제 4 부 — 프로시저와 모듈

12장

모듈과 프로그램 구성

Organizing the Whole

module · use · public / private · 모듈 변수 · contains

A REUSABLE PART

```
module my_mod
   implicit none
   private
   public :: f
contains
   ...
end module my_mod

use my_mod, only: f
```

학습 목표

이 장에서 익히는 네 가지

01

모듈 구성

모듈을 만들고 use 로 불러 쓴다.

02

캡슐화로 데이터 보호

public 과 private 로 공개 범위를 정한다.

03

코드 묶기

모듈 변수와 contains 로 관련 코드를 한곳에 모은다.

04

범위 이해

서브모듈과 범위(scope)를 참고 수준에서 파악한다.

모듈은 상수 · 자료형 · 변수 · 프로시저를 하나의 독립된 단위로 묶어 둔 코드 패키지다.

도입

코드가 커지면 부딪히는 두 가지 문제

1

재사용의 한계

pi 값이나 사다리꼴 적분 루틴을 세 개의 프로그램이 똑같이 쓴다고 하자. 매번 복사·붙여넣기 하면, 나중에 수식을 고칠 때 일부 파일을 빠뜨려 코드 간 불일치가 생긴다.

한 번만 정의하고 여러 곳에서 가져다 쓸 장치가 필요하다

2

상태 보호의 필요

통계를 계산하는 함수 내부의 합계 변수를 외부에서 임의로 바꿔 버리면 전체 계산값이 달라진다. 정해진 통로로만 접근을 허용하고 내부 변수는 숨겨야 한다.

정해진 통로 외에는 접근을 차단할 장치가 필요하다

이 두 과제를 해결하는 도구가 바로 모듈(module)이다 — 그 자체로는 실행되지 않고, 다른 프로그램이 가져다 쓰는 독립된 부품이다.

모듈을 쓰는 본질적 이유는 코드의 재사용성을 높이고, 데이터를 안전하게 보호하기 위함이다.

12.1 모듈의 구조와 use

만들고, 불러 쓴다

```
module module_name
    ! 선언부 및 프로시저 포함
end module [module_name]
```

use 는 맨 위에

use 문은 반드시 프로그램이나 모듈 선언부의 맨 위, implicit none 보다도 먼저 작성해야 한다.

```
Error: USE statement at (1) cannot follow
  IMPLICIT NONE statement at (2)
```

모듈 프로시저는 명시적 인터페이스를 자동으로 가지므로 인터페이스 블록이 필요 없다 (11장).

전체 가져오기

```
use module_name
```

모듈이 공개(public)한 모든 요소를 제한 없이 가져온다.

선택 가져오기

```
use module_name, only: name1, name2
```

필요한 요소만 지정해 가져온다. 의존 관계가 명확해지고 이름 충돌을 막아 적극 권장된다.

이름 변경 (rename)

```
use module_name, only: new_name => old_name
```

가져올 요소의 이름이 현재 프로그램의 다른 이름과 충돌할 때 새 이름으로 바꿔 가져온다.

12.1 [예제]

상수 모듈과 선택 가져오기

```
module constants_mod
   use iso_fortran_env, only: real64
   implicit none
   private

   public :: real64
   public :: pi, two_pi, deg_to_rad

   real(real64), parameter :: pi = 3.141592653589793_real64
   real(real64), parameter :: two_pi = 2.0_real64 * pi
   real(real64), parameter :: deg_to_rad = pi / 180.0_real64
end module constants_mod

program use_constants
   use constants_mod, only: real64, pi, deg_to_rad
   implicit none
```

```
 pi         =    3.14159265
 deg     60.00 -> rad    1.04719755
 sin(60deg)=    0.86602540
```

한 파일에 module 과 program

함께 배치해 컴파일하면, 컴파일러가 의존성 구조에 따라 module 을 먼저 빌드한 뒤 이를 참조하는 program 을 그다음에 빌드한다.

real64 를 재공개한다

public :: real64 로 외부에서 가져온 이름을 다시 공개했다. 덕분에 메인 프로그램은 iso_fortran_env 를 직접 부를 필요 없이 constants_mod 하나만 참조하면 된다.

private 먼저, public 나중

기본을 닫아 두고 공개할 것만 여는 방식이다 — 12.3 의 캡슐화 원칙이다.

