제 3 부 — 배열

8장
배열 연산

Whole-Array Thinking

전체 배열 연산 · where 마스킹 · 축약 · reshape · matmul

NO LOOP NEEDED

```
c = a + b

c = 2.0 * a + 1.0

where (y < 0.0)
   y = 0.0
end where

c = matmul(a, b)
```

학습 목표

이 장에서 익히는 세 가지

01

전체 배열 연산

반복문 없이 배열 전체를 한 번에 계산한다.

02

조건부 연산

where 구문 마스킹을 이용해 선택적으로 연산한다.

03

내장 함수 활용

sum · matmul · transpose · reshape 등으로 배열을 다룬다.

과학·공학 계산에서 Fortran 이 여전히 독보적인 이유 중 하나가 바로 이 전체 배열 연산(whole-array operation) 기능이다.

도입 문제

루프 없이 끝낼 수는 없을까

1

길이 1000인 두 배열의 합

do i = 1, 1000 루프를 돌리는 대신, c = a + b 라는 단 한 줄로 끝낼 수는 없을까?

```
c = a + b
```

2

60 × 60 격자의 음수만 0으로

if 문을 품은 이중 do 루프를 설계하는 대신, 조건식 한 줄로 처리할 수는 없을까?

```
where (m < 0.0) m = 0.0
```

Fortran 은 배열을 하나의 독립된 값처럼 다룬다 — 사칙연산은 물론 sin · sqrt 같은 수학 함수, 합계와 최댓값을 구하는 통계 함수, 행렬 곱셈까지 배열 전체에 곧바로 적용된다.

코드가 짧아질 뿐 아니라, 컴파일러가 벡터화(vectorization)와 최적화를 수행하기에 매우 유리한 환경이 된다.

8.1 전체 배열 연산

요소별 연산과 스칼라 확장

요소별 연산 (element-by-element)

```
c = a + b
```

모든 인덱스 i 에 대해 c(i) = a(i) + b(i) 를 수행한다. + − * / ** 가 모두 요소별로 동작한다.

스칼라 확장 (scalar expansion)

```
c = 2.0 * a + 1.0
```

스칼라 값은 컴파일러가 배열의 형상만큼 내부적으로 확장해 모든 원소에 동일하게 적용한다.

1 **정합성 (conformance)** — 연산에 참여하는 모든 배열 피연산자는 차원의 수(rank)와 각 차원의 크기(extent)가 정확히 일치해야 한다.

2 **대입 규칙** — 대입문 좌변 배열의 형상은 우변 연산 결과와 반드시 동일해야 한다.

3 **요소별 함수 (elemental)** — sin · cos · sqrt · exp · abs 등 표준 수학 내장 함수는 대부분 요소별 함수라, 스칼라 대신 배열을 그대로 넘길 수 있다.

sqrt(b) 를 호출하면 배열 b 의 각 요소의 제곱근을 계산해 동일한 형상의 배열로 돌려준다.

8.1 [예제]

루프가 한 줄도 없는 배열 연산

```
a = [(real(i), i = 1, n)]        ! 1.0 2.0 3.0 4.0 5.0
b = [(real(i)**2, i = 1, n)]     ! 1.0 4.0 9.0 16.0 25.0

c = a + b              ! 요소별 덧셈
c = a * b              ! 요소별 곱셈
c = 2.0 * a + 1.0      ! 스칼라 확장
c = sqrt(b)            ! 요소별 내장 함수
```

```
a + b   =    2.0    6.0   12.0   20.0   30.0
a * b   =    1.0    8.0   27.0   64.0  125.0
2a + 1  =    3.0    5.0    7.0    9.0   11.0
sqrt(b) =    1.0    2.0    3.0    4.0    5.0
```

반복 출력 서식 *(f7.1)

출력 버퍼에 남은 원소가 있는 한 f7.1 서식을 계속 반복하라는 뜻이다. 배열 길이를 명시하지 않아도 전체를 한 번에 출력한다.

