제 5 부 — 자료 구조와 파일

13장
파생형과 포인터

Building Your Own Types

type · % 접근 · 형 생성자 · pointer / target · 연결 리스트

YOUR OWN TYPE

```
type :: node
   integer :: value
   type(node), pointer &
        :: next => null()
end type node

p => t
p = 99.0
```

학습 목표

이 장에서 익히는 네 가지

01

파생형 정의

파생형(derived type)으로 나만의 자료 구조를 만든다.

02

구조 조립

형 생성자와 중첩 구조로 복잡한 자료를 표현한다.

03

포인터 기초

pointer · target 과 연관을 기초 수준에서 다룬다.

04

연결 구조

포인터로 간단한 연결 구조를 만든다.

파생형과 포인터를 유기적으로 결합할 때 Fortran 의 구조적 데이터 처리 능력이 온전히 발휘된다.

도입

서로 다른 자료형이 한 덩어리로 묶여 있을 때

1 물리계의 입자

하나의 입자는 공간상의 위치 (x, y) 와 물리적 속도 (vx, vy) 를 동시에 지닌다.

```
real :: pos(2), vel(2)
```

2 학생 정보

학생 한 명의 데이터는 이름(문자형), 학번(정수형), 성적(실수형)이 한 덩어리로 결합되어 존재한다.

```
character / integer / real
```

따로 저장하면 무엇이 문제인가

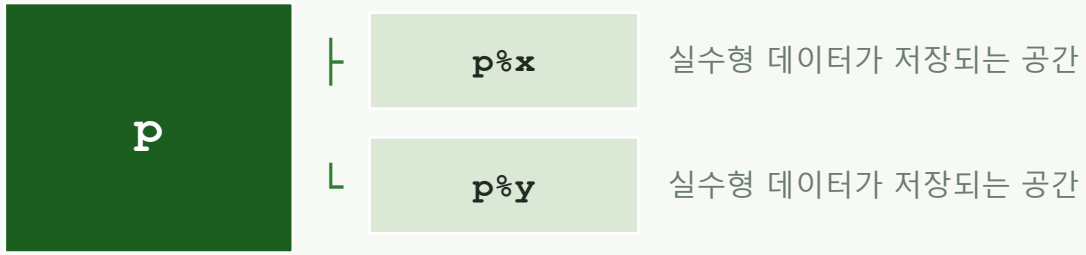
위치는 위치대로, 속도는 속도대로 서로 다른 배열에 분산해 두면 "3번 입자의 속도"를 조회하거나 수정할 때마다 각 배열의 인덱스를 일일이 수동으로 일치시켜야 한다. 코드가 복잡해지고 논리적 실수가 잦아진다.

파생형은 서로 다른 자료형의 변수들을 하나의 새로운 자료형으로 통합한다 — C 의 struct, 파이썬의 dataclass 와 같은 개념이다.

type ... end type, 그리고 퍼센트(%) 접근

```
type :: 형-이름
   구성요소-자료형 :: 구성요소-이름 [= 초깃값]
   ! ... 추가 구성 요소 선언
end type [형-이름]
```

변수 p 의 메모리 구조



중첩 구조에서는 c%center%x 처럼 % 를 사슬처럼 연이어 연결해 탐색한다.

설계도와 실체는 다르다

type :: point2d ... end type 은 새로운 구조의 설계도를 알려주는 선언부다. 이 단계에서는 메모리가 할당되지 않는다.

메모리가 잡히는 시점

type(point2d) :: p 문장을 만났을 때 비로소 설계도에 따라 실제 변수 공간이 구축된다.

인수 전달이 간결해진다

밀접하게 연관된 값들이 하나의 변수 영역에 통합되므로, 프로시저로 넘길 때 여러 인수 대신 파생형 변수 하나만 주고받으면 된다.

13.1 [예제]

2차원 평면 위의 점 표현

```
program point_demo
   use, intrinsic :: iso_fortran_env, only: real64
   implicit none

   type :: point2d
      real(real64) :: x
      real(real64) :: y
   end type point2d

   type(point2d) :: p
   real(real64)  :: dist

   p%x = 3.0_real64
   p%y = 4.0_real64
```

```
point = (3.00, 4.00)
distance from origin = 5.00
```

선언부 — 설계도

type :: point2d ... end type point2d 는 새 구조의 형태만 알려 준다. 메모리는 아직 잡히지 않는다.

