제 5 부 — 자료 구조와 파일

14장

문자열과 파일 입출력

Text, Records, and Persistence

character · open / close / inquire · access · form · iostat · namelist

READING THE WORLD

```
open(newunit=u, ...,
     iostat=ios)

do
   read(u, *, iostat=ios) x
   if (is_iostat_end(ios)) &
      exit
end do
```

학습 목표

이 장에서 익히는 네 가지

01

문자열 다루기

character 자료형과 문자열 연산 · 내장 함수를 익힌다.

02

파일 열고 닫기

open · close 로 파일을 다루고 단위(unit)를 관리한다.

03

순차 · 직접 접근

순차 접근과 스트림 접근 파일을 구분해 쓴다.

04

데이터 저장

계산 결과를 파일로 저장하고 다시 읽어 들인다.

6장이 결과를 내보내는 쪽이었다면, 이 장은 외부 데이터를 읽어 들이는 쪽(data ingestion)에 무게를 둔다.

도입

외부 데이터를 다루기 위한 네 가지 장비

1

문자열 제어 도구

외부 텍스트를 파싱하고 정돈하기 위한 파일 경로 및 문자열 조작 기법.

`trim · index · scan`

2

파일 서술자와 상태 검증

물리 파일을 안전하게 열고 닫으며, 존재 여부와 접근 권한을 사전 점검한다.

`open · close · inquire`

3

접근과 저장 형식

데이터의 밀도와 목적에 따라 순차/스트림 접근과 텍스트/바이너리 형식을 고른다.

`access · form`

4

입출력 오류 예외 처리

실행 도중 멈추는 에러를 막기 위해 입출력 구문에 오류 지시자를 붙인다.

`iostat · iomsg`

실무에서는 결과를 내보내는 일보다, 이미 존재하는 실험·관측 자료를 프로그램 안으로 정밀하게 읽어 들이는 공정이 훨씬 많다.

이 장은 그 반대 방향 — 파일에서 메모리로 들어오는 길을 다룬다.

선언한 길이만큼 공간이 고정된다

character(len=10) :: name 으로 선언하면 정확히 10 글자만 담는 정적 공간이 할당된다. 짧은 값을 넣으면 오른쪽이 공백으로 채워지고, 긴 값을 넣으면 초과분이 잘려 나간다(truncation).

선언

```
character(len=길이) :: 변수명
```

메모리에 정해진 길이만큼의 문자열 공간을 확보한다. len 은 저장 가능한 최대 유효 글자 수다.

연결

```
'char1' // ' ' // 'char2'
```

두 문자열을 이어 붙인다. 사이에 여백이 자동으로 들어가지 않으므로 공백이 필요하면 직접 명시한다.

부분 문자열

```
변수(시작 : 끝)
```

8장의 배열 슬라이싱과 같은 표기법이다. str(2:4) 는 2번째부터 4번째 글자까지를 잘라 반환한다.

[주의] 부분 문자열의 인덱스는 선언된 전체 길이의 경계를 넘어서는 안 된다. 비어 보이는 우측 여백도 엄연히 메모리를 점유하며 유효 인덱스 범위에 포함된다.

14.1 [예제]

공백 채움과 trim

```
character(len=10) :: first = "Grace"
character(len=10) :: last  = "Hopper"
character(len=21) :: full

full = trim(first) // " " // trim(last)
```

first 변수가 실제로 점유하는 10칸

len_trim = 5 **len = 10**

```
first     = [Grace     ]      trim   = [Grace]
full      = [Grace Hopper]    substring = [Grace]
```

공백 채움의 가시화

"Grace" 뒤에 공백 다섯 칸이 따라붙어 총 10자를 점유한다. 선언 길이가 고정이라 남는 공간을 시스템이 자동으로 메운 결과다.

trim 과 // 의 조합

trim 은 우측 끝 공백을 완전히 제거한다. trim(first) // " " // trim(last) 로 단어가 멀리 떨어지는 현상 없이 "Grace Hopper" 를 얻는다.

len 과 len_trim 의 구별

len(first) 는 선언 시 배정받은 공간 크기 10 을, len_trim(first) 는 유효 글자만의 실제 길이 5 를 돌려준다.

