제 3 부 — 배열

9장

동적 배열과 메모리 할당

Memory on Demand

allocatable · allocate · deallocate · 자동 재할당

RUNTIME SIZING

```
real, allocatable &
     :: v(:)

read *, n
allocate(v(n))

! ... 연산 ...

deallocate(v)
```

학습 목표

이 장에서 익히는 두 가지

01

동적 할당의 마스터

allocatable 속성과 allocate · deallocate 구문을 사용하여 실행 중에 메모리를 유연하게 확보하고 해제한다.

02

자동 재할당 활용

현대 Fortran의 강력한 편의 기능인 할당가능 대입(allocatable assignment) 시 발생하는 자동 재할당 메커니즘을 이해하고 안전하게 구사한다.

실행해 보기 전에는 데이터의 크기를 짐작할 수 없는 상황 — 고성능 수치 해석 환경에서는 이것이 일반적이다.

도입

왜 실행 중에 크기를 정해야 하는가

정적 배열 (static array)

```
real :: v(100)
```

컴파일 단계에서 크기가 고정된다. 프로그램을 빌드하기 전에 필요한 원소 수를 설계자가 미리 완벽하게 알고 있어야 한다.

동적 배열 (dynamic array)

```
real, allocatable :: v(:)
```

선언할 때는 차원(rank) 정보만 지정하고 실제 크기는 비워 둔 채, 실행 시점에 메모리를 부여받는다.

현실의 상황

관측 데이터 처리 프로그램은 사용자가 실행한 뒤에야 n 개의 관측 표본을 입력받는다. 그다음 실행 때는 표본 수 n 이 어떻게 달라질지 예측할 수 없다. 대규모 물리 시뮬레이션에서는 필요한 시점에만 대형 배열을 메모리에 올리고 사용 후 즉시 제거해야 시스템 자원을 아낄 수 있다.

선언 시점에 지정한 콜론의 개수(계수)는 도중에 바꿀 수 없지만, 각 차원의 실제 길이는 실행 중에 자유롭게 결정할 수 있다.

9.1 allocatable 속성

동적 배열을 다루는 네 단계

```
real :: v(100)                ! 컴파일 시점에 길이 100으로 고정되는 정적 배열
real, allocatable :: v(:)     ! 1차원(rank 1), 크기 미정
real, allocatable :: m(:,:)   ! 2차원(rank 2), 크기 미정
```

1 선언

```
real, allocatable :: v(:)
```

동적 할당 대상임을 컴파일러에 알린다. 괄호 안에 차원의 수만큼 콜론을 적는다.

2 할당

```
allocate(v(1:n), stat=status)
```

실제 물리 메모리를 확보한다. 하한을 생략하면 인덱스는 1부터 n 까지 부여된다.

3 해제

```
deallocate(v, stat=status)
```

사용이 끝난 메모리를 운영체제에 반환한다. 누락하면 메모리 누수가 발생한다.

4 점검

```
allocated(v)
```

현재 메모리에 남아 있는지 확인한다. 할당 상태면 참, 미할당이면 거짓.

선언 직후의 allocatable 변수는 아직 실제 메모리 주소를 갖지 못한 미할당(unallocated) 상태다.

9.1 알아두기

메모리 보호와 예외 처리 안전 규칙

이중 할당 · 미할당 해제 금지

이미 메모리를 부여받은 배열에 또 allocate 를 호출하거나, 할당된 적 없는 배열을 deallocate 하려 하면 런타임 에러로 즉시 강제 종료된다.

```
if (.not. allocated(v)) allocate(v(n))
if (allocated(v)) deallocate(v)
```

상태 변수 stat= 을 통한 예외 제어

메모리 확보에 성공하면 지정한 정수 변수에 0 이 저장되고, 시스템 메모리 부족 등으로 실패하면 0 이 아닌 에러 코드가 대입된다.

```
allocate(v(n), stat=ierr)
if (ierr /= 0) stop 1
```

stat= 을 쓰면 비정상 종료를 막을 수 있다

프로그램이 갑자기 죽는 대신 "메모리가 부족합니다" 같은 메시지를 출력하고 안전하게 정상 종료(stop 1)하도록 유도할 수 있다.

