograms, data, and interface blocks
- Define global data that can be shared by many subroutines
- Declare variables that are automatically available in any subroutine you choose
- Import an entire module into another function or subroutine for use.

7. Module

Concept

A module consists of two parts:

- Statement declarations
- Subroutine and function definitions.

Format style:

```
 module name
  [statement declarations]
  [contains [subroutine and function definitions] ]
end module [name]
```

Usage and scoping rules:

- You can add as **many modules** as you need; each module resides its own file and is compiled separately.
- A single module can be used by many **different** programs.
- The same module can be **reused** any number of times within one program.
- Variables declared in the module’s specification part are **global** within the module.
- Variables declared in the module become **global** in scoping unit (program, subroutine,function, module)that uses the module.
- The **use** statement may appear in the main program or in any subroutine or module that needs access to entities declared in the given module.

7. Module

中国科学院计算机网络信息中心
Computer Network Information Center, Chinese Academy of Sciences

➢ Example

```fortran
module constants
implicit none
   real, parameter :: pi = 3.1415926536
   real, parameter :: e = 2.7182818285

contains
   subroutine show_consts()
      print*, "Pi = ", pi
      print*,  "e = ", e
   end subroutine show_consts

end module constants
```

```fortran
program module_example
use constants
implicit none
   real :: x, ePowerx, area, radius
   x = 2.0
   radius = 7.0
   ePowerx = e ** x
   area = pi * radius**2
   call show_consts()
   print*, "e raised to the power of 2.0 = ", ePowerx
   print*, "Area of a circle with radius 7.0 = ", area
end program module_example
```

Compilation ↓

constants.mod

```
Pi =     3.14159274
e =    2.71828175
e raised to the power of 2.0 =     7.38905573
Area of a circle with radius 7.0 =     153.938049
```

7. Module

中国科学院计算机网络信息中心
Computer Network Information Center, Chinese Academy of Sciences

Accessibility

Accessibility of Variables and Subprograms in a Module:

By default, every variable and subroutine in a module is made available to any program unit that uses the module via **the use statement**. However, you can restrict this visibility by using the **PRIVATE** and **PUBLIC** attributes. Any variable or subroutine declared **PRIVATE** cannot be referenced outside the module.

Example:

The following sample illustrates the idea. In the earlier example we had two module variables, e and pi. Let us declare them PRIVATE and observe the resulting output.

```
module constants
implicit none
  real, parameter, private  :: pi = 3.1415926536
  real, parameter, private :: e = 2.7182818285
contains
  subroutine show_consts()
    print*, "Pi = ", pi
    print*,  "e = ", e
  end subroutine show_consts
end module constants
```

```
program module_example
use constants
implicit none
  real :: x, ePowerx, area, radius
  x = 2.0
  radius = 7.0
  ePowerx = e ** x
  area = pi * radius**2
  call show_consts()
  print*, "e raised to the power of 2.0 = ", ePowerx
  print*, "Area of a circle with radius 7.0 = ", area
end program module_example
```

Error!!!

```
    ePowerx = e ** x
                1
Error: Symbol ‘e’ at (1) has no IMPLICIT type
module_2.F90:22:12:

    area = pi * radius**2
              1
Error: Symbol ‘pi’ at (1) has no IMPLICIT type
```

8. Example & Conclusion

Example (Compute π)

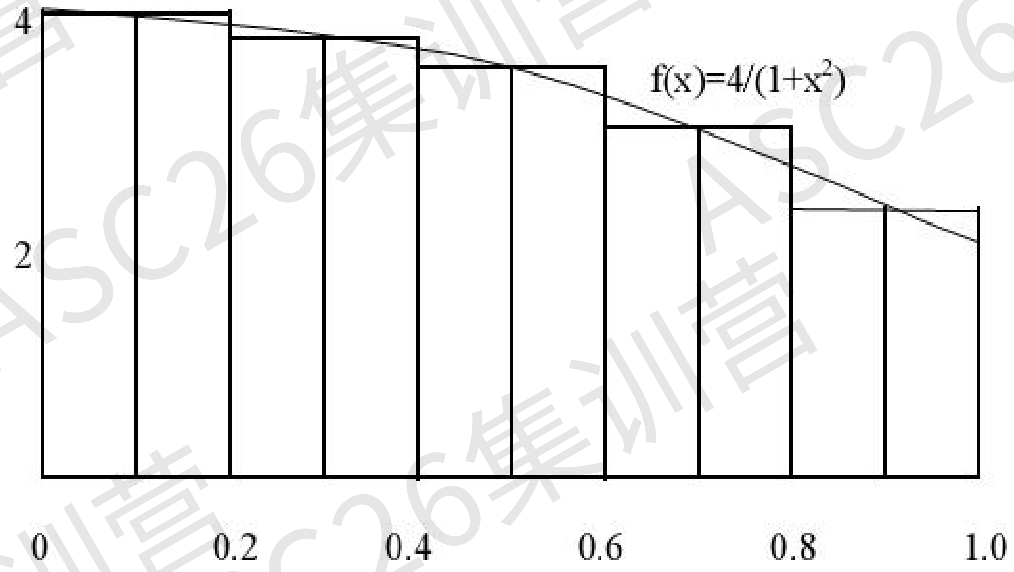
$$\int_0^1 \frac{1}{1+x^2}\,dx = arctan(x)|_0^1 = arctan(1) - arctan(0) = arctan(1) = \frac{\pi}{4}$$