변수 선언 — 실체

type(point2d) :: p 를 만나야 비로소 x 와 y 를 담을 공간이 실제로 만들어진다.

% 로 개별 공간에 접근

p%x 와 p%y 로 값을 나누어 보관하고, 원점과의 유클리드 거리를 안전하게 계산한다.

13.2 형 생성자

한 줄로 모든 값을 초기화한다

형 생성자의 이름은 사용자가 정의한 파생형의 이름과 완전히 동일하다.

순서 지정 방식

Positional Association

정의할 때 나열한 구성 요소의 순서에 맞추어 값을 차례대로 기입한다.

```
type-name(value1, value2, ...)
```

키워드 지정 방식

Keyword Association

구성 요소의 이름을 명시하고 = 로 값을 매핑한다. 정의된 순서와 상관없이 안전하게 대입할 수 있어 가독성이 높다.

```
type-name(comp1=value1, comp2=value2, ...)
```

기본 초기화

정의 시점에 구성 요소 옆에 = 초깃값 을 지정해 두면, 변수가 선언될 때 자동으로 그 값이 채워진다.

생략하려면 키워드로

기본 초기화가 설정된 요소는 생성자에서 생략할 수 있다. 단, 일부를 생략할 때는 컴파일러가 위치를 혼동하지 않도록 반드시 키워드 지정 방식으로 호출해야 한다.

[흔한 실수] 기본값 없는 요소 누락

기본값이 지정되지 않은 필드를 생성자 인수 목록에서 하나라도 빠뜨리면 오류다 — No initializer for component 'y' given in the structure constructor at (1)

13.2 [예제]

중첩 구조와 기본 초기화

```
type :: point2d
   real(real64) :: x = 0.0_real64
   real(real64) :: y = 0.0_real64      ! 각 구성요소에 기본값
end type point2d

type :: circle
   type(point2d) :: center             ! 중첩 구조
   real(real64)  :: radius = 1.0_real64
end type circle

c1 = circle(center = point2d(2.0_real64, 1.0_real64), &
            radius = 3.0_real64)
c2 = circle()          ! 모든 구성요소가 기본값

area = pi * c1%radius**2
```

```
c1: center=(2.00,1.00), r=3.00
c1 area = 28.274
c2: center=(.00,.00), r=1.00
```

c1 — 키워드로 명시 초기화

(2.0, 1.0) 좌표에 반지름 3.0. 내부의 point2d 구조도 생성자 안에서 중첩 연동되어 초기화된다. 면적은 π×3² ≈ 28.27.

c2 — 빈 생성자

circle() 로 인수를 모두 생략했으므로 설계 단계에서 지정한 규격대로 중심 (0.0, 0.0), 반지름 1.0 을 획득한다.

% 를 체인으로 연결

반지름은 c1%radius 로 바로 가리키고, 중심점 내부의 세부 좌표는 c1%center%x 로 두 단계를 거쳐 가리킨다.

13.2 [예제]

파생형 배열과 구성 요소 일괄 연산

```
type :: student
   character(len=20) :: name
   integer           :: id
   real(real64)      :: score
end type student

type(student) :: class(3)

class(1) = student("Kim",  101, 88.5_real64)
class(2) = student("Lee",  102, 92.0_real64)
class(3) = student("Park", 103, 79.5_real64)

avg = sum(class%score) / size(class)
```

```
101 Kim          88.5
102 Lee          92.0
103 Park         79.5
average = 86.67
```

class%score 는 점수만 모은 크기 3의 배열처럼 동작한다

배열 이름에 % 를 붙인다

개별 인덱스를 지정하지 않고 class%score 라고 쓰면, 배열 전체에서 score 값만 골라낸 배열처럼 동작한다.

루프 없이 집계

덕분에 반복문을 구성하지 않고도 내장 함수 sum 과 size 만으로 학급 전체의 평균을 산출할 수 있다 (8장).

유연한 확장성

관리할 데이터가 수천 명으로 늘어도 배열의 선언 크기만 조정하면 통계 계산 코드는 그대로 유지된다.