루프 없는 연산의 이점

c = a + b 는 내부적으로 크기 n 만큼의 요소별 덧셈을 일괄 수행한다. 데이터 규모가 커질수록 가독성과 자동 최적화에서 큰 강점을 준다.

8.1 [흔한 실수]

형상 정합과 정수 나눗셈

형상이 다르면 연산이 안 된다

```
real :: a(5), b(4), c(5)

c = a + b      ! [5] 와 [4]
```

```
Error: Shapes for operands at (1) and (2)
  are not conformable
```

정합 오류는 대개 인덱스 실수나 잘못된 크기 선언에서 온다. 슬라이싱할 때도 좌변과 우변의 길이를 일치시켜야 한다. a(2:5) = b(1:4) 는 둘 다 길이 4라 정합이다.

정수 나눗셈은 배열에서도 정수다

```
[(i, i = 1, 4)] / 2

! → [0, 1, 1, 2]
```

real 배열이 아니라 integer 배열을 나누면 요소마다 정수 나눗셈이 일어난다. 실수 결과가 필요하면 피연산자를 real(...) 로 만든다. 4장의 정수 나눗셈 규칙이 배열에서도 똑같이 적용된다.

8.2 where 구문

조건을 만족하는 요소에만 대입한다

마스크(mask)란 무엇인가

x < 0.0 같은 관계식을 배열 전체에 적용해 얻은, 각 요소의 참·거짓을 담은 논리 배열이다. 원본 배열과 동일한 차원과 크기를 가진다. where 는 이 마스크가 참인 위치에만 대입 연산을 적용한다.

단일 where 문

```
where (mask) array = expression
```

조건을 만족하는 요소에 대한 대입을 단 한 줄로 명시한다.

`elsewhere (mask2)` 새 조건을 덧붙이면 if-else if 처럼 다단계 필터링 분기가 된다.

`elsewhere` 단독 이전 모든 조건에 해당하지 않는 '나머지 모든 요소'를 대상으로 한다.

블록 where 문

```
where (mask1)
   assignments_1
[elsewhere (mask2)
   assignments_2]
[elsewhere
   assignments_3]
end where
```

8.2 [예제]

음수 클리핑 — 마스크가 하는 일

```
x = [(-4.0 + real(i), i = 0, n - 1)]

y = x
where (y < 0.0)
   y = 0.0
end where
```

x	-4.0	-3.0	-2.0	-1.0	0.0	1.0	2.0	3.0
mask	T	T	T	T	F	F	F	F
y	0.0	0.0	0.0	0.0	0.0	1.0	2.0	3.0

마스크가 참(T)인 앞의 네 자리에만 0.0 이 대입되고, 나머지는 원래 값이 그대로 보존된다.

클리핑(clipping)이란

특정 범위를 벗어나는 데이터의 끝을 잘라내어 지정한 범위에 맞추는 것. 여기서는 0보다 작은 값을 모두 찾아 0으로 강제 변환했다.

컴파일러가 만드는 논리 배열

where (y < 0.0) 가 실행될 때 [.true., .true., .true., .true., .false., .false., .false., .false.] 라는 내부 논리 마스크가 만들어진다.

8.2 [예제]

부호 분류 — where · elsewhere · elsewhere

```
x = [-2.0, -1.0, 0.0, 1.0, 2.0, 3.0]

where (x > 0.0)
   sgn =  1.0
elsewhere (x < 0.0)
   sgn = -1.0
elsewhere
   sgn =  0.0
end where
```

x	-2.0	-1.0	0.0	1.0	2.0	3.0
sgn	-1.0	-1.0	0.0	1.0	1.0	1.0

복잡한 분기 조건을 처리할 때 where 구문은 가독성을 높이고 구현 의도를 명확히 드러낸다.

1 **where (x > 0.0)**

양수 위치를 찾아 sgn 의 해당 칸에 1.0 을 대입한다.

2 **elsewhere (x < 0.0)**

앞선 필터에서 제외된 요소 중 음수 위치에 -1.0 을 대입한다.