[흔한 실수] 큰 메모리를 다루거나 크기를 사용자 입력·파일에서 받는 코드에서는 stat= 을 받는 습관을 들인다. 생략하고 실패하면 프로그램은 곧바로 비정상 종료한다.

9.1 [예제]

실행 단계에서 메모리 크기 정하기

```
integer :: n, i
real, allocatable :: v(:)

read *, n                ! 크기를 실행 시점에 결정

allocate(v(n))
do i = 1, n
   v(i) = real(i)**2
end do

print *, "allocated size:", size(v)
deallocate(v)
```

```
!echo 5 | ./dynamic_basics
```

```
 allocated size:  5
 elements:  1.0  4.0  9.0  16.0  25.0
```

실행 시점의 유연한 크기 결정

선언할 때만 해도 크기는 백지 상태다. read *, n 으로 5 라는 숫자를 읽는 순간 allocate(v(5)) 가 작동하며 비로소 5개의 실수를 담을 공간이 확보된다.

재컴파일이 필요 없다

echo 10 이나 echo 1000 으로 바꿔 주면, 코드를 전혀 수정하지 않고 재컴파일조차 없이 길이 10, 1000 인 배열이 자동으로 생성된다.

깔끔한 마무리

계산이 끝난 뒤 deallocate(v) 로 시스템에 메모리를 즉시 돌려준다.

9.1 [예제]

할당 실패에 대비하기 — stat=

```
integer(i8) :: request
integer :: ierr
real, allocatable :: big(:)

request = 10_i8**11        ! 일부러 너무 크게

allocate(big(request), stat=ierr)
if (ierr /= 0) then
   print *, "allocation failed, stat =", ierr
   stop 1
end if
```

```
 allocation failed, stat =        5020
 STOP 1
```

gfortran 기준 실패 코드는 5020 이다 — 값 자체보다 "0 이 아니다"가 중요하다.

의도적인 대량 요청

10^{11} (1,000억) 개를 4바이트 실수형으로 확보하려면 약 400GB 가 필요하다. 일반적인 계산 서버나 코랩 환경의 한계를 훌쩍 넘는 크기다.

안전한 예외 처리

stat= 없이 썼다면 Segmentation fault 나 Out of memory 로 비정상 종료했을 것이다. stat=ierr 덕분에 에러 코드를 받아 두고 제어권을 계속 유지한다.

시스템 전파 제어

if (ierr /= 0) 이 실패를 포착해 원인을 출력한 뒤 stop 1 로 결함을 알리며 방어 체계 안에서 종료한다.

9.1 [예제]

allocated 로 상태 확인하기

```
real, allocatable :: vals(:)

print *, "before:", allocated(vals)

allocate(vals(3))
vals = [1.0, 2.0, 3.0]

if (allocated(vals)) deallocate(vals)
```

```
before                  F

after allocate          T   size: 3

after deallocate        F
```

외워 둘 관용구

if (allocated(vals)) deallocate(vals) 는 수치 해석 프로그래밍에서 비정상 종료를 막기 위해 관용적으로 쓰는 코드다. 할당되어 있을 때만 해제하므로 런타임 에러를 방지한다.

[흔한 실수] 런타임 에러를 부르는 셋

- 해제 없이 allocate 중복 호출
- 미할당 배열을 deallocate
- 미할당 배열에 vals(1) = 0.0 처럼 원소 접근

상태가 모호할 때는 반드시 allocated() 로 먼저 검증한다.

9.2 할당 가능 대입

대입만으로 메모리가 맞춰진다

```
real, allocatable :: u(:)
u = [1.0, 2.0, 3.0]      ! 별도의 allocate 없이 u 는 크기 3으로 자동 할당
```

1 좌변이 미할당

우변 배열의 형상과 크기를 고스란히 물려받아 시스템 메모리가 자동으로 최초 할당된다.

최초 할당

2 형상이 다름

기존에 할당된 메모리 공간을 즉시 해제한 뒤, 우변의 새 규격에 맞춰 공간을 완전히 재할당한다.

재할당

3 형상이 일치

추가 공정 없이 기존 공간을 그대로 유지한 채 우변의 데이터 값만 메모리에 고속 복사한다.

