제 4 부 — 프로시저와 모듈

11장

인수 전달과 인터페이스

The Contract Between Caller and Callee

intent · optional · 키워드 인수 · 가정형상 배열 · interface

THE CONTRACT

```
real, intent(in)  :: x
real, intent(out) :: y

optional :: sigma
if (present(sigma)) ...

call f(v(:), tol=1e-9)
```

학습 목표

이 장에서 익히는 네 가지

01

의도 명시

intent(in/out/inout) 로 인수의 쓰임을 배운다.

02

유연한 호출

선택적 인수와 키워드 인수로 호출을 활용한다.

03

배열 전달

배열 인수를 안전하게 주고받는다.

04

인터페이스 이해

인터페이스 블록과 명시적 인터페이스의 역할을 안다.

프로시저가 제 역할을 하려면, 호출하는 측과 호출되는 측 사이의 데이터 전달을 제어할 수 있어야 한다.

도입

인수 전달을 다스리는 네 개의 장치

1

intent 속성

인수의 입력 및 출력 방향을 지정하여 코드의 안정성을 높인다.

`in · out · inout`

2

선택적 인수

호출 시 인수를 생략하도록 한다. present 로 확인한 뒤에만 쓴다.

`optional · present`

3

키워드 인수

인수의 순서를 임의로 재배열하도록 한다.

`이름 = 값`

4

배열 인수 전달

다차원 데이터를 메모리 낭비 없이 주고받는 규칙.

`v(:)  가정형상`

그리고 이 모두를 떠받치는 것 — 명시적 인터페이스

선택적 인수 · 키워드 인수 · 가정형상 배열은 모두 명시적 인터페이스가 확보된 상태를 전제로 작동한다. 이 장치들이 오작동하지 않도록 유효성을 검증하는 것이 인터페이스의 역할이다.

내부 프로시저와 모듈 프로시저는 이 조건을 자동으로 만족한다 — contains 아래에 두면 별도의 작업이 필요 없다.

11.1 인수의 방향

intent(in / out / inout)

```
⟨type⟩, intent(⟨spec⟩) :: ⟨dummy-arg⟩
```

in

읽기 전용

프로시저가 이 인수를 읽기만 한다. 내부에서 값을 바꾸려 하면 컴파일 오류가 발생한다.

호출 측 → 프로시저

out

결과 출력

프로시저가 이 인수에 결과를 써서 내보낸다. 들어올 때의 기존 값은 정의되지 않은 것으로 취급되므로, 읽기 전에 반드시 먼저 대입해야 한다.

프로시저 → 호출 측

inout

읽고 갱신

들어온 값을 읽고 다시 갱신하여 돌려보낸다. 누산기처럼 이전 값에 새 결과를 더하는 경우에 쓴다.

양방향

왜 intent 를 지정하는가

컴파일러가 프로그래머의 의도와 다르게 작성된 부분을 정확히 잡아내어 버그를 미리 막아 주고, 컴파일 최적화에 필요한 정보를 제공한다. 이 책의 모든 가인수에는 intent 를 명시한다.

11.1 [예제]

입력은 보호하고 다중 결과 내보내기

```
r = 2.5_real64
call circle(r, area, perimeter)

contains

   subroutine circle(radius, a, p)
      real(real64), intent(in)  :: radius
      real(real64), intent(out) :: a, p
      a = pi * radius**2
      p = 2.0_real64 * pi * radius
   end subroutine circle
```

```
 radius = 2.5000      area = 19.6350      perimeter = 15.7080
```

intent(in) 인 radius 를 내부에서 바꾸려 하면

```
Error: Dummy argument 'radius' with INTENT(IN) in
  variable definition context (assignment) at (1)
```

원본을 보호한다

intent(in) 덕분에 서브루틴이 호출 측이 넘겨준 원본 변수 r 의 값을 절대 변경하지 못한다. 실수로 대입하는 코드를 써도 컴파일 단계에서 차단된다.

결과는 두 개

함수라면 값 하나뿐이지만, intent(out) 가인수를 둘 두어 넓이와 둘레를 한 번에 내보낸다.