14.1 문자열 내장 함수

검색 · 정렬 · 변환

`len(s)`	선언 시점에 할당받은 고정 길이 전체를 정수로 돌려준다.
`len_trim(s)`	우측 끝 공백을 제외한, 유효 문자들만의 실제 길이를 반환한다.
`trim(s)`	문자열 끝 부분의 모든 공백 문자를 제거한 순수 문자열을 생성한다.
`adjustl(s) / adjustr(s)`	유효 텍스트를 각각 왼쪽 또는 오른쪽으로 정렬하고, 밀려난 자리는 공백으로 채운다.
`index(s, sub [, back])`	sub 가 처음 나타나는 위치를 찾는다. 없으면 0. back=.true. 면 뒤에서부터 탐색한다.
`scan(s, set [, back])`	문자 집합 set 에 포함된 글자 중 하나라도 처음 발견되는 위치를 찾는다.
`verify(s, set)`	set 에 속하지 않는 문자가 처음 등장하는 인덱스를 돌려준다.
`repeat(s, n)`	원본 문자열 s 를 n 번 이어 붙여 확장된 새 문자열을 만든다.
`achar(i) / iachar(c)`	ASCII 코드 변환. achar 는 정수 → 문자, iachar 는 문자 → 정수.

파일 이름을 동적으로 만들거나 로그 메시지를 기록하거나 설정 파일의 경로를 파싱할 때 필수적으로 쓰인다.

14.1 [예제]

확장자 추출 — 역방향 탐색의 정석

```
character(len=*), parameter :: text = "  fortran 2018  "
character(len=*), parameter :: path = "data/input.csv"

print *, "len =", len(text)          ! 16
print *, "len_trim =", len_trim(text) ! 14
print *, "index =", index(path, ".")  ! 11
print *, "scan  =", scan(path, "/.")  ! 5
print *, "verify=", verify("12345x", "0123456789")  ! 6

dot = index(path, ".", back=.true.)
print *, "extension= [", path(dot+1:), "]"   ! csv
```

path = "data/input.csv" 의 인덱스

character(len=*)

고정 길이를 명시하지 않으면 우변에 대입되는 초깃값의 실제 글자 수를 컴파일러가 자동으로 찾아준다.

back=.true. 의 역할

문자열 끝에서부터 거꾸로 탐색한다. 경로에 점이 여러 개 섞여 있어도(directory.v1/data.input.csv) 가장 오른쪽 확장자 구분자만 정확히 짚는다.

상한 생략 슬라이싱

path(dot+1:) 처럼 끝을 비워 두면 그 지점부터 문자열 끝까지 전부를 긁어온다. 동적으로 변하는 경로에서 확장자만 뽑아내는 관용구다.

14.2 가변 길이 할당 가능 문자열

길이를 실행 시점에 정한다

```
character(len=:), allocatable :: s
```

지연 크기 지정 (len=:)

문자열의 길이를 컴파일 시점이 아닌, 실제 텍스트가 대입되는 실행 시점에 결정하겠다는 뜻이다.

시스템 내부의 자동 재할당

새 텍스트를 대입하면 9장 동적 배열의 자동 할당과 같은 규칙이 적용된다. 우변 길이에 맞춰 공간을 확보하고 기존 공간은 해제한다.

버퍼 오버플로우 차단

외부 파일에서 읽어 들이는 문자 크기가 일정하지 않아도, 메모리 낭비나 배열 경계 침범 없이 대용량 텍스트를 읽을 수 있다.

공백 채움 배제

고정 길이 방식과의 가장 큰 차이다. 불필요한 오른쪽 공백이 추가되지 않고 실제 텍스트 길이만큼만 메모리를 차지한다.

할당은 여전히 필수

우변 대입으로 자동 할당되게 하거나 allocate 로 미리 확보해야 한다. 미할당 상태에서 읽거나 연산하면 세그멘테이션 오류

파일 경로나 외부 텍스트 입력처럼 실행 전에는 길이를 전혀 예측할 수 없는 데이터를 메모리 낭비 없이 다룰 수 있다.