값만 복사

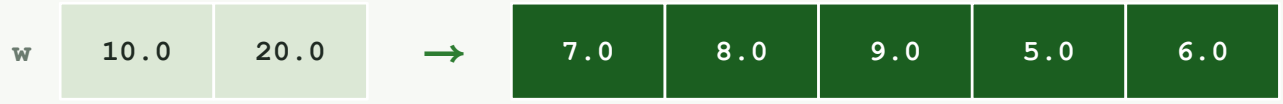
이런 메커니즘을 할당 가능 대입 시의 자동 재할당(reallocation on assignment)이라 한다 — Fortran 2003 이후의 표준이다.

9.2 [예제]

대입만으로 메모리 재할당

```
real, allocatable :: u(:), w(:)

u = [1.0, 2.0, 3.0, 4.0]          ! 크기 4로 자동 할당

w = [10.0, 20.0]                  ! 크기 2

w = [7.0, 8.0, 9.0, 5.0, 6.0]     ! 크기 5로 자동 재할당
```

```
u size: 4    u:  1.0  2.0  3.0  4.0
w size: 2
w size: 5    w:  7.0  8.0  9.0  5.0  6.0
```



미할당 상태에서의 최초 할당

u 와 w 는 선언 단계에서 공간을 갖지 못한 채 시작한다. 우변의 원소 개수를 판정해 그 크기로 최초 할당된다.

유연한 자동 재할당

이미 크기 2인 w 에 원소 5개를 대입하면, 형상이 불일치함에도 크기 5인 새 배열로 자동 갱신된다.

오버헤드의 감소

크기 변화가 일어나는 동안 메모리 관리를 위한 조건문이나 수동 제어 구문이 전혀 개입하지 않았다.

9.2 [흔한 실수]

자동 재할당은 allocatable 에만 작동한다

```
real :: fixed(3)

fixed = [1.0, 2.0]
```

```
Different shape for array assignment at (1)
  on dimension 1 (expected 3 and got 2)
```

컴파일 단계에서 차단된다

좌변이 고정 크기 배열이면 형상 불일치를 컴파일러가 감지해 막는다. 대입 연산을 할 때는 좌변이 고정 크기인지 재할당 가능한 동적 배열인지 명확히 인지해야 한다.

좌변이 allocatable 이면

```
real, allocatable :: flex(:)
flex = [1.0, 2.0]     ! 크기 2로 재할당
```

같은 대입이 오류 없이 통과하며 크기가 자동으로 맞춰진다.

표준 모드로 컴파일한다

자동 재할당은 Fortran 2003 이후 표준이다. 일부 컴파일러나 옛 호환 옵션(gfortran 의 -fno-realloc-lhs)에서는 이 동작이 비활성되어 좌변 크기가 유지되고 값이 잘릴 수 있다. -std=f2018 로 컴파일하면 안전하다.

고정 크기 배열이라면 우변의 데이터 개수를 반드시 일치시켜야 한다.

9.2 [예제]

배열 크기 동적 확장하기 (append)

```
integer, allocatable :: collected(:)

allocate(collected(0))              ! 크기 0인 빈 배열로 시작

do i = 1, 20
   sq = i*i
   if (mod(sq, 2) == 0) then
      collected = [collected, sq]   ! 한 칸씩 늘린다
   end if
end do
```

```
count: 10
values:  4  16  36  64  100  144  196  256  324  400
```

collected 가 자라나는 모습

내부에서 일어나는 일

collected = [collected, sq] 가 실행되면 기존 배열의 모든 원소와 새 원소 하나를 결합한 임시 배열이 먼저 만들어진다. 그 뒤 자동 재할당으로 좌변이 한 칸 확장되며 값이 대입된다.

최종 개수를 몰라도 된다

수집될 데이터의 개수를 미리 예측할 수 없으므로 크기 0인 빈 배열로 시작한다. 조건에 맞는 원소만 골라 차례대로 붙여 나간다.

대량 데이터라면

매번 전체를 복사하므로 원소 수가 많아지면 비용이 급격히 커진다. 이럴 때는 용량을 두 배씩 늘려 두고 move_alloc 으로 옮기는 기법을 쓴다.