[흔한 실수] kind 불일치

단정밀도 실수를 real(real64) 가인수에 그대로 넘기면 치명적인 오류가 될 수 있다. 리터럴에도 2.0_real64 처럼 접미사를 붙여 정밀도를 통일한다.

11.1 [예제]

intent(inout) 으로 값을 누적하기

```
total = 0.0_real64
do i = 1, 5
   call add_square(total, real(i, real64))
end do

   subroutine add_square(acc, x)
      real(real64), intent(inout) :: acc
      real(real64), intent(in)    :: x
      acc = acc + x**2
   end subroutine add_square
```

```
sum of squares 1..5 =      55.0
```

acc 가 쌓여 가는 과정

0	1	5	14	30	55
	+1	+4	+9	+16	+25

왜 inout 인가

acc 는 매번 이전까지의 합산 값을 읽어 들인 뒤 새 값으로 갱신한다. 읽기와 쓰기를 모두 하므로 inout 이 정답이다.

[흔한 실수] out 으로 누적

intent(out) 을 쓰면 서브루틴이 시작되자마자 total 이 가지고 있던 값을 지워 버린다. 빈 공간에서 시작하는 셈이라 누적이 불가능하다.

신뢰할 수 없는 결과

out 으로 선언한 채 acc = acc + x**2 을 쓰면, 더하려는 기준 acc 에 쓰레기 값이 들어 있을 수 있어 결과를 전혀 신뢰할 수 없다.

11.2 유연한 호출

선택적 인수와 키워드 인수

키워드 인수

keyword argument

이름=값 형태로 실인수를 건네면, 정의된 순서와 상관없이 인수를 자유롭게 배치할 수 있다. 인수가 많을 때 호출문의 가독성을 높인다.

선택적 인수

optional argument

optional 속성을 붙인 가인수는 호출할 때 생략할 수 있다. 다만 프로시저 안에서는 present 로 실제 전달되었는지 확인한 뒤에만 써야 한다.

1	**선언부 규칙**	다른 속성과 함께 optional 을 추가로 지정한다.	`⟨type⟩, intent(in), optional :: ⟨dummy-arg⟩`
2	**호출부 규칙**	가인수의 이름을 명시하는 키워드 인수 방식으로 부른다.	`call ⟨name⟩(⟨keyword⟩=⟨value⟩, ...)`
3	**존재 확인 규칙**	present 함수로 전달 여부를 안전하게 검사한다.	`if (present(⟨dummy-arg⟩)) then`

[흔한 실수] 전달되지 않은 인수를 present 확인 없이 참조하면 표준 위반이며, 실행 중에 존재하지 않는 메모리를 건드려 세그멘테이션 오류로 이어진다.

11.2 [예제]

평균과 폭을 선택할 수 있는 가우스 함수

```
y1 = gaussian(1.0_real64)
y2 = gaussian(1.0_real64, mu=0.5_real64)
y3 = gaussian(1.0_real64, mu=0.5_real64, sigma=2.0_real64)

   function gaussian(x, mu, sigma) result(y)
      real(real64), intent(in)           :: x
      real(real64), intent(in), optional :: mu, sigma
      real(real64) :: y, center, width

      center = 0.0_real64
      width  = 1.0_real64
      if (present(mu))    center = mu
      if (present(sigma)) width  = sigma

      y = exp(-0.5_real64 * ((x - center) / width)**2)
   end function gaussian
```

둘 다 생략 0.606531

```
mu=0, sigma=1
```

mu 만 지정 0.882497

```
mu=0.5
```

둘 다 지정 0.969233

```
mu=0.5, sigma=2.0
```

present 로 전달 여부를 확인한 뒤, 전달된 값 또는 기본값을 분기해 연산에 쓴다.

11.2 [예제]

키워드로 인수의 순서를 바꿔 부르기

```
print '(f12.4)', power(base=2.0_real64, exponent=10.0_real64)
print '(f12.4)', power(exponent=10.0_real64, base=2.0_real64)

! 두 호출은 순서만 다를 뿐 같은 값을 계산한다
```

```
1024.0000
1024.0000
```

함수를 정의할 때 가인수의 순서는 상관없다 — 문제는 호출할 때다.