14.2 [예제]

대입과 결합에 따라 길이가 따라온다

```
character(len=:), allocatable :: name, greeting

name = "Ada"            ! len = 3
name = "Margaret"       ! len = 8 로 자동 재할당

greeting = ""
do i = 1, 3
   greeting = greeting // "ha"
end do                  ! len = 6, "hahaha"
```

greeting 이 자라나는 과정

초기	1회	2회	3회
""	"ha"	"haha"	"hahaha"
len = 0	len = 2	len = 4	len = 6

```
len =  3  [Ada]      len =  8  [Margaret]      len =  6  [hahaha]
```

런타임 크기 자동 피팅

"Ada" 를 대입하면 길이 3 으로 설정되고, "Margaret" 을 대입하면 기존 3바이트를 해제하고 8바이트를 재할당한다. 고정 길이의 여백 채움이나 잘림이 없다.

점진적 누적 결합

빈 문자열로 시작해 루프마다 자기 자신에게 조각을 이어 붙인다. 반복마다 컴파일러가 2 → 4 → 6 바이트로 메모리를 실시간 확장한다.

데이터 처리의 관용 패턴

길이를 모르는 텍스트를 조각조각 모아 나갈 때 자주 쓰는 구조다.

연결 → 전송 → 해제

연결 제어문

`open, close`

디스크 상의 파일과 내부 메모리 사이의 데이터 흐름 통로를 개설하고 정리한다.

데이터 전송문

`read, write, print`

통로를 통해 데이터를 양방향으로 전송시킨다.

파일 관리문

`inquire, rewind, backspace`

파일의 상태를 조사하거나 내부 데이터 포인터의 위치를 통제한다.

```
! 1. 파일 열기 및 장치 연결
open(newunit = u, file = 이름 [, status = 상태] [, action = 동작] [, form = 형식]
     [, access = 접근] [, iostat = 정수변수] [, iomsg = 문자변수])

! 2. 파일 닫기 및 장치 해제
close(u)

! 3. 파일 물리 정보 조회
inquire(file = 이름, exist = 논리변수 [, size = 정수변수])
```

버퍼 관리와 강제 기록

close 를 호출해야 메모리 버퍼에 남은 데이터가 디스크로 완전히 전송(flush)되어 데이터 유실을 막는다.

사전 상태 확인

inquire 로 파일의 존재 유무와 접근 상태를 미리 파악하면 예기치 않은 런타임 오류를 방지할 수 있다.

14.3 지정자(specifier)

파일의 성격과 처리 방식을 규정한다

status — 파일의 초기 상태

`old`	이미 존재하는 파일만 연결. 없으면 런타임 오류.
`new`	새 파일을 생성. 같은 이름이 있으면 덮어쓰지 않고 오류.
`replace`	이미 있으면 삭제 후 새 파일로 대체 생성.
`scratch`	실행 중에만 쓰는 임시 파일. 정상 종료 시 자동 삭제.
`unknown`	생략 시 기본값. 있으면 열고, 없으면 새로 만든다.

action — 작업 권한

`read`	읽기 전용
`write`	쓰기 전용
`readwrite`	양방향 (기본값)

iostat — 상태 코드

0 이면 정상 완료, 0 이 아니면 오류. 예외 처리 구문과 함께 써서 비정상 종료를 막는다.

읽을 때는 status="old", 덮어쓸 때는 status="replace" 를 명시해 코드의 목적을 분명히 기록한다.

자열로 받아 디버깅에 활용한다.

14.3 [예제]

파일 쓰기와 입출력 오류 방어

```
integer :: u, ios
character(len=200) :: msg

open(newunit=u, file="output.txt", status="replace", &
     action="write", iostat=ios, iomsg=msg)
if (ios /= 0) then
   print *, "open failed: ", trim(msg)
   error stop 1
end if

write(u, '(a)') "first line"
close(u)
```

output.txt written → first line / second line

읽기에는 status="old"

생략하면 일부 오래된 빌드 환경에서 신규 임시 파일을 멋대로 만들거나 에러를 내며 멈출 수 있다.