13.3 포인터 기초

값을 담지 않고, 대상을 가리킨다

포인터는 스스로 값을 저장하지 않고 가리키는 대상의 '별명(alias)' 역할을 한다 — 대상은 target 속성이 명시된 변수이거나, allocate 로 만든 이름 없는 동적 메모리다.

연관 (Association)

```
p => t
```

포인터 대입 연산자 => 로 포인터 p 가 타겟 t 를 가리키도록 설정한다. 이 단계를 거치면 p 는 t 를 참조하는 유효한 별명이 된다.

값 대입 (Assignment)

```
p = v
```

일반 대입 연산자 = 로 포인터 p 가 현재 가리키는 대상에 값 v 를 기록한다. 값은 p 자체가 아니라 t 의 메모리에 들어간다.

별명 효과 (aliasing)

p => t 로 연관시킨 뒤 p = 5 라고 쓰면, 값 5 는 포인터 p 자체가 아니라 p 가 가리키는 t 의 메모리 공간에 기록되어 원본 t 가 5 로 바뀐다. 포인터의 본질적 기능인 동시에 논리적 혼동을 가장 자주 유발하는 원인이다.

[흔한 실수] 독립적인 복사본을 만들 의도였는데 실수로 p => t 라고 쓰면, 나중에 p 를 수정할 때 원본 t 까지 함께 바뀌는 별명 버그가 생긴다.

13.3 상태 관리

포인터는 항상 셋 중 하나의 상태에 있다

○ 연관됨

Associated

메모리상의 유효한 대상 변수나 동적 할당 공간을 올바르게 가리키고 있어 안전하게 접근할 수 있는 상태.

— 해제됨

Disassociated

null() 이나 nullify 로 명시적으로 연결을 끊어 아무것도 가리키지 않는 상태. 안전한 참조 차단이 가능하다.

× 정의되지 않음

Undefined

선언만 하고 초기화하지 않아 메모리 어디를 가리키는지 알 수 없는 극히 위험한 상태.

1 선언 시 즉시 초기화

포인터를 선언할 때 반드시 => null() 을 결합해, '정의되지 않음'이 아닌 명확한 '해제됨' 상태로 출발시킨다. 초기화되지 않은 포인터는 임의의 메모리 주소를 가리키고 있을 수 있다.

2 접근 전 유효성 검사

포인터를 통해 값을 읽거나 쓰기 전에 반드시 associated() 를 조건문과 결합해 대상이 실재하는지 확인하는 방어적 코딩을 체득한다.

[흔한 실수] 초기화하지 않은 포인터는 if (associated(p)) 로 검사하는 것조차 안전하지 않다 — 검사 구문 자체에서 런타임 오류가 날 수 있다.

13.3 문법 정리

연관 제어와 상태 검사

타겟 변수 선언	`자료형, target :: 타겟_변수명`	
포인터 선언 및 초기화	`자료형, pointer :: 포인터_변수명 => null()`	
포인터 연관 (포인터 대입)	`p => t`	포인터 p 가 타겟 변수 t 가 위치한 메모리 주소를 가리키도록 연결을 생성한다.
연관 해제 (Nullify)	`nullify(p)`	포인터 p 의 기존 메모리 연결을 안전하게 끊고 상태를 '해제됨'으로 즉시 변경한다.
연관 여부 확인	`associated(p)`	p 가 유효한 대상을 가리키면 .true., 해제된 상태라면 .false. 를 반환한다.
특정 타겟 검증	`associated(p, t)`	p 가 가리키는 대상이 우측에 지정한 특정 타겟 t 와 정확히 일치하는지 검증한다.

[흔한 실수] target 속성이 없는 평범한 변수를 가리키려 하면 — Error: Pointer assignment target is neither TARGET nor POINTER at (1)

13.3 [예제]

연관 · 역참조 · 해제의 흐름

```
real(real64), target  :: a = 10.0_real64
real(real64), target  :: b = 20.0_real64
real(real64), pointer :: p => null()

p => a                      ! associate p with a
p = 99.0_real64             ! writes through p into a
p => b                      ! re-point to b

print *, associated(p, a)
print *, associated(p, b)
nullify(p)
```

선언 직후	p associated? F
p => a	T, p = 10.0
p = 99.0	a = 99.0 ← 원본이 바뀐다
p => b	p = 20.0
associated(p, a) / (p, b)	F / T
nullify(p)	associated? F

p = 99.0 이 a 를 바꾼다

일반 대입 = 를 쓰면 포인터 자체가 아니라 p 가 역참조하는 대상 a 에 값이 직접 대입된다. p 가 a 의 별명이었기 때문이다.