위치 인수가 키워드 인수보다 항상 먼저 와야 한다 — 이름=값 을 한 번이라도 쓰기 시작하면 그 뒤의 모든 인수는 키워드 형태를 유지해야 한다.

가능

```
power(10.0_real64, exponent=2.0_real64)
```

값만 적는 위치 인수를 앞에, 키워드 인수를 뒤에 두었다.

오류

```
power(base=10.0_real64, 2.0_real64)
```

키워드를 쓰기 시작했으면 뒤의 2.0 도 exponent= 를 붙여야 한다.

11.3 배열 인수 전달

명시적 형상 배열과 가정형상 배열

명시적 형상 배열

explicit-shape array

가인수를 선언할 때 배열의 크기를 직접 적는다. 크기를 나타내는 정수형 인수를 별도로 함께 넘겨받는다. 과거에 주로 쓰던 방식이며 인터페이스 종류와 상관없이 항상 동작한다.

```
real :: v(n)
```

가정형상 배열

assumed-shape array

크기 자리를 콜론으로 비워 둔다. 실제 크기와 형상을 호출 측의 실인수로부터 그대로 물려받는다. 코드가 간결하고 크기 인수가 필요 없어 현대 Fortran 에서 권장된다.

```
real :: v(:)
```

프로시저 안에서 배열 정보를 알아내는 세 함수

`size(array)`	전체 요소 개수
`lbound(array)`	하한 인덱스 (시작 번호)
`ubound(array)`	상한 인덱스 (끝 번호)

가정형상 배열을 쓰려면 호출 시점에 명시적 인터페이스가 반드시 보여야 한다 — 내부 프로시저와 모듈 프로시저에서는 자동으로 보장된다.

11.3 [예제]

배열의 평균과 표준편차 계산

```
real(real64) :: sample(5) = [4.0_real64, 8.0_real64, 15.0_real64, &
                             16.0_real64, 23.0_real64]
call stats(sample, avg, sd)

   subroutine stats(v, mean, stddev)
      real(real64), intent(in)  :: v(:)
      real(real64), intent(out) :: mean, stddev
      integer :: n
      n = size(v)
      mean = sum(v) / real(n, real64)
      stddev = sqrt(sum((v - mean)**2) / real(n, real64))
   end subroutine stats
```

```
n         = 5
mean      =   13.2000
std dev   =    6.6151
```

lbound 와 ubound 를 조합하면 호출 측이 인덱스를 1부터 시작했든 다른 정수부터 시작했든 안전하게 순회할 수 있다.

길이를 몰라도 된다

stats 는 전달되는 배열의 길이가 5인지 500인지 상관하지 않는다. size(v) 가 호출 시점의 실제 길이를 조사해 돌려주므로, 같은 서브루틴을 임의의 길이에 재사용할 수 있다.

8장의 배열 연산과 결합

전체 배열 연산 v - mean 과 내장 함수 sum 이 가정형상 배열 문맥에서도 아무 제약 없이 자연스럽게 결합된다.

크기 인수가 필요 없다

명시적 형상 배열이었다면 call stats(sample, 5, avg, sd) 처럼 길이를 따로 넘겨야 했다.

11.4 인터페이스

명시적인가, 암시적인가

컴파일러가 호출 지점에서 인수 개수 · 종류 · 속성을 정확히 알고 있으면 명시적(explicit), 모르면 암시적(implicit) 인터페이스다.

○ 내부 프로시저

program 이나 다른 프로시저의 contains 내부에 정의된다.

명시적 – 자동

○ 모듈 프로시저

모듈의 contains 내부에 정의되고, 호출 측에서 use 로 가져온다.

명시적 – 자동

× 외부 프로시저

어떤 프로그램 단위에도 속하지 않고 파일 수준에 독자적으로 존재한다.

암시적 – 직접 제공 필요

인터페이스 블록의 구성 규격

```
interface
   ⟨procedure-header⟩
      ⟨declarations of dummy arguments⟩
   ⟨end-statement⟩
end interface
```

선언부만 기술	헤더와 가인수의 자료형·속성 선언만 넣는다.
실행문 배제	실제 연산문, 지역 변수 선언, contains 구문은 일절 넣지 않는다.
본질	"이 프로시저를 부를 때 인수의 형태가 이러하다"는 호출 규약의 선언이다.