하드 코딩 장치 번호 퇴출

소스에 10 이나 20 같은 번호를 직접 써 넣으면 타 서브루틴 결합 시 충돌 위험이 있다. 무조건 newunit 을 쓴다.

newunit = u

장치 번호를 자동으로 확보해 정수형 변수 u 에 저장한다. 라이브러리 간 번호 중복 충돌을 차단하는 기법이다.

iostat = ios

성공하면 0, 디스크 쓰기 금지나 경로 누락 등 실패 시에는 0 이 아닌 고유 에러 코드를 넘겨받는다.

iomsg = msg + error stop

"Permission denied" 같은 원인을 사람이 읽을 문자열로 받아 출력한 뒤, 종료 코드 1 을 반환하며 안전하게 차단한다.

14.3 [예제]

inquire — 열지 않고 미리 확인한다

```
logical :: found
integer :: fsize

inquire(file="output.txt", exist=found, size=fsize)
if (found) then
   print *, "output.txt exists, size =", fsize, "bytes"
else
   print *, "output.txt not found"
end if
```

```
 output.txt exists, size =  23 bytes        missing.txt exists: F
```

왜 21 이 아니라 23 바이트인가

"first line"	10 바이트	+1 (개행)	= 11
"second line"	11 바이트	+1 (개행)	= 12

```
                    11 + 12  =  23 bytes
```

exist = found

파일이 실제로 존재하는지 점검해 논리형 변수에 저장한다. 있으면 .true., 없으면 .false.

size = fsize

파일이 점유하는 실제 용량을 바이트 단위로 측정한다. 파일이 없으면 쓰레기 값이 반환될 수 있어 반드시 exist 판정과 함께 써야 한다.

[흔한 실수] 가드 패턴의 부재

존재 여부를 확인하지 않고 size 값부터 꺼내 배열 할당 인자로 쓰면 정의되지 않은 동작, 쓰레기 값 유입, 시스템 크래시로 이어진다. exist 를 먼저 확인하고 참인 경로 안에서만 세부 정보를 다룬다.

14.4 접근 방식과 저장 형식

두 축의 조합이 성격을 결정한다

access — 접근 방식

`sequential`

기본값. 첫 부분부터 한 줄(레코드)씩 순차적으로 읽고 쓴다. csv 처리의 표준 규격이다.

`stream`

고정된 레코드가 아닌 연속된 바이트 흐름으로 취급한다. 오프셋 지정으로 임의 위치에 즉시 접근한다.

form — 저장 형식

`formatted`

기본값. 사람이 읽을 수 있는 ASCII 문자열로 변환해 저장한다. csv · 설정 파일 · 로그.

`unformatted`

메모리의 이진 데이터를 변환 없이 직접 기록한다. 빠르고 오차가 없지만 편집기로 볼 수 없다.

네 가지 조합과 용도

sequential × formatted

텍스트 · csv 처리. 눈으로 확인 가능해 설정 파일이나 로그 생성에 쓴다.

sequential × unformatted

실행 중 산출되는 중간 계산 결과를 저장했다가 다시 읽어 복원할 때.

stream × formatted

텍스트 내부에서 바이트 단위 제어가 필요할 때. 전체를 다시 쓰지 않고 일부만 수정.

stream × unformatted

대용량 수치 배열을 이진 상태로 저장. 하드웨어 최대 속도로 기록한다.

[파일 포인터 제어] rewind(u) 는 첫 레코드로 되돌리고, backspace(u) 는 한 행 뒤로 물러난다 — 크기를 모르는 파일을 두 번 읽는 패턴(two-pass)에 쓴다.

14.4 [예제]

비서식 스트림 입출력

```
real(real64) :: out(5) = [1.0_real64, 2.0_real64, 3.0_real64, &
                          4.0_real64, 5.0_real64]

open(newunit=u, file="values.bin", access="stream", &
     form="unformatted", status="replace", action="write")
write(u) out                  ! 서식 지정자가 없다
close(u)

open(newunit=u, file="values.bin", access="stream", &
     form="unformatted", status="old", action="read")
do i = 1, 5
   read(u) v
end do
```

```
v(1) = 1.00   v(2) = 2.00   v(3) = 3.00   v(4) = 4.00   v(5) = 5.00
```

8 바이트 × 5 개 = 40 바이트

같은 데이터를 텍스트로 저장하면 문자와 개행 기호가 추가되어 훨씬 큰 공간을 차지한다.