재연관

p => b 로 새 대상을 가리키면 기존 a 와의 연결 사슬은 자동으로 끊어진다.

두 인수 associated

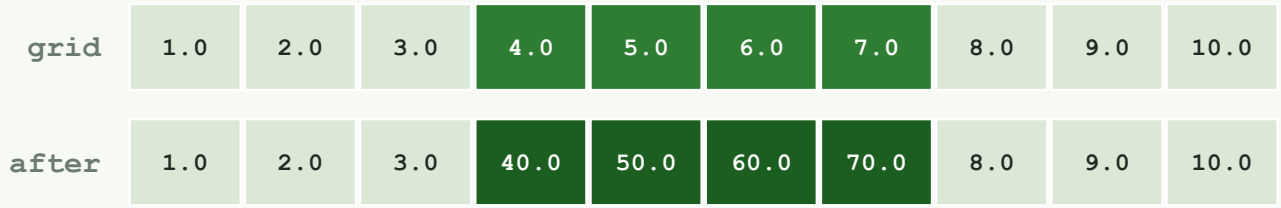
associated(p, a) 와 associated(p, b) 로 p 가 정확히 어떤 타겟과 연동되어 있는지 논리값으로 판정할 수 있다.

13.3 [예제]

배열 구간을 가리키는 창(window)

```
real(real64), target  :: grid(10)
real(real64), pointer :: window(:) => null()

window => grid(4:7)                ! alias elements 4..7
window = window * 10.0_real64      ! modifies grid(4:7)
```



window => grid(4:7) — 이 네 칸만 가리킨다

*window = window * 10.0 한 줄이 원본 grid 의 4~7번 칸을 즉각 10배로 바꾼다.*

복사 없는 접근 (zero-copy)

임시 배열을 할당하고 데이터를 복사하는 과정이 없다. 원본 메모리를 그대로 직접 참조하므로 오버헤드가 없고 연산 속도 면에서 효율적이다.

가독성

grid(4:7) 같은 인덱스 표현식을 매번 기술할 필요 없이 직관적인 이름으로 추상화할 수 있다.

부분 영역 집중 연산

대규모 행렬이나 유체 역학 격자 모델에서 전체 중 특정 관심 도메인만 떼어내어 독립된 격자망처럼 다룰 때 강력하다.

13.4 연결 리스트

배열과 무엇이 다른가

배열 (Array)

연속된 메모리를 점유하므로 인덱스를 통한 빠른 접근이 가능하다. 하지만 중간에 삽입하거나 삭제하려면 이후의 모든 원소를 밀거나 당겨야 해 크기 변경이 비효율적이다.

연결 리스트 (Linked List)

삽입·삭제 시 포인터의 연결 고리만 바꾸면 된다. 메모리가 허용하는 한 노드를 계속 추가할 수 있어, 데이터 개수가 유동적인 환경에서 강력하다.

노드(Node)의 내부 구조

추가 (Push)

새 노드를 생성한 뒤 기존 리스트의 시작(Head) 또는 끝(Tail)에 포인터를 연결해 사슬을 확장한다.

순회 (Traversal)

Head 부터 출발해 각 노드의 next 를 타고 마지막(null)에 도달할 때까지 차례로 이동한다.

삭제 (Delete)

대상의 이전 노드가 대상의 '다음 노드'를 직접 가리키도록 우회시킨 뒤, 대상 노드의 메모리를 해제한다.

13.5 자기 참조 포인터

파생형 안에서 자기 자신을 가리킨다

```
type :: node
    real :: value
    type(node), pointer :: next => null()
end type node
```

재귀적 포인터 선언

아직 정의가 끝나지 않은 node 파생형 내부에서 자기 자신을 참조하는 포인터를 선언할 수 있다. 반드시 => null() 로 초기화해 연결의 끝을 안전하게 명시한다.

동적 메모리 관리

실행 중에 allocate(node_pointer) 로 이름 없는 익명 노드를 힙(Heap)에 필요한 만큼 할당하고, 끝나면 deallocate 로 즉각 반환한다.