9.2 알아두기

자동 해제는 언제 일어나는가

지역 allocatable 변수는 스코프를 벗어날 때 자동으로 해제된다

다만 그 시점은 반복문 블록의 끝이 아니라, 해당 변수가 선언된 프로그램 단위(주 프로그램이나 프로시저)가 완전히 끝나는 시점이다. 포인터는 이 자동 해제 대상이 아니다 (13장).

× **반복문 블록의 끝**

여기서는 해제되지 않는다. 다음 회차에서 다시 allocate 를 만나면 "이미 할당됨" 오류가 난다.

○ **프로그램 단위의 끝**

여기서 비로소 자동 해제가 작동한다. 주 프로그램이 종료될 때 남은 메모리는 정리된다.

그래서 반복문 안에서 매 회차 allocate 를 한다면, 회차 끝에서 반드시 deallocate 하거나 할당 가능 대입을 쓴다.

프로시저 안에서는 가정 형상 더미 x(:) 가 호출마다 형상을 받고, 자동 배열 work(size(x)) 가 임시로 생긴다 — 정식 문법은 10~12장에서 다룬다.

9.3 배열 크기와 정밀도

격자를 촘촘히 하면 언제나 정확해질까

배열의 크기를 키워 격자를 촘촘하게 하는 것만으로는 정확도가 무한정 향상되지 않는다. 수치 해석에서는 수학적 알고리즘 자체의 오차 수준과 하드웨어의 부동소수점 정밀도 한계선이 복합적으로 맞물려 최종 정확도를 결정하기 때문이다.

```
integral of 4/(1+x^2) from 0 to 1  =  pi
```

사다리꼴 법칙으로 위 정적분을 계산한다. 구간 수 n 을 2부터 시작해 매 루프마다 2배씩 키우며(n = 2, 4, 8, …, 65536), 단정밀도와 배정밀도로 나란히 수행한다.