포맷 변환 생략

수치를 문자열로 바꾸는 과정에서 CPU 를 쓰지 않고 메모리의 8바이트 원시 데이터를 디스크로 직접 복사하므로 기록 속도가 대폭 향상된다.

정밀도의 완전 보존

자릿수 자르기나 반올림이 없다. 소수점 이하 마지막 비트까지 그대로 유지되어 다시 읽을 때 부동소수점 오차가 생기지 않는다.

엄격한 정렬 요구

이진 데이터는 눈으로 볼 수 없다. 읽을 때는 저장 시 지정한 자료형 규격(real64)과 완전히 동일한 타입으로 선언해야 한다.

[한계] 이식성

아키텍처마다 엔디언 규격이 달라, 한 시스템에서 만든 이진 파일을 타 기종에서 못 읽을 수 있다. 공유·장기 보존용은 csv 로 변환한다.

멈추지 않고 안전하게 우회한다

```
integer :: ios

read(u, *, iostat=ios) my_var    ! 오류가 나도 프로그램이 멈추지 않는다

if (ios /= 0) then
   ! 오류 처리 로직 수행
end if
```

에러 코드를 하드코딩하지 말 것

iostat 이 돌려주는 0 이 아닌 정수 코드는 컴파일러마다 다르다. 특정 숫자 상수로 오류 종류를 식별하면 이식성이 크게 떨어진다.

`is_iostat_end(ios)`

파일 끝(EOF)에 정상적으로 도달했는지 확인한다. 참이면 더 읽을 데이터가 없다는 뜻이므로 읽기 루프를 종료하는 조건으로 쓴다.

`is_iostat_eor(ios)`

비전진 입출력 중 현재 레코드의 끝(EOR)에 도달했는지 확인한다. 비전진 입출력은 줄바꿈 없이 원하는 크기만큼 나누어 읽고 쓰는 방식이다.

동적 파일 종단 루프 탈출

행 개수가 가변적인 시계열 데이터를 읽을 때, do 무한 루프 안에 read(..., iostat=ios) 를 배치하고 is_iostat_end 가 참이 되는 시점에 exit 한다.

원인 추적을 위한 메시지

iomsg=msg 를 함께 지정하면 파일 부재나 권한 부족 등의 시스템 에러 텍스트를 받아 원인을 파악할 수 있다.

예외 발생 시 방어적 처리

입력 일부가 손상되었을 때 전체가 죽는 것을 막는다. 문제 행을 건너뛰거나 error stop 으로 외부에 상태를 전달하고 안전하게 종료한다.

14.5 [예제]

EOF 감지로 가변 데이터 읽기

```
open(newunit=u, file="output.txt", status="old", action="read")
count = 0
do
   read(u, '(a)', iostat=ios) line
   if (is_iostat_end(ios)) exit
   if (ios /= 0) then
      print *, "read error code:", ios
      exit
   end if
   count = count + 1
   print '(i0,": ",a)', count, trim(line)
end do
close(u)
```

```
1: first line
2: second line
 total lines =  2
```

행 수를 미리 알 수 없는 관측 로그나 센서 데이터를 안전하게 읽는 표준 관용구다.

do 무한 루프

반복 한계를 지정하지 않는다. read(u, '(a)', iostat=ios) line 이 한 줄을 읽어 line 에 저장하고 상태 코드를 ios 에 반환한다.

is_iostat_end 로 탈출

파일 끝에 도달하면 ios 에 종단 상태 코드가 담기고, 질의 함수가 참을 반환해 exit 로 안전하게 루프를 빠져나온다.

EOF 와 오류의 구분

ios 가 0 이 아니면서 파일 끝도 아니라면 서식 오류나 접근 문제다. 정상적인 종단 도달과 구분해 에러 코드를 출력하고 탈출한다.

14.6 namelist

재컴파일 없이 외부 설정으로 제어한다

격자 간격이나 타임스텝을 바꿀 때마다 소스를 고치고 다시 컴파일하는 방식은 비효율적이다. namelist 는 제어 변수를 하나의 그룹으로 묶어, 실행 시점에 외부 텍스트 파일만 수정해 매개변수를 동적으로 바꾸게 해 준다.