연결 리스트는 흩어진 데이터를 포인터라는 실로 꿰어 놓은 꾸러미와 같다.

```
subroutine push_front(head, v)
   type(node), pointer, intent(inout) :: head
   integer,             intent(in)    :: v
   type(node), pointer :: new_node

   allocate(new_node)
   new_node%value = v
   new_node%next  => head    ! 새 노드가 옛 head 를 가리킨다
   head           => new_node  ! head 를 새 노드로 옮긴다
end subroutine push_front
```

전방 삽입 (push_front)

allocate 로 힙에 독립된 노드 공간을 개설한 뒤, 새 노드의 next 가 기존 첫 노드를 가리키게 연관시키고, 마지막으로 head 가 가리키는 주소를 새 노드로 바꾼다.

13.5 [예제]

만들고, 순회하고, 해제한다

```
subroutine print_list(head)
   type(node), pointer :: cur
   cur => head
   do while (associated(cur))
      write(*, '(i0, 1x)', advance="no") cur%value
      cur => cur%next
   end do
end subroutine print_list

subroutine free_list(head)
   type(node), pointer :: cur, tmp
   cur => head
   do while (associated(cur))
      tmp => cur%next        ! 해제 전에 다음 노드를 미리 저장
      deallocate(cur)
      cur => tmp
   end do
   nullify(head)
end subroutine free_list
```

```
50 40 30 20 10
list empty after free? T
```

push_front 를 1~5 로 부르면 앞에 끼워지므로 역순이 된다

표준 순회 패턴

임시 추적 포인터 cur 에 head 를 대입해 시작점을 잡고, do while (associated(cur)) 로 링크가 끊어지는 지점(null)까지 cur => cur%next 를 반복한다. 모든 포인터 기반 구조의 교과서적 패턴이다.

해제 순서가 핵심

동적 메모리는 가비지 컬렉터의 지원을 받지 못하므로 수동 해제가 필요하다. deallocate(cur) 직전에 반드시 tmp => cur%next 로 다음 노드의 행방을 확보해 둔다.

13.5 [예제]

입자 군집의 포물선 분수

```
type :: particle
   real(real64) :: pos(2)
   real(real64) :: vel(2)
end type particle

type(particle) :: cloud(n_part)          ! n_part = 9

do i = 1, n_part
   angle = (20.0_real64 + 60.0_real64 * real(i-1, real64)  &
            / real(n_part-1, real64)) * pi / 180.0_real64
   cloud(i)%pos = [0.0_real64, 0.0_real64]
   cloud(i)%vel = [speed * cos(angle), speed * sin(angle)]
end do

do step = 0, n_step
   do i = 1, n_part
      if (cloud(i)%pos(2) < 0.0_real64) cycle   ! landed
      cloud(i)%pos    = cloud(i)%pos + cloud(i)%vel * dt
      cloud(i)%vel(2) = cloud(i)%vel(2) - gravity * dt
   end do
```

```
particles.csv written
```

speed = 20 m/s, dt = 0.05 s, 각도 20°~80°, 입자 9개, 80 스텝

구조적 추상화

pos(2) 와 vel(2) 를 멤버로 갖는 particle 파생형을 선언하고, 이를 다시 cloud(n_part) 배열로 구축했다. 다수의 물리적 객체가 하나의 자료 구조 안에서 직관적으로 제어된다.