Assume $f(x) = \dfrac{4}{1+x^2}$

Then $\int_0^1 f(x)dx = \pi$

$$\pi \approx \sum_{i=1}^{n} f\left(\frac{2 \times i - 1}{2N}\right) \times \frac{1}{N}$$

$$= \frac{1}{N} \times \sum_{i=1}^{n} f\left(\frac{i - 0.5}{N}\right)$$



8. Example & Conclusion

➢ Example (Compute π)

```
Main Program (program main)
    │
    ├── Module pi_computation
    │    │
    │    ├── Constant definition: dp = kind(8)     [Double precision type definition]
    │    │
    │    ├── Function 1: f_integrand(x)           [Integrand function]
    │    │    └── Computes: 4.0_dp / (1.0_dp + x*x)
    │    │
    │    └── Function 2: compute_pi_integral(n)      [Main function to compute π]
    │         ├── Calls f_integrand(x)           [Computes function value at each point]
    │         └── Returns π approximation
    │
    └── Internal subroutine in main program: test_different_intervals()
         └── Calls compute_pi_integral()          [Tests accuracy for different n values]
```

```
========= Computing π by Numerical Integration =========
Integration Method: Midpoint Rectangle Method
Integration Interval: [0, 1]
Integrand Function: f(x) = 4/(1+x²)

Parameter Settings:
  Number of intervals n =            100
  Step size h =     1.0000000000000000E-002

Computation Results:
  Approximate value π ≈     3.1416009869231254
  Exact value π =    3.1415926535897931
  Absolute error =    8.3333333322777037E-006
  Relative error =    2.6525823845289050E-006

Performance:
  Computation time =    4.0000000000005309E-006  seconds

Accuracy Test for Different Number of Intervals:
------------------------------------------------------------
   n      |     π Approx.     |    Abs. Error    |  Rel. Error
------------------------------------------------------------
     10 |    3.1424259850 |      8.3333E-04 |    2.6526E-04
    100 |    3.1416009869 |      8.3333E-06 |    2.6526E-06
   1000 |    3.1415927369 |      8.3333E-08 |    2.6526E-08
  10000 |    3.1415926544 |      8.3334E-10 |    2.6526E-10
 100000 |    3.1415926536 |      8.3684E-12 |    2.6637E-12
------------------------------------------------------------
```

8. Example & Conclusion

Example (Compute π)

Module & Function

```fortran
module pi_computation
  implicit none

  ! Define double precision type
  integer, parameter :: dp = kind(8)

  ! Public interface
  public :: compute_pi_integral, f_integrand

contains

  ! Integrand function
  real(dp) function f_integrand(x) result(y)
    real(dp), intent(in) :: x
    y = 4.0_dp / (1.0_dp + x*x)
  end function f_integrand

  ! Main function to compute π
  real(dp) function compute_pi_integral(num_intervals)  result(pi_approx)
    integer, intent(in) :: num_intervals
    integer :: i
    real(dp) :: h, x_midpoint, sum_val
  ! Initialization
    h = 1.0_dp / real(num_intervals, dp)
    sum_val = 0.0_dp
  ! Numerical integration using midpoint rectangle method
    do i = 1, num_intervals
      ! Calculate interval midpoint
      x_midpoint = h * (real(i, dp) - 0.5_dp)
      sum_val = sum_val + f_integrand(x_midpoint)
    end do
  ! Compute π approximation
    pi_approx = h * sum_val
  end function compute_pi_integral
end module pi_computation
```

8. Example & Conclusion

➢ Example (Compute π)