12.2 이름 충돌 방지

화살표(=>) 로 이름을 바꿔 받는다

서로 다른 모듈에 같은 이름의 상수(예: pi)가 선언되어 이름 충돌(name conflict)이 생길 수 있다 — use 문 뒤에 => 를 써서 새 이름으로 바꿔 수령한다.

```
use module_name, only:  로컬에서_사용할_새_이름  =>  모듈_내부의_원래_이름
```

새 이름 (내 프로그램에서 쓸 이름)

원래 이름 (모듈 안의 이름)

방향을 기억한다

geo_pi => pi 는 geo_constants 안의 pi 를, 내 프로그램에서는 geo_pi 라는 별칭으로 부르겠다는 뜻이다.

섞어 쓸 수 있다

use phys_constants, only: phys_pi => pi, c_light — 이름을 바꿀 식별자와 그대로 쓸 식별자를 쉼표 하나로 묶어 한 줄에 적는다.

실무의 정석

외부 라이브러리를 조합할 때 변수들이 우연히 겹치는 일이 잦다. 소스를 직접 고치는 위험을 감수하는 대신 => 를 전면에 배치하는 것이 안전하다.

12.2 [예제]

두 모듈의 pi 를 나란히 쓰기

```
module geo_constants
   real, parameter :: pi = 3.14159265
end module geo_constants

module phys_constants
   real, parameter :: pi = 3.1415927
   real, parameter :: c_light = 2.99792458e8
end module phys_constants

program rename_demo
   use geo_constants,  only: geo_pi => pi
   use phys_constants, only: phys_pi => pi, c_light
   implicit none
```

```
 geo_pi   =    3.14159274
 phys_pi  =    3.14159274
 c_light  =    2.9979E+08
```

```
Cannot open module file 'b_mod.mod' for reading at (1): No such file or directory
```

한 줄에 섞어 쓴 예

phys_pi => pi 는 이름을 바꿔 받고, c_light 는 원래 이름 그대로 받는다. 두 방식을 쉼표 하나로 묶어 한 줄에 기술했다.

[흔한 실수] 모듈 간 순환 참조

A 가 B 를 use 하고 B 도 A 를 use 하면, 두 모듈이 서로의 .mod 파일을 기다리는 교착 상태에 빠진다. 공통 상수·자료형을 제3의 모듈로 분리하고 A 와 B 가 각각 그것을 use 하도록 고친다.

12.3 캡슐화

기본은 닫고, 필요한 것만 연다

모듈이 선언하는 식별자는 기본적으로 모두 공개(public) 상태다. 그러나 모듈이 커질수록 외부에 제공할 핵심 기능과 내부 구현을 위해 숨겨야 할 변수를 철저히 구분해야 한다 — 이것이 캡슐화(encapsulation)다.

문장 형태

Statement Form

모듈 선언부 상단에 독립된 명령문으로 선언한다. private 한 줄을 적으면 그 아래 모든 요소의 기본 접근성이 비공개로 전환되고, 외부에 공개할 것만 public :: 이름 으로 명시한다.

```
private
public :: pi, two_pi
```

속성 형태

Attribute Form

변수나 상수를 선언하는 문장에 접근성 키워드를 속성으로 직접 결합한다. 자료형 선언문 안에 쉼표로 구분해 기술한다.

```
real(real64), public  :: pi
integer,      private :: status
```

기본 접근성을 private 로 닫고 외부에 노출할 인터페이스만 public 으로 여는 것을 권장한다 — 새 변수를 추가하며 접근성 지정을 실수로 누락해도 데이터가 새어 나가지 않는다.