배열 일괄 연산

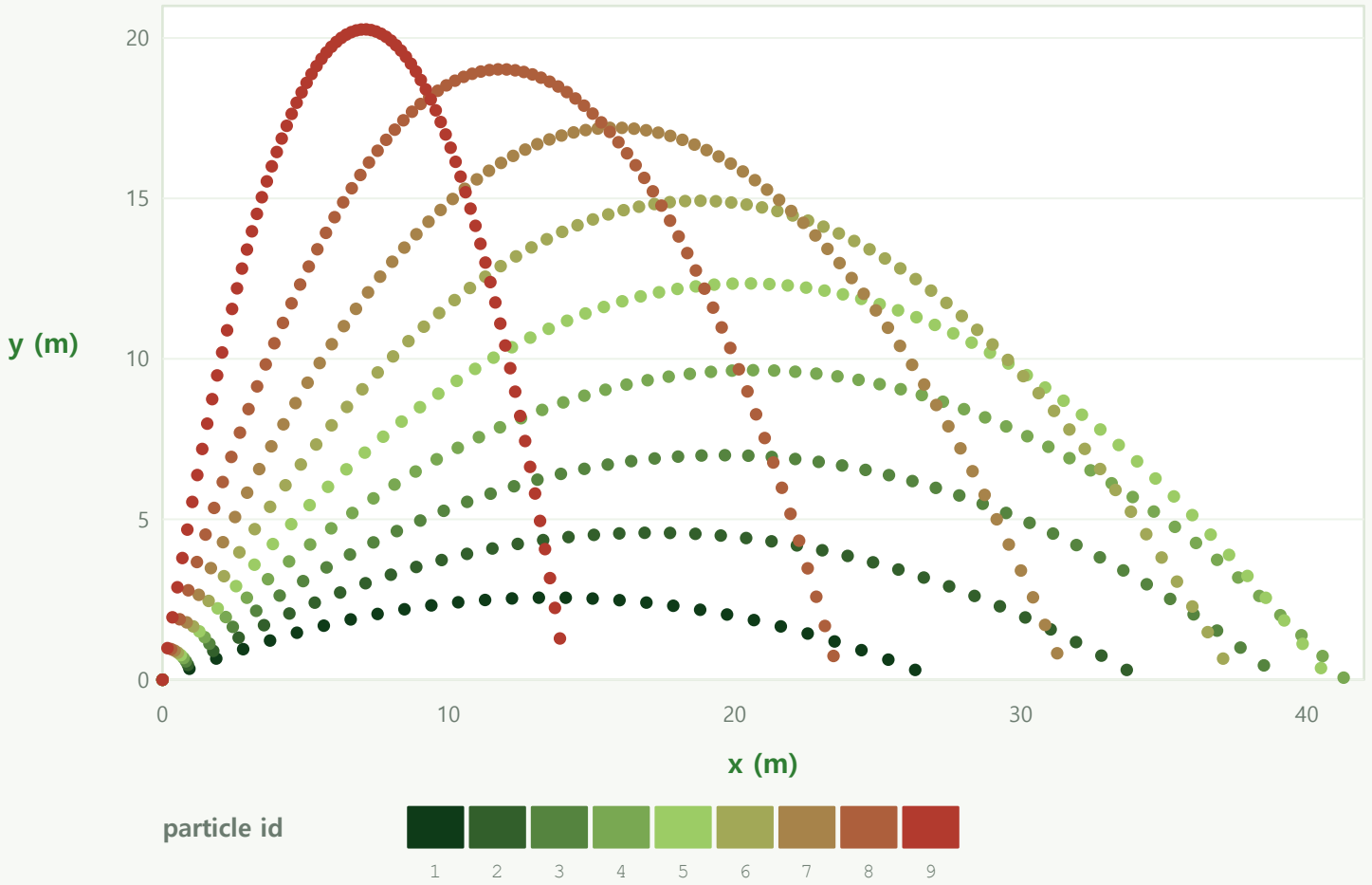
cloud(i)%pos = cloud(i)%pos + cloud(i)%vel * dt 는 길이 2 배열의 요소별 연산이다. x·y 좌표에 대해 별도의 인덱스 제어문 없이 성분별 합산이 동시에 수행된다 (8장).

현실적 제약

if (cloud(i)%pos(2) < 0.0) cycle 로 지면에 닿은 입자가 지하로 뚫고 내려가지 않고 그 자리에 멈추도록 처리했다.

13.5 [예제]

포물선 분수 — Fortran 계산, Python 그림



45° 부근에서 수평 도달 거리가 가장 길고, 각도가 커질수록 더 높이 솟는다.

아홉 개의 포물선

같은 초기 속력 20 m/s 로 20°부터 80°까지 각도만 바꿔 쏘아 올렸다. 공기 저항이 없는 중력장의 전형적인 등가속도 포물선 궤적이 분수 형태의 군집으로 나타난다.

자료 구조 하나로 표현된 입자계

pos(2), vel(2) 처럼 배열을 구성 요소로 갖는 파생형과, 그 파생형의 배열 cloud(n_part) 가 결합해 입자계 전체가 하나의 자료 구조로 깔끔하게 표현된다.