3 **elsewhere**

두 조건을 모두 만족하지 못한 나머지, 즉 0.0 인 위치를 최종 식별한다.

8.2 주의 사항

where 는 if 가 아니다

1

형상 정합성

where 구문 내부의 모든 배열 — 마스크 조건식, 피연산자, 대입 대상 — 은 반드시 동일한 형상을 가져야 한다. 다르면 컴파일 오류다.

2

대입만 올 수 있다

where 블록 내부에는 원칙적으로 배열 대입 연산만 올 수 있다. 일반적인 print 문이나 do 반복문 같은 제어문을 혼용할 수 없다.

3

최적화에 유리하다

명시적 do 루프 안에 if 를 배치하는 것보다 컴파일러 최적화에 훨씬 유리하다. 배열 전체에 하드웨어 수준의 병렬 연산을 유도할 수 있다.

where 의 마스크는 배열 전체에 대한 요소별 조건이지, if 처럼 한 번 평가되는 스칼라 조건이 아니다 — 분기 전체를 건너뛸지 결정하는 것은 if, 요소별 선택 대입은 where 로 구분한다.

8.3 축약 함수

배열을 하나의 값으로 줄인다

`sum(a)` / `product(a)`	배열 내 모든 요소의 총합 또는 총곱을 계산한다.
`maxval(a)` / `minval(a)`	배열 내 전체 요소 중 최댓값 또는 최솟값을 반환한다.
`maxloc(a)` / `minloc(a)`	최댓값 또는 최솟값이 위치한 인덱스(첨자)를 찾아 반환한다.
`count(mask)`	마스크(논리 배열) 조건에서 참인 요소의 총 개수를 구한다.
`any(mask)` / `all(mask)`	단 하나라도 참인지, 또는 모든 요소가 참인지를 논리값으로 반환한다.

축약 함수는 배열을 받아 스칼라(또는 원본보다 낮은 차원의 배열)로 변환해 돌려준다.

조건 검색이 한 줄로 줄어든다

"기상 관측 데이터 중 임계 온도를 초과한 일수"를 구하려면 복잡한 루프를 설계할 필요 없이 count(temp > limit) 한 줄이면 된다.

maxloc 의 반환 규칙

다차원 대응이 기본이라, 인수를 생략하면 크기 1인 1차원 배열로 결과를 돌려준다. 1차원 배열에서 단일 정수로 받으려면 dim=1 을 지정한다.

8.3 [예제]

축약 함수를 한자리에서 보기

```
real :: a(6)

a = [3.0, 1.0, 4.0, 1.0, 5.0, 9.0]
```

배열 → 스칼라

여덟 개의 호출이 모두 배열 하나를 받아 단일 값으로 줄여 돌려준다.

`sum(a)`	= 23.0	모든 요소의 합
`product(a)`	= 540.0	모든 요소의 곱
`maxval(a)`	= 9.0	가장 큰 값
`minval(a)`	= 1.0	가장 작은 값

`maxloc(a, dim=1)`	= 6	최댓값 9.0 이 있는 위치
`count(a > 3.0)`	= 3	조건을 만족하는 개수
`any(a > 8.0)`	= T	하나라도 참인가
`all(a > 0.0)`	= T	모두 참인가

count · any · all 은 마스크 조건식과 결합할 때 강력한 효과를 발휘한다.

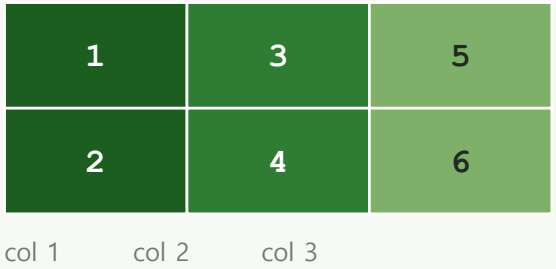
8.3 형상 함수

reshape 는 열부터 채운다

```
m = reshape([1, 2, 3, 4, 5, 6], [2, 3])
```

1차원 원본

2 × 3 행렬 (열 우선으로 채워짐)




```
sum(m) = 21          sum(m, dim=1) = 3  7  11          sum(m, dim=2)
= 9  12
```

dim 인수는 그 차원을 없앤다

sum(m, dim=1) 은 행 방향으로 축소해 각 열의 합 [3, 7, 11] 을, sum(m, dim=2) 는 열 방향으로 축소해 각 행의 합 [9, 12] 를 돌려준다.