11.4 [예제]

외부 서브루틴과 인터페이스 블록

외부 서브루틴 normalize.f90

```
subroutine normalize(v)
   use iso_fortran_env, only: real64
   implicit none
   real(real64), intent(inout) :: v(:)
   real(real64) :: total
   total = sum(v)
   if (total /= 0.0_real64) v = v / total
end subroutine normalize
```

호출 측 use_normalize.f90

```
   interface
      subroutine normalize(v)
         use iso_fortran_env, only: real64
         implicit none
         real(real64), intent(inout) :: v(:)
      end subroutine normalize
   end interface
```

```
 normalized weights:
   0.2000   0.3000   0.4000   0.1000         sum =    1.0000
```

인터페이스 블록이 하는 일

컴파일러는 normalize 가 배정밀도 실수형 가정형상 배열 하나를 intent(inout) 으로 전달받는다는 사실을 호출 지점에서 정확히 인지한다. 이를 바탕으로 실행 시점에 배열의 실제 크기와 형상 정보를 외부 프로시저로 올바르게 넘겨준다.

11.4 [예제]

매개변수만 바꿔 곡선군 만들기

```
real(real64) :: sigmas(3) = [0.5_real64, 1.0_real64, 2.0_real64]

write(u, '(a)') "x,s0p5,s1p0,s2p0"
do i = 0, n
   x = x_min + (x_max - x_min) * real(i, real64) / real(n, real64)
   write(u, '(f9.4)', advance="no") x
   do j = 1, size(sigmas)
      write(u, '(",", f9.6)', advance="no") gaussian(x, sigma=sigmas(j))
   end do
   write(u, '(a)') ""
end do
```

```
x,s0p5,s1p0,s2p0
  -6.0000, 0.000000, 0.000000, 0.011109
  -5.9400, 0.000000, 0.000000, 0.012150
```

키워드 인수의 실전 활용

gaussian(x, sigma=sigmas(j)) — 중간의 mu 를 건너뛰고 sigma 만 지정한다. 키워드가 없었다면 mu 자리를 채워야 했다.

advance="no"

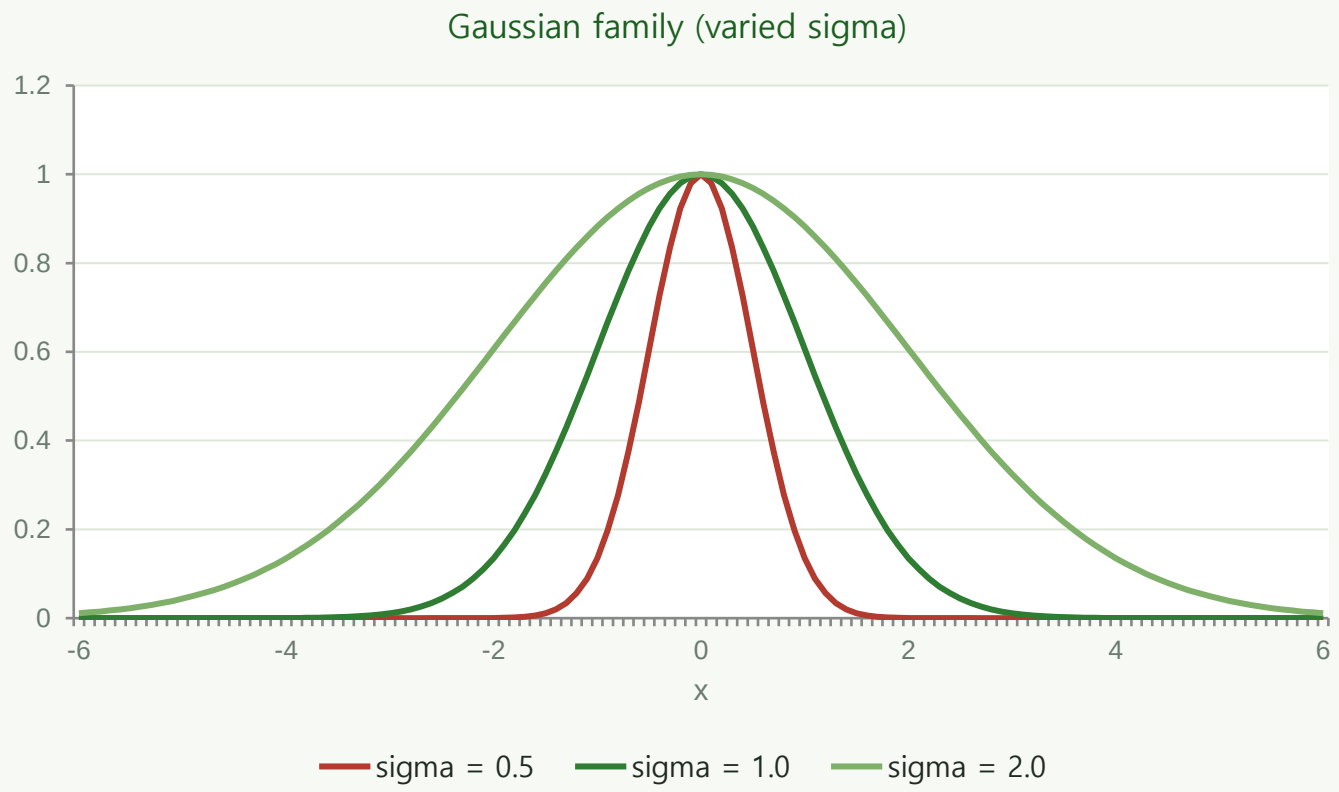
줄바꿈 없이 이어서 쓴다. 한 행에 x 와 세 개의 결과를 나란히 기록하기 위한 장치다.