매 회차 크기가 달라진다

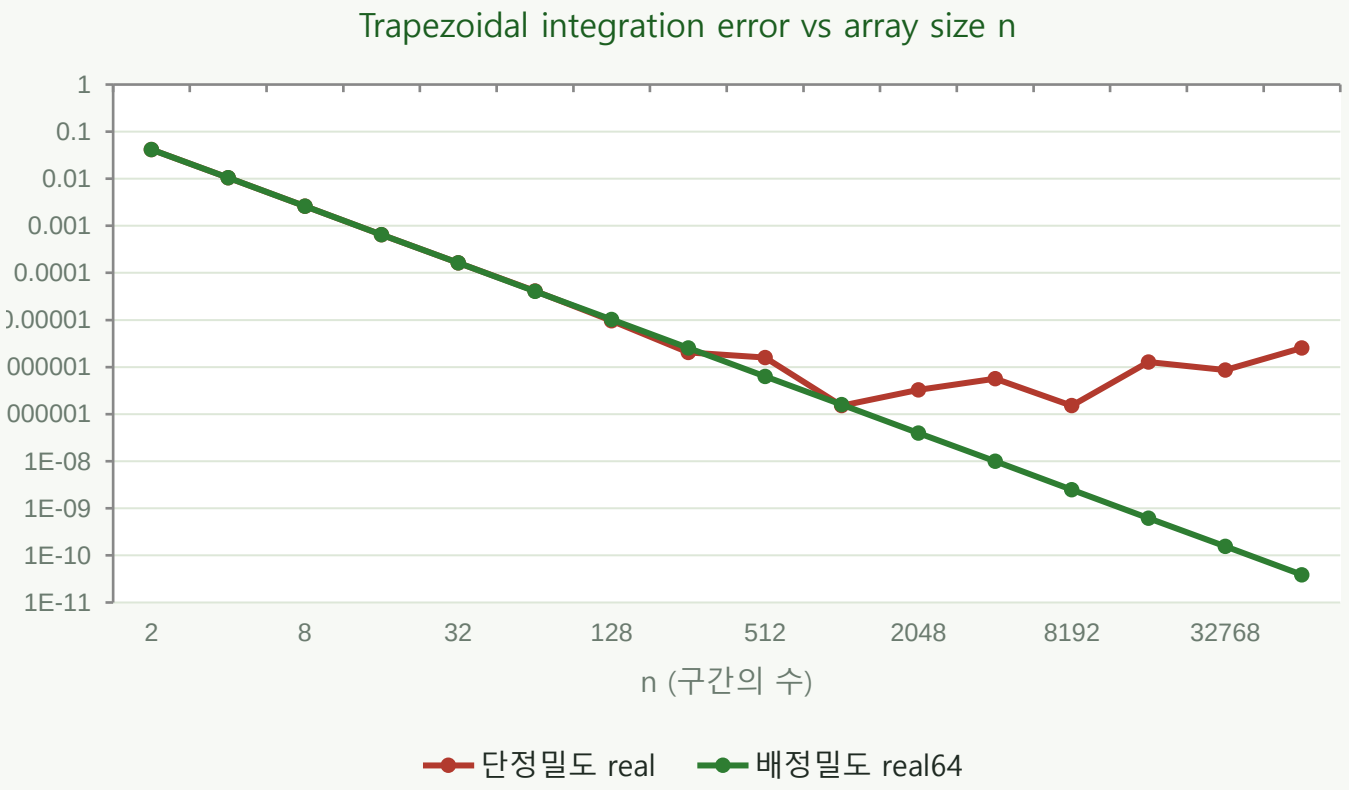
격자점 개수에 맞춰 매번 길이 n+1 인 가변 배열이 필요하다. 명시적 해제를 한 줄이라도 빠뜨리면 k=2 진입 순간 중복 할당 예외가 발생한다.

```
do k = 1, 16
   n = 2**k
   allocate(xs(n+1), ys(n+1), xd(n+1), yd(n+1))

   hd = 1.0_dp / real(n, dp)
   do i = 0, n
      xd(i+1) = real(i, dp) * hd
   end do
   yd = 4.0_dp / (1.0_dp + xd**2)

   approx_d = hd * (sum(yd) &
              - 0.5_dp * (yd(1) + yd(n+1)))

   deallocate(xs, ys, xd, yd)
end do
```

수식과 sum 은 8장의 전체 배열 연산이라 루프 없이 처리된다.

9.3 [예제]

격자를 키울수록 정확해지는가 — 로그 그래프



배정밀도 — 기울기 −2 의 직선

사다리꼴 법칙의 알고리즘 오차가 $O(1/n^2)$ 을 따르므로, n 을 2배 키우면 오차가 대략 1/4 로 정확히 줄어든다. 10^{-11} 영역까지 부드럽게 하강한다.

단정밀도 — 바닥에서 요동친다

$n \approx 256$ 까지는 배정밀도와 겹쳐 하강하다가, 약 10^{-6} 부근에서 멈추고 들쭉날쭉해진다. 반올림 오차가 알고리즘 오차를 넘어선 것이다.

이 불안정 지점부터는 격자 배열의 크기를 아무리 키워도 정확도가 조금도 개선되지 않는다 — 오히려 n = 65536 에서는 오차가 거꾸로 증가한다.

9.3 결과 해석

숫자로 확인하는 정밀도의 한계선

n	단정밀도 오차	배정밀도 오차	해석
2	4.15927490E-02	4.15926536E-02	두 정밀도가 거의 같은 오차 수준
8	2.60415872E-03	2.60415910E-03	둘 다 감소하며 수렴성을 증명
256	2.05834443E-06	2.54313151E-06	여전히 유의미한 결합 양상 유지
4096	5.64259938E-07	9.93410953E-09	단정밀도가 정체, 격차가 벌어진다
65536	2.53518159E-06	3.88196142E-11	단정밀도는 거꾸로 증가 — 수치적 불안정성

고성능 과학 계산과 기상 시뮬레이션에서 유연한 동적 배열 확보와 정밀한 종류(kind) 선택이 왜 함께 필수적인지 보여 준다.