프로그램 쪽 — 그룹 선언과 읽기

```
namelist /그룹이름/ 변수1, 변수2, ...

read(u, nml=그룹이름 [, iostat=변수])
```

설정 파일 쪽 — config.nml

```
&sim_config
   n_steps = 500
   dt = 0.005
   gravity = 9.80665
/
```

독립적 실험 제어

물리 상수를 소스에 하드코딩하지 않고 외부 파일로 분리한다. 하나의 실행 파일로 설정만 바꿔 다수의 시나리오를 실험할 수 있다.

사용자 편의성

텍스트 기반의 직관적 형식이라, 프로그래밍을 몰라도 편집기로 매개변수를 고쳐 모델을 재가동할 수 있다.

자동 식별 매핑

문자열 파싱이나 위치 서식 처리를 따로 구현할 필요가 없다. 런타임 엔진이 변수명을 대조해 순서나 공백에 관계없이 값을 대입한다.

파일은 &그룹이름 으로 시작해 변수명 = 값 을 나열하고 / 로 블록을 닫는다. namelist 선언문은 변수 선언부와 실행부 사이에 놓는다.

14.6 [예제]

설정 파일이 없어도 죽지 않는 구조

```
namelist /sim_config/ n_steps, dt, gravity

n_steps = 100                        ! 선제적 기본값 할당
dt      = 0.01_real64
gravity = 9.81_real64

open(newunit=u, file="config.nml", status="old", &
     action="read", iostat=ios, iomsg=msg)
if (ios /= 0) then
   print *, "cannot open config: ", trim(msg)
   print *, "using defaults"
else
   read(u, nml=sim_config, iostat=ios, iomsg=msg)
   if (ios /= 0) print *, "namelist warning: ", trim(msg)
   close(u)
end if
```

파일이 없어도 비정상 종료 없이 안내 메시지를 출력하고 기본값으로 계산을 이어 간다. 파일이 있으면 그 값이 기본값을 덮어쓴다.

config.nml 이 없을 때

```
cannot open config: No such file
using defaults
n_steps = 100    dt = 1.0E-002
```

config.nml 이 있을 때

```
n_steps = 500
dt = 5.0E-003
gravity = 9.80665
```

두 가지 방어 설계

① 선제적 기본값 할당 ② open · read 양쪽에 iostat 과 iomsg 배치

14.6 [예제]

외부 csv 를 읽어 이동평균 내기

```
! pass 1: 표본 수를 센다 (헤더 한 줄 건너뛰기)
open(newunit=u, file="sensor.csv", status="old", action="read")
read(u, '(a)') header
n = 0
do
   read(u, *, iostat=ios) d, t
   if (is_iostat_end(ios)) exit
   n = n + 1
end do

allocate(day(n), temp(n), smooth(n))

! pass 2: 실제 값을 읽는다
rewind(u)
read(u, '(a)') header
do i = 1, n
   read(u, *) day(i), temp(i)
end do

! 중심 이동평균 (가장자리는 있는 만큼만)
do i = 1, n
   lo = max(1, i - half);  hi = min(n, i + half)
   smooth(i) = sum(temp(lo:hi)) / real(hi - lo + 1, real64)
end do
```

```
            samples read: 14      smoothed.csv written
```

목록 지정 입력의 쉼표 파싱

read(u, *) d, t 는 서식을 지정하지 않아도 공백과 쉼표를 구분자로 인식한다. csv 행을 추가 파싱 없이 정수·실수 변수에 곧바로 대입한다.

two-pass 와 rewind

배열을 동적으로 할당하려면 행 개수를 먼저 알아야 한다. 1차 탐색으로 개수를 세고 allocate 한 뒤, rewind 로 포인터를 첫 행으로 되돌려 2차 탐색에서 값을 채운다.