하나의 함수, 여러 곡선

같은 프로시저를 매개변수만 바꿔 반복 호출하면 곡선군(family of curves)이 만들어진다.

11.4 [예제]

sigma 만 바꾼 세 곡선



mu 는 세 호출 모두에서 생략되어 기본값 0 이 쓰였다.

sigma 가 폭을 결정한다

값이 작을수록 중심이 좁고 뾰족하게 솟아오르고, 클수록 좌우로 넓고 완만한 종형 곡선이 된다. 세 곡선이 하나의 평면 위에 겹쳐 표현된다.

설계 목적과 정확히 부합한다

매개변수의 지정 값만 바꾸어 동일한 프로시저를 반복 호출하는 방식은, 선택적 인수와 키워드 인수가 존재하는 이유 그 자체다.

오류 학습

명시적 인터페이스가 인수를 검사한다

```
real(real64) :: a
a = rect_area(3.0_real64)    ! height 가 없다
```

```
Error: Missing actual argument
  for argument 'height' at (1)
```

인터페이스 블록이 없었다면

컴파일러는 호출 지점에서 인수의 개수가 맞는지조차 검사하지 못하고 그대로 통과시킨다. 그 결과 프로그램 실행 중에 메모리 참조 오류나 비정상 종료를 일으킨다.

인터페이스 블록이 있었기 때문에

컴파일러가 빌드 단계에서 즉시 오류를 내고 실행 파일 생성을 거부한다. 명시적 인터페이스의 가장 큰 가치가 바로 이 컴파일 단계의 유효성 검사다.

그래서 원칙은

- 모든 프로시저는 내부 또는 모듈 프로시저로 설계한다
- 부득이 외부 프로시저를 쓴다면 인터페이스 블록을 반드시 작성한다

선택적 인수 · 키워드 인수 · 가정형상 배열은 모두 명시적 인터페이스를 전제로 작동한다 — 구조화되지 않은 외부 프로시저는 이 전제를 스스로 만족할 수 없다.