오류 학습

반복문 안에서 deallocate 를 빠뜨리면

```
real, allocatable :: x(:)

do k = 1, 3
   n = 2*k
   allocate(x(n))      ! 2회차: x 는 아직 할당된 상태
   x = real(k)
   print *, "k =", k, " size =", size(x)
   ! deallocate(x) 가 없다
end do
```

```
 k =  1  size =  2
At line 8 of file loop_no_dealloc.f90
Fortran runtime error: Attempting to allocate
  already allocated variable 'x'
```

해결 — 회차 끝에서 명시적으로 반환한다

```
   deallocate(x)       ! 다음 회차의 신규 할당에 대비한다
```

첫 회차는 정상이다

k = 1 에서는 x 가 미할당 상태였으므로 allocate(x(2)) 가 정상 작동해 크기 2 의 격자가 생성되고 값이 출력된다.

두 번째 회차에서 멈춘다

자동 해제는 반복문 블록의 끝이 아니라 프로그램 단위가 끝나는 시점에 작동한다. k = 2 에서 allocate(x(4)) 를 만나는 순간, x 는 여전히 크기 2 의 메모리를 점유한 상태다.

대안 — 할당 가능 대입

매 회차 x = 어떤 배열식 으로 대입하면 자동 재할당이 크기를 맞춰 주므로 명시적 해제 없이도 안전하다.

9장 정리

요약

할당과 해제

- 동적 배열 선언: type, allocatable :: a(:) — 콜론 개수가 계수(rank).
- 할당·해제: allocate(a(n)), deallocate(a). 안전 장치는 allocate(a(n), stat=ierr).
- 상태 점검: allocated(a) → 논리값. if (allocated(a)) deallocate(a) 관용구.
- 지역 allocatable 변수는 스코프를 벗어날 때 자동 해제된다 (포인터는 그렇지 않다 — 13장).
- 반복문에서 매 회차 allocate 한다면 회차 끝에서 deallocate 하거나 할당 가능 대입을 쓴다.

자동 재할당

- 할당 가능 대입: allocatable 좌변은 우변 형상에 맞춰 자동으로 (재)할당된다.
- 정적(고정) 좌변에는 자동 재할당이 적용되지 않는다 — 형상이 다르면 오류.
- 배열 키우기: 소량은 v = [v, x], 대량은 용량 두 배 + call move_alloc(tmp, v).
- 프로시저 안에서는 가정 형상 더미 x(:) 가 호출마다 형상을 받고, 자동 배열 work(size(x)) 가 임시로 생긴다 (10~12장).

한 줄 요약: 크기를 모르면 allocatable 로 선언하고, 할당했으면 반드시 돌려준다.

연습 문제

1. 정수 n 을 입력받아 1~n 을 담는 allocatable 배열을 만들고 합을 출력하라. deallocate 도 호출하라.

2. 길이 미정의 실수 배열 a(:) 를 할당 가능 대입으로 채운 뒤 size(a) 와 maxval(a) 를 출력하라.

3. allocatable 배열 x 에 대해 할당 전·후·해제 후의 allocated(x) 값을 각각 출력하라.

4. 행 수와 열 수를 입력받아 m(:,:) 을 allocate 하고, 0.0 으로 채운 뒤 shape(m) 을 출력하라.

5. 너무 큰 크기를 일부러 요청하고 stat= 을 받아, 성공이면 크기를 실패면 안내 문구를 출력하라.

6. 빈 배열에서 시작해 1~30 중 3의 배수만 v = [v, k] 로 모아 개수와 값을 출력하라.

7. 길이 n+1 배열에 $x_i = i/n$ 을 채우고 $y = \sin(2\pi x)$ 를 wave.csv 로 저장한 뒤 선그래프를 그려라.

8. v 를 크기 4로 할당해 채운 뒤 크기 7짜리 값을 대입하고, 재할당 전후의 size(v) 를 출력하라.

9. 용량을 두 배가 아니라 약 1.5배(올림)로 늘리도록 바꾸고, 100개를 담는 동안 재할당 횟수를 세어라.

10. 두 allocatable 배열 a, b 를 c = [a, b] 로 이어 붙이고 size(c) 가 size(a)+size(b) 인지 확인하라.

다음 장에서는 프로시저 — 함수와 서브루틴, 인수 전달과 intent 를 다룬다.