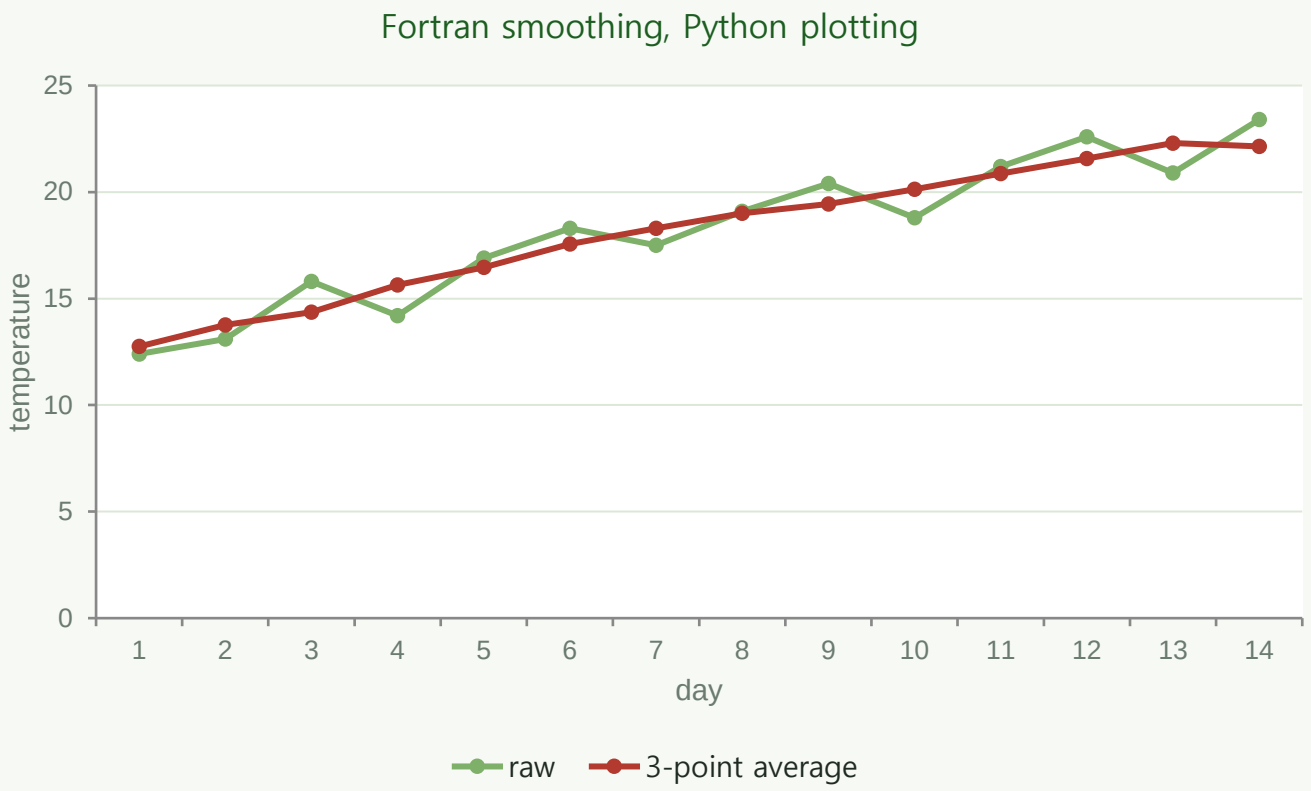
경계면 처리

배열 양 끝에서 인덱스 범위를 벗어나지 않도록 max(1, i-half) 와 min(n, i+half) 로 유효 범위 안에서만 부분합을 낸다.

헤더 한 줄을 read(u, '(a)') header 로 먼저 소모하는 것이 이 예제의 안전장치다.

14.6 [예제]

Fortran 이 평활화하고 Python 이 그린다



노이즈가 걷힌 경향선

원시 기온(raw)에는 센서 특유의 노이즈가 섞여 거시적 경향을 읽기 어렵다. 3점 이동평균을 거친 곡선은 단기 변동이 제거되어 완만한 상승 궤적이 뚜렷해진다.

검산

1일차는 가장자리라 두 점 평균 (12.4+13.1)/2 = 12.75, 2일차는 세 점 평균 (12.4+13.1+15.8)/3 = 13.767 로 출력과 일치한다.

14일치 관측 데이터를 읽어 평활화한 결과를 smoothed.csv 로 내보낸 뒤 그렸다.