Main & Print & Call test_different_intervals

```
! Main program
program main
  use pi_computation
  implicit none
  integer :: n
  real(dp) :: pi_approx, pi_exact, error
  real(dp) :: start_time, end_time
  ! Set number of intervals
  n = 100
  ! Measure computation time
  call cpu_time(start_time)
  ! Compute π
  pi_approx = compute_pi_integral(n)
  pi_exact = 4.0_dp * atan(1.0_dp)
  error = abs(pi_approx - pi_exact)
call cpu_time(end_time)
! Output results
  print *, "========= Computing π by Numerical
Integration ========="
```

```
print *, "Integration Method: Midpoint Rectangle Method"
 print *, "Integration Interval: [0, 1]"
 print *, "Integrand Function: f(x) = 4/(1+x²)"
 print *, ""
 print *, "Parameter Settings:"
 print *, "  Number of intervals n = ", n
 print *, "  Step size h = ", 1.0_dp / real(n, dp)
 print *, ""
 print *, "Computation Results:"
 print *, "  Approximate value π ≈ ", pi_approx
 print *, "  Exact value π = ", pi_exact
 print *, "  Absolute error = ", error
 print *, "  Relative error = ", error / pi_exact
 print *, ""
 print *, "Performance:"
 print *, "  Computation time = ", end_time - start_time, " seconds“
! Test accuracy for different n values
 call test_different_intervals()
```

8. Example & Conclusion

➢ Example (Compute π)

Main & test_different_intervals & Print

```
contains
  ! Test accuracy for different numbers of intervals
  subroutine test_different_intervals()
    integer :: n_values(5)
    real(dp) :: approx_values(5), errors(5)
    integer :: i
    n_values = [10, 100, 1000, 10000, 100000]
    print *, "Accuracy Test for Different Number of Intervals:"
    print *, repeat("-", 60)
    print *, "   n     |     π Approx.     |    Abs. Error   |  Rel. Error"
    print *, repeat("-", 60)
    do i = 1, 5
      approx_values(i) = compute_pi_integral(n_values(i))
      errors(i) = abs(approx_values(i) - pi_exact)
      print '(I8, A, F15.10, A, ES14.4, A, ES12.4)', n_values(i), " | ", &
            approx_values(i), " | ",  errors(i), " | ",     errors(i) / pi_exact
    end do
    print *, repeat("-", 60)
  end subroutine test_different_intervals
end program main
```

```
========= Computing π by Numerical Integration =========
Integration Method: Midpoint Rectangle Method
Integration Interval: [0, 1]
Integrand Function: f(x) = 4/(1+x²)

Parameter Settings:
  Number of intervals n =          100
  Step size h =    1.0000000000000000E-002

Computation Results:
  Approximate value π ≈     3.1416009869231254
  Exact value π =    3.1415926535897931
  Absolute error =    8.3333333322777037E-006
  Relative error =    2.6525823845289050E-006

Performance:
  Computation time =    4.0000000000005309E-006  seconds

Accuracy Test for Different Number of Intervals:
------------------------------------------------------------
    n    |     π Approx.     |   Abs. Error   |  Rel. Error
------------------------------------------------------------
      10 |    3.1424259850 |      8.3333E-04 |  2.6526E-04
     100 |    3.1416009869 |      8.3333E-06 |  2.6526E-06
    1000 |    3.1415927369 |      8.3333E-08 |  2.6526E-08
   10000 |    3.1415926544 |      8.3334E-10 |  2.6526E-10
  100000 |    3.1415926536 |      8.3684E-12 |  2.6637E-12
------------------------------------------------------------
```

8. Example & Conclusion

Conclusion

- **Language Features**: Fortran is concise, efficient, and specifically designed for scientific computing.
- **Learning Path**: We followed the progression of "variable → operator → Control Structure → procedure → module", achieving a knowledge structure that evolves from foundational elements to functional organization.
- **Key Advice**: Write more, practice more, and think more deeply. True mastery stems from consistent practice.

[1] https://fortran-lang.org/zh_CN/learn/
[2] https://www.cainiaoya.com/fortran/fortran-module.html
[3] 彭国伦, Fortran 95 程序设计, 中国电力出版社, 2002.
[4] 白云等, Fortran 95 程序设计, 清华大学出版社, 2011.
[5] S. J. Chapman, Fortran 95/2003 for Scientists and Engineers, 3rd edition, McGraw-Hill, 2007

中国科学院计算机网络信息中心
Computer Network Information Center,
Chinese Academy of Sciences

Thank you!