12.3 [예제]

캡슐화를 적용한 계수기(counter)

```
module counter_mod
   implicit none
   private

   integer :: count = 0        ! 은폐된 상태

   public :: reset_counter, increment, current_count

contains

   subroutine reset_counter()
      count = 0
   end subroutine reset_counter

   subroutine increment(step)
      integer, intent(in), optional :: step
      if (present(step)) then
         count = count + step
      else
         count = count + 1
      end if
   end subroutine increment

   function current_count() result(value)
      integer :: value
      value = count
   end function current_count
end module counter_mod
```

```
call reset_counter()
call increment()
call increment()
call increment(step=5)
```

```
current count = 7
```

오직 세 개의 통로

count 는 private 라 메인 프로그램에서 count = 10 처럼 직접 수정하려 하면 컴파일러가 빌드를 거부한다. 호출 측은 공개된 세 프로시저만 조합해 상태를 제어한다.

1 + 1 + 5 = 7

인수를 생략해 기본값 1 씩 두 번 증가시킨 뒤, 키워드 인수로 5 를 더했다. 캡슐화된 내부에서 정확히 수행된다.

12.3 [예제]

비공개 변수에 손대면 컴파일러가 막는다

```
program bad_access
   use counter_mod
   implicit none

   count = 100      ! private 변수에 직접 대입
end program bad_access
```

```
Error: Symbol 'count' at (1)
  has no IMPLICIT type
```

왜 이런 메시지인가

count 는 private 이므로 use counter_mod 를 써도 그 이름이 프로그램 안으로 들어오지 못한다. 메인 프로그램 입장에서는 count 라는 변수가 아예 존재하지 않는 상태이고, 자료형이 지정되지 않은 이름으로 판정되어 차단된다.

캡슐화 장벽이 작동했다

내부 상태 정보가 안전하게 보호되었다. 모듈 바깥에서는 오직 모듈이 공식적으로 제공하는 통로만 보이고, 그 밖의 모든 것은 존재하지 않는 것과 같다.

분할 컴파일

```
!gfortran -std=f2018 -c counter_mod.f90
!gfortran -std=f2018 main.f90 counter_mod.o -o main
```

재사용할 모듈은 단독 파일로 분리하고 -c 로 먼저 컴파일한 뒤 링크한다.

12.3 알아두기

호스트 결합 (Host Association)

상위 구조(호스트)에 선언된 자원을 하위 구조(내부 프로시저)가 그대로 물려받아 쓰는 변수 공유 메커니즘 — 모듈 선언부의 상수·변수·자료형은 contains 아래 모든 프로시저에서 별도 선언 없이 인식된다.

호스트 결합 비사용

함수가 일을 할 때마다 외부에서 매번 pi 값을 알려 줘야 한다.

```
function calc_area(r, pi) result(area)
   real :: r, pi, area
   area = pi * r**2
end function calc_area
```

호스트 결합 사용

모듈 안에 상수를 고정해 두면, contains 아래 함수는 pi 를 인수로 받지 않아도 원래 함수 안에 있던 것처럼 쓸 수 있다.

```
module circle_mod
   real :: pi = 3.141592
contains
   function calc_area(r) result(area)
      real :: r, area
      area = pi * r**2      ! 그냥 보인다
   end function calc_area
end module circle_mod
```

프로시저 간에 복잡한 인수를 전달하지 않고도 내부 상태를 공유할 수 있다.

12.4 모듈 변수와 contains

명세부와 프로시저부의 경계

```
module ⟨이름⟩
   implicit none
   private
   ⟨모듈 변수 선언⟩        ← contains 위: 모든 프로시저가 공유
   public :: ...
contains
   ⟨function / subroutine 들⟩   ← 모듈 변수를 인수 없이 직접 사용
end module ⟨이름⟩
```

contains 상단부

데이터를 은폐하고 보호하기 위한 모듈 변수 선언과, 외부 노출을 제어하는 접근성을 기술한다.

contains 하단부

실제 로직을 수행하는 프로시저들을 배치한다. 위쪽에 선언된 모듈 변수를 자유롭게 읽고 쓴다.