오류 학습

헤더 행을 건너뛰지 않으면 실행 중에 죽는다

문제 코드

```
integer :: u, d
real :: t

open(newunit=u, file="sensor.csv", status="old", action="read")
read(u, *) d, t      ! 첫 줄은 헤더 "day,temperature"
```

```
Fortran runtime error:
  Bad integer for item 1 in list input
```

왜 실행 중에야 죽는가

문법적 결함이 없으므로 컴파일은 정상 완료된다. 그러나 실행 시 d 가 정수형으로 선언되어 있어 정수 입력을 기다리는데, 첫 행의 "day" 를 정수로 변환하는 과정에서 규격 위반이 발생한다. item 1 은 바로 그 변수 d 를 가리킨다.

안전 설계

```
character(len=200) :: header

read(u, '(a)') header       ! 헤더 행을 먼저 소모
read(u, *, iostat=ios) d, t
if (ios /= 0) then
   print *, "read failed, code =", ios
end if
```

```
           1   12.3999996
```

두 가지 방어 설계

① 파싱 전에 헤더 판독용 문자열 변수로 첫 행을 읽어 건너뛴다.
② 읽기 블록에 iostat 을 지정해 예외 발생 시 즉시 중단되지 않게 한다.

파일이 아예 없을 때는 read 가 아니라 open 단계에서 죽는다 — 모든 open 과 read 에 iostat 과 iomsg 를 함께 배치한다.

14장 정리

요약

문자열

- 문자형은 고정 길이가 기본이며 짧은 값은 오른쪽이 공백으로 채워진다. 연결은 //, 부분 문자열은 s(i:j).
- 자주 쓰는 내장 함수: len, len_trim, trim, adjustl, index, scan, verify, repeat.
- 할당가능 문자열 character(len=:), allocatable 은 대입에 따라 길이가 자동으로 맞춰져 파일 입출력에 적합하다.

파일 입출력

- 파일은 open(가급적 newunit) → 입출력 → close 순서로 다루고 status · action 을 명시한다. inquire 로 존재·크기를 미리 점검한다.
- 입출력은 접근(sequential / stream)과 형식(formatted / unformatted)의 두 축으로 갈린다. 텍스트 교환에는 서식·순차, 빠른 중간 저장에는 비서식을 쓰되 이식성 한계를 기억한다.
- 모든 입출력에 iostat(필요하면 iomsg)을 붙이고, is_iostat_end 로 EOF 를 판별한다.
- namelist 로 매개변수를 외부 설정 파일에서 읽되, 변수는 반드시 먼저 기본값으로 초기화한다.

한 줄 요약: 열기 전에 확인하고, 읽을 때마다 상태를 살피고, 다 쓰면 닫는다.

14장 마무리

연습 문제

1. character(len=20) :: city = "Busan" 을 선언하고 len 과 len_trim 의 결과를 출력해 차이를 확인하라.

2. 성과 이름을 따로 입력받아 // 로 이어 전체 이름을 만들어 출력하라.

3. "2025-06-14" 에서 부분 문자열과 내부 읽기로 연·월·일을 각각 정수로 분리해 출력하라.

4. 할당가능 문자열에 "a" 를 다섯 번 이어 붙여 "aaaaa" 를 만들고 각 단계의 len 을 출력하라.

5. 임의의 파일 이름을 inquire 로 검사해 존재하면 크기를, 없으면 없다는 문구를 출력하라.

6. 텍스트 파일에 세 줄을 쓰고, 같은 프로그램에서 다시 열어 줄 수를 세어 출력하라.

7. csv(한 열, 실수 여러 줄)를 읽어 평균과 표준편차를 계산하라. 줄 수는 EOF 로 판별한다.

8. namelist 로 n, x_min, x_max 를 읽어 구간을 n 등분한 점들을 csv 로 출력하라. 설정 파일이 없으면 기본값으로.

9. 실수 배열을 비서식 스트림으로 저장한 뒤 별도 프로그램에서 읽어 합을 구하라. kind 가 같아야 함을 확인하라.

10. 이름과 점수가 콤마로 적힌 csv 를 읽어 점수가 가장 높은 사람의 이름을 출력하라 (한 read 로 처리).

제5부의 마지막 장이다 — 다음 부에서는 더 넓은 주제로 넘어간다.