[흔한 실수] 행 우선으로 착각

출력이 표처럼 보인다고 [[1,2,3],[4,5,6]] 이 아니다. Fortran 은 열 우선이라 [[1,3,5],[2,4,6]] 이 된다. 행 우선처럼 채우려면 transpose 하거나 reshape 의 order 인수를 쓴다.

8.3 선형대수 함수

외부 라이브러리 없이 바로 쓴다

`matmul(a, b)`

행렬 곱셈

(l × m) 행렬과 (m × n) 행렬을 곱해 (l × n) 크기의 행렬을 반환한다.

`dot_product(u, v)`

벡터 내적

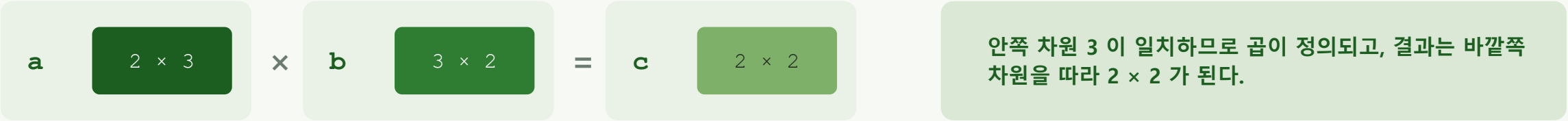
크기가 같은 두 1차원 배열을 받아 각 원소의 곱을 모두 더한 스칼라를 반환한다.

`transpose(a)`

전치

2차원 배열의 행과 열을 서로 맞바꾼다.

행렬 곱이 성립하는 조건 — 안쪽 차원이 같아야 한다



안쪽 차원 3 이 일치하므로 곱이 정의되고, 결과는 바깥쪽 차원을 따라 2 × 2 가 된다.

중첩 do 루프로 직접 구현하지 않고 내장 함수를 쓰면 수치해석적 의도가 명확해지고, 이미 최적화된 선형대수 라이브러리를 내부적으로 호출하므로 속도에서도 큰 이점을 얻는다.

8.3 [예제]

matmul · dot_product · transpose

```
a = reshape([1.0, 2.0, 3.0, 4.0, 5.0, 6.0], [2, 3])
b = reshape([1.0, 0.0, 1.0, 0.0, 1.0, 1.0], [3, 2])

c  = matmul(a, b)          ! (2x3) x (3x2) -> (2x2)
at = transpose(a)          ! 형상이 (3x2) 로

print *, dot_product([1.0,2.0,3.0], [4.0,5.0,6.0])
```

```
matmul(a, b):
     6.0     8.0
     8.0    10.0
dot_product(u, v) =    32.0
transpose(a):
     1.0     2.0         3.0     4.0         5.0     6.0
```

행렬 곱셈 조건

a(2×3) 와 b(3×2) 는 안쪽 차원 3 이 일치하므로 연산이 정의되며, 결과 c 는 (2×2) 형상을 갖는다.

벡터 내적

1×4 + 2×5 + 3×6 = 32.0 이라는 스칼라 결과를 돌려준다.

효율성

이미 최적화된 선형대수 라이브러리를 내부적으로 호출하므로 연산 속도에서 큰 이점이 있다.