오류 학습

허상 포인터(dangling pointer)와 세그멘테이션 오류

잘못된 순서

```
cur => head
do while (associated(cur))
   deallocate(cur)      ! freed here ...
   cur => cur%next      ! ... then read freed memory
end do
```

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

무엇이 사라졌나

deallocate(cur) 가 수행되는 순간 그 노드의 메모리는 시스템에 즉각 반환된다. 내부 필드였던 value 와 다음 주소 이정표 next 정보도 함께 소멸한다. 그런데 바로 다음 문장에서 이미 사라진 cur%next 를 참조하려 한다.

올바른 순서

```
do while (associated(cur))
   tmp => cur%next      ! save the link first
   deallocate(cur)      ! then it is safe to free cur
   cur => tmp
end do
```

정상 종료 — 리스트의 모든 노드가 안전하게 해제된다

두 줄의 순서만 바꿨을 뿐

이미 해제되어 실체가 없는 대상을 계속 가리키는 포인터를 허상 포인터(dangling pointer)라 한다. 이를 역참조하는 순간 예외 없이 세그멘테이션 오류가 난다. "해제하기 직전, 다음 링크를 대피시켰는가"를 항상 검증한다.

[알아두기] 단순히 동적 배열만 필요하다면 9장의 allocatable 이 더 안전하다 — 메모리 누수와 잘못된 참조를 컴파일러가 더 잘 막아 준다. 포인터는 연결 리스트나 트리처럼 노드끼리 서로를 가리켜야 하는 구조, 또는 별명이 꼭 필요한 경우에 쓴다.

13장 정리

요약

파생형

- 파생형은 서로 다른 형의 값을 하나로 묶는 사용자 정의 자료형이다.
- 정의: type :: name … end type name, 구성 요소에 = 기본값 지정 가능.
- 변수: type(name) :: var, 접근: var%component (중첩은 var%a%b).
- 형 생성자 name(...) 로 한 번에 초기화한다 (순서 지정 또는 comp=value 키워드 지정).
- 파생형 배열에서 arr%component 는 그 구성 요소만 모은 배열을 준다 (sum(arr%score) 등).

포인터와 연결 구조

- 포인터는 대상을 가리키는 별명이다. 대상에는 target(또는 포인터) 속성이 필요하다.
- p => t 는 연관(포인터 대입), p = v 는 대상에 값 쓰기(보통 대입) — 구분이 핵심.
- 선언 시 => null() 로 초기화, 검사 associated(p), 해제 nullify(p).
- 연결 구조는 자기 참조 포인터 구성 요소와 allocate / deallocate 로 만든다. 해제 시 다음 노드를 먼저 저장한다.
- 단순 동적 배열이면 포인터보다 9장의 allocatable 을 우선한다.

한 줄 요약: 관련된 값은 파생형으로 묶고, 가리켜야 할 때만 포인터를 쓴다.

연습 문제

1. re, im 을 가진 파생형 complex_t 를 정의하고 (3, 4) 를 만들어 크기를 출력하라 (내장 complex 금지).

2. student 형으로 5명을 초기화하고, 최고 점수 학생의 이름과 점수를 출력하라.

3. point2d 두 개를 받아 두 점 사이 거리를 돌려주는 함수 distance(a, b) 를 작성하라.

4. real64 변수에 target 을 주고 포인터로 가리킨 뒤, 포인터를 통해 값을 바꿔 원본이 바뀜을 확인하라.

5. 길이 8의 target 배열에서 포인터로 arr(2:8:2) 를 가리켜 그 구간에만 0을 써 넣어라.

6. 연결 리스트에 노드 개수를 세는 함수 list_length(head) 를 추가하고, 5개를 넣어 확인하라.

7. point2d 배열로 다각형 꼭짓점을 표현하고 둘레 길이를 계산하라 (마지막→처음 변 포함).

8. particle 형으로 자유낙하시키되 y < 0 에서 vel(2) = -0.8 * vel(2) 로 튕기게 하고 그래프를 그려라.

9. 연결 리스트 맨 뒤에 노드를 추가하는 push_back(head, v) 를 작성하라 (빈 리스트도 처리).

10. name, mass 를 가진 body 형 배열로 천체 데이터를 담고, 질량이 평균보다 큰 천체만 출력하라.

다음 장에서는 파일 입출력과 문자열 처리를 다룬다.