오류 학습

인터페이스 없는 외부 프로시저의 오작동

```
! ext_proc.f90 — 외부 서브루틴
subroutine scale_vec(v, factor)
   real(real64), intent(inout) :: v(:)    ! 가정형상 배열
   real(real64), intent(in)    :: factor
   v = v * factor
end subroutine scale_vec
```

컴파일은 아무 경고 없이 통과한다

```
Program received signal SIGSEGV:
  Segmentation fault - invalid memory reference.
```

무엇이 전달되는가

명시적 인터페이스 O → **배열 서술자 (주소 + 형상 + 보폭)**

명시적 인터페이스 X → **시작 주소만**

왜 죽는가

가정형상 배열은 시작 주소만이 아니라 크기(형상)와 메모리 간격(보폭)까지 묶은 배열 서술자(array descriptor) 형태로 전달된다. 암시적 인터페이스 상태에서는 컴파일러가 서술자 대신 평범한 시작 주소만 넘기고, 받는 쪽은 그 주소값을 서술자 구조체로 오해해 해석한다. 결국 엉뚱한 메모리를 참조하며 멈춘다.

1 인터페이스 블록 작성

호출 측에 interface … end interface 를 명시해 컴파일러가 배열 서술자를 정상 생성하게 한다.

2 내부 · 모듈 프로시저로 설계

훨씬 간결하고 권장되는 방법. 명시적 인터페이스가 자동 생성되므로 블록을 수작업할 필요가 없다.

11장 정리

요약

인수의 방향과 유연성

- intent — 모든 가인수에 intent(in/out/inout) 을 명시한다. in 은 읽기 전용, out 은 결과 출력(들어온 값은 신뢰 불가), inout 은 읽고 갱신. 위반은 컴파일 단계에서 잡힌다.
- 선택적 인수 — optional 속성을 붙이면 호출할 때 생략 가능하다. 사용 전에 반드시 present 로 확인한다.
- 키워드 인수 — 이름=값 형태로 순서에 얽매이지 않고 호출한다. 위치 인수를 먼저, 키워드 인수를 나중에 둔다.

배열과 인터페이스

- 배열 인수 — 가정형상 배열 v(:) 은 형상을 실인수에서 물려받고 size · lbound · ubound 로 알아낸다. 명시적 형상보다 간결하지만 명시적 인터페이스가 필요하다.
- 명시적 인터페이스 — 내부·모듈 프로시저는 자동으로 갖는다. 외부 프로시저는 interface … end interface 블록으로 직접 제공해야 하며, 선택적 인수·가정형상 배열을 쓰려면 필수다.
- 명시적 인터페이스가 있어야 컴파일러가 인수를 검사한다.

한 줄 요약: 인수마다 방향을 밝히고, 프로시저는 contains 안에 둔다.

연습 문제

1. 섭씨를 intent(in) 으로 받아 화씨를 intent(out) 으로 돌려주는 서브루틴 to_fahrenheit 를 작성하라.

2. 두 실수 값을 맞바꾸는 서브루틴 swap 을 작성하고, 두 가인수의 intent 를 무엇으로 할지 근거와 함께 밝혀라.

3. intent(in) 가인수에 값을 대입하는 코드를 일부러 작성해 컴파일하고, gfortran 오류 메시지를 옮겨 적어라.

4. base 와 exponent 를 받는 거듭제곱 함수를 만들고, 키워드 인수로 두 가지 순서로 호출해 결과가 같음을 확인하라.

5. 가정형상 배열의 최댓값과 최솟값을 각각 intent(out) 으로 돌려주는 서브루틴을 작성하라.

6. 가정형상 정수 배열을 받아 size 로 원소 수를 출력하고, 길이가 다른 두 배열로 각각 호출해 보라.

7. 선택적 인수 tol 을 받는 is_close(a, b, tol) 을 작성하라. 생략되면 1.0e-9_real64 를 기본값으로 쓴다.

8. stats 서브루틴을 확장해, 선택적 인수 use_sample 이 참이면 표본 표준편차(분모 n−1)를 계산하게 하라.

9. 외부 함수 dot(u, v) 를 별도 파일에 작성하고, 메인 프로그램에서 인터페이스 블록을 통해 호출하라.

10. 가정형상 배열의 원소를 구간으로 자르는 서브루틴을 작성하되, 하한 lo 와 상한 hi 를 선택적 인수로 두어라.

다음 장에서는 모듈 — module 과 use, 접근 제어와 모듈 프로시저를 다룬다.