8.3 [흔한 실수]

a * b 는 행렬 곱이 아니다

```
a = reshape([1.0, 2.0, 3.0, 4.0], [2, 2])
b = reshape([5.0, 6.0, 7.0, 8.0], [2, 2])
```

a * b

요소별 곱 (element-wise)

5.0	12.0	21.0	32.0

같은 위치의 원소끼리 곱한다. 정사각 행렬이라 형상이 맞더라도 이것은 행렬 곱이 아니다.

matmul(a, b)

행렬 곱 (matrix product)

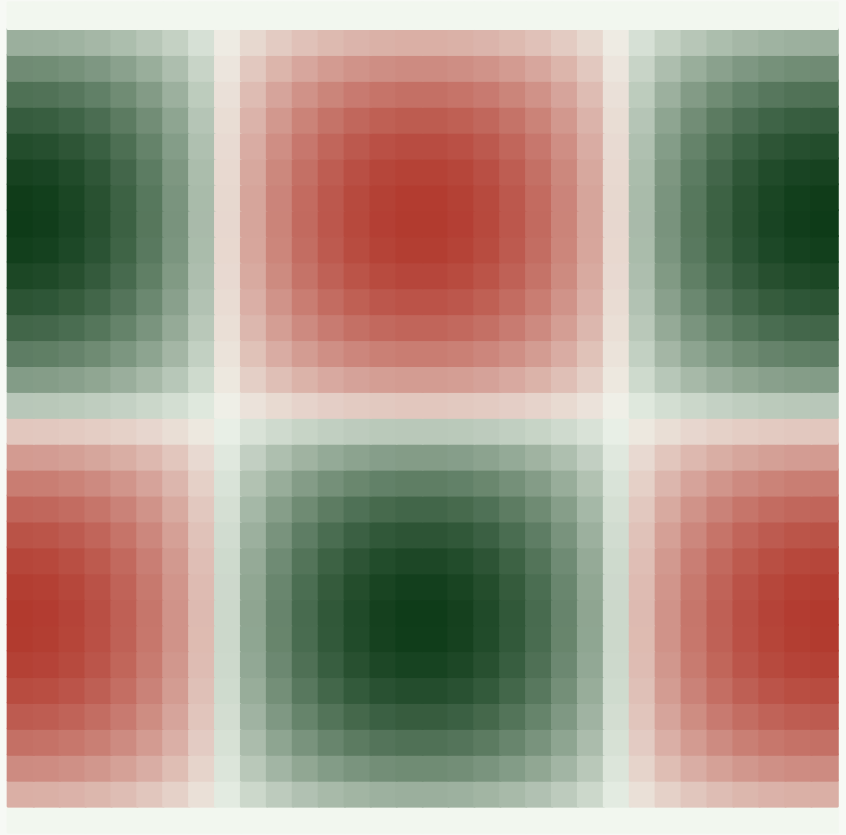
23.0	34.0	31.0	46.0

선형대수의 행렬 곱셈이다. 행렬 곱이 필요하면 반드시 matmul을 쓴다.

두 결과는 전혀 다르다 — 같은 형상의 두 정사각 행렬에서도 마찬가지다.

8.3 [예제]

행렬 곱 결과를 히트맵으로



m(i, j) = u(i) × v(j) **u = sin, v = cos**

```
do i = 1, n
   u(i, 1) = sin(2.0 * pi * real(i - 1) / real(n - 1))
end do

m = matmul(u, v)     ! (n x 1) x (1 x n) -> (n x n)
```

외적(outer product)

열벡터 u(n×1) 와 행벡터 v(1×n) 를 matmul 로 곱하면 (n×n) 행렬이 나온다. 각 원소는 m(i, j) = u(i) × v(j) 다.

바둑판처럼 대칭 배열된 무늬

sin 과 cos 의 주기적 상호작용이 행렬 곱으로 결합되면서 네 개의 마루(붉은색, 양)와 골(초록색, 음)이 규칙적으로 교차한다. 숫자 텍스트로 볼 때보다 데이터의 구조와 경향성이 훨씬 명확하게 드러난다.