전역 데이터 공유

모듈에 포함된 모든 함수와 서브루틴이 호스트 결합을 통해 동일한 모듈 변수에 제약 없이 접근한다. 인수를 주고받지 않아도 내부 상태를 공유할 수 있다.

데이터의 지속성

모듈 변수는 프로그램 시작부터 종료까지 메모리에 유지되는 긴 수명을 가진다. 선언 시 지정한 초깃값은 프로그램 시작 때 단 한 번만 설정되므로, 프로시저가 여러 번 호출되어도 누적된 값이 계속 유지된다.

이 긴 수명은 암묵적 save 속성에서 온다.

12.4 [예제]

캡슐화와 모듈 변수를 활용한 누적 통계

```
module running_stats_mod
   use iso_fortran_env, only: real64
   implicit none
   private

   integer      :: n = 0                    ! 은폐된 모듈 변수
   real(real64) :: total = 0.0_real64
   real(real64) :: total_sq = 0.0_real64

   public :: stats_reset, stats_add, stats_count, &
             stats_mean, stats_variance
contains
   subroutine stats_add(x)
      real(real64), intent(in) :: x
      n = n + 1
      total = total + x
      total_sq = total_sq + x * x
```

```
 count    = 5
 mean     =   4.0000
 variance =   2.0000
```

단 하나의 통로

메인 프로그램은 내부의 합계 변수나 제곱합 변수의 존재를 알지 못한 채, 오직 stats_add 라는 공개된 통로만으로 데이터를 안전하게 누적한다.

호출 사이에도 살아남는다

모듈 변수가 각 프로시저 호출 사이에도 메모리에 계속 남아 누적 연산을 이어간다 — 모듈 변수의 긴 수명이 만드는 특성이다.

검산

[2, 4, 4, 6, 4] 의 평균은 4. 표본분산은 (4+0+0+4+0)/(5−1) = 2 로 결과와 일치한다.

12.4 [예제]

수치 적분 루틴을 모듈로 분리하기

```
module calculus_mod
   use iso_fortran_env, only: real64
   implicit none
   private
   public :: real64
   public :: linspace, cumulative_trapz
contains
   subroutine cumulative_trapz(x, y, c)
      real(real64), intent(in)  :: x(:), y(:)
      real(real64), intent(out) :: c(:)
      integer :: i
      c(1) = 0.0_real64
      do i = 2, size(x)
         c(i) = c(i-1) + 0.5_real64 * (y(i) + y(i-1)) * (x(i) - x(i-1))
      end do
   end subroutine cumulative_trapz
end module calculus_mod

program integrate_demo
```

```
 integral.csv written
 max abs error =    8.3074E-05
```

역할 분담

적분 루틴은 calculus_mod 안에 있고, 주 프로그램은 그것을 가져다 쓰기만 한다. 다른 프로그램에서도 use 한 줄로 같은 루틴을 재사용할 수 있다.