오류 학습

안쪽 차원 불일치와 스칼라 마스크

matmul 의 안쪽 차원이 다르다

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

c = matmul(a, b)
! 안쪽 차원 3 과 2 가 다르다
```

```
Error: Different shape on dimension 2 for
  argument 'matrix_a' and dimension 1 for
  argument 'matrix_b' ... for intrinsic matmul
```

해결 방법

a 는 (2×3) 이라 안쪽 차원이 3, b 는 (2×2) 라 안쪽 차원이 2 다. b 의 선언을 real :: b(3, 2) 로 고치면 (2×3)·(3×2) 형태의 정상적인 행렬 곱이 성립한다.

where 에 스칼라 조건을 넣었다

```
integer :: n = 3
real :: y(5)

where (n > 0)       ! 스칼라 조건
   y = 0.0
end where
```

```
Error: WHERE/ELSEWHERE clause at (1)
  requires a LOGICAL array
```

해결 방법

n > 0 은 참·거짓 하나뿐인 스칼라 논리값이다. where 는 "어느 요소에 대입할지"를 요소별로 판별하므로, 대입 대상과 같은 형상의 배열 조건식(예: where (y > 0.0))으로 고쳐야 한다.

8장 정리

요약

연산과 조건

- 전체 배열 연산: 형상이 정합한 배열끼리 + − * / ** 를 쓰면 요소별로 계산된다. 루프가 필요 없다.
- 스칼라는 모든 요소에 확장된다(스칼라 확장).
- 정합 규칙: 연산·대입에 참여하는 배열은 형상이 같아야 한다. 다르면 컴파일 오류.
- 요소별 함수: sin · sqrt · abs 등 수학 내장 함수는 배열에 그대로 적용된다.
- where 구문: 마스크가 참인 요소에만 대입. elsewhere 로 나머지·추가 조건을 처리한다.

내장 함수

- 축약 함수: sum · product · maxval · minval · maxloc · count · any · all.
- dim 인수로 특정 차원만 줄일 수 있다 — sum(m, dim=1)은 열별 합.
- 형상 함수: reshape(source, shape) 는 열 우선으로 채운다.
- 선형대수: 행렬 곱은 matmul, 내적은 dot_product, 전치는 transpose.
- a * b(요소별 곱)와 matmul(a, b)(행렬 곱)를 혼동하지 않는다.

한 줄 요약: 루프를 쓰기 전에, 배열 전체를 한 번에 다룰 방법이 있는지 먼저 생각한다.

연습 문제

1. 크기 5인 실수 배열 a, b 를 채우고 a+b · a−b · a*b · a/b 를 전체 배열 연산으로 계산해 출력하라.

2. 섭씨 배열을 화씨로 변환하는 식을 전체 배열 연산 한 줄로 적용하고, 실수 리터럴을 쓴 이유를 설명하라.

3. 크기 8인 정수 배열에서 sum · maxval · minval · count(a > 0) 를 구해 출력하라.

4. 크기 10인 실수 배열에서 where 만으로 음수 요소를 절댓값으로 바꿔라 (abs 사용 금지).

5. 정수 12개를 reshape 로 [3, 4] 배열에 담아 행 단위로 출력하고 열 우선으로 채워졌음을 확인하라.

6. 크기 4인 두 벡터의 내적을 dot_product 로 구하고, do 루프로 누적한 값과 같은지 비교하라.

7. 크기 n 인 실수 배열을 z-점수로 표준화하고, 평균이 0, 표준편차가 1에 가까움을 확인하라.

8. 2×3 과 3×2 행렬을 matmul 로 곱하고, 삼중 do 루프 결과와 maxval(abs(...)) 로 검증하라.

9. 점수 배열을 where / elsewhere 로 구간 분류하라 (90 이상 3, 70 이상 2, 그 외 1).

10. 4×5 행렬에서 sum(m, dim=1) 과 sum(m, dim=2) 를 구하고, 총계가 sum(m) 과 같은지 확인하라.

다음 장에서는 동적 배열 — allocatable 과 할당·해제, 그리고 자동 재할당을 다룬다.