가정형상 배열

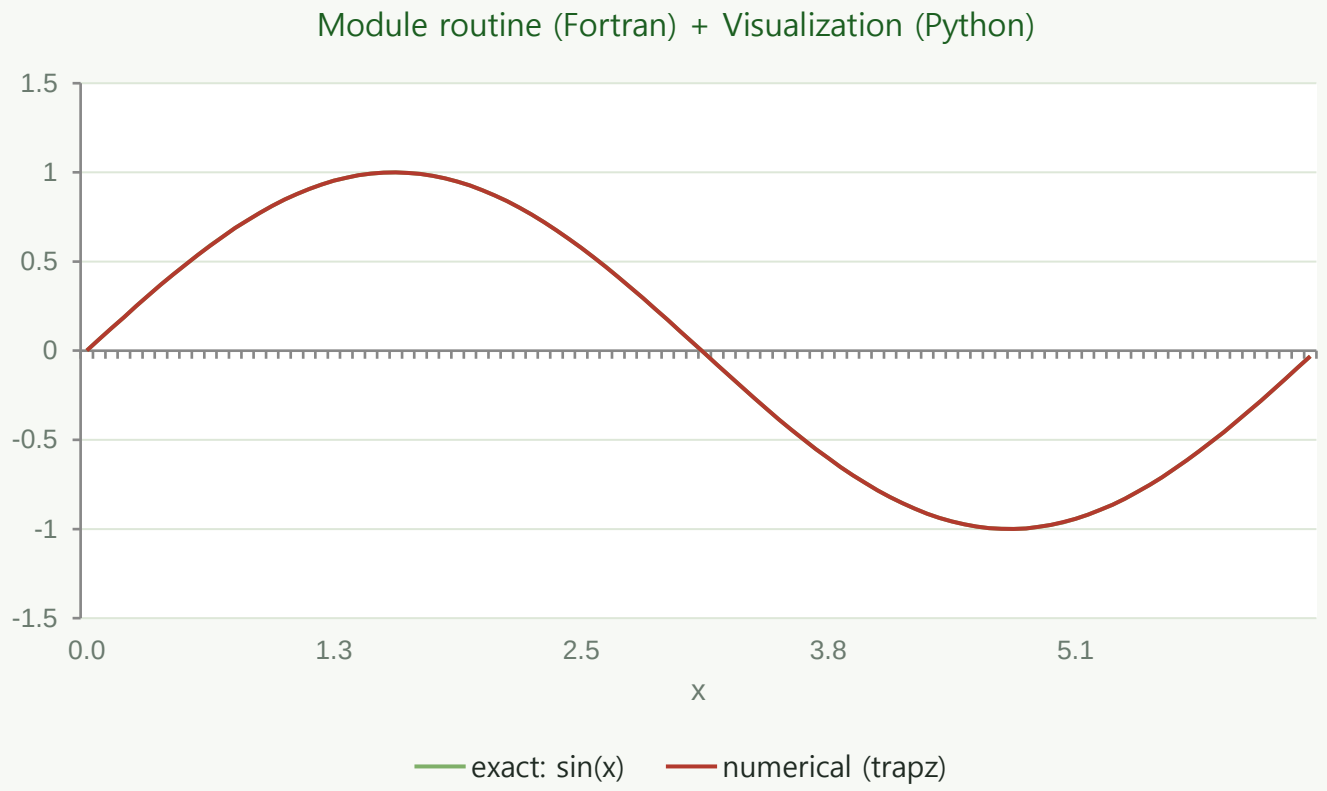
x(:), y(:), c(:) 로 선언해 크기 인수를 따로 넘기지 않는다. 모듈 프로시저라 명시적 인터페이스가 자동 확보된다 (11장).

검증 가능한 설계

cos(x) 의 적분은 sin(x) 이므로 수치해와 정확해를 함께 저장해 비교했다. 최대 오차는 약 8.3×10^{-5}.

12.4 [예제]

모듈 루틴이 계산하고 Python 이 그린다



한 곡선처럼 보인다

두 곡선을 겹쳐 그렸지만 차이가 최대 8.3×10^{-5} 에 불과해 한 줄로 보인다. cos(x) 의 누적 적분이 정확해 sin(x) 와 사실상 일치한다 — 200 개 격자점의 사다리꼴 법칙만으로 얻은 결과다.

모듈화의 이점

적분 루틴은 모듈 안에 있고 주 프로그램은 그것을 가져다 쓰기만 한다. 다른 피적분 함수로 바꿀 때도 모듈은 손대지 않는다.

두 곡선의 차이는 이 그래프의 선 굵기보다 훨씬 작다 — 사다리꼴 법칙의 오차는 $O(1/n^2)$ 을 따른다 (9장).

오류 학습

모호한 참조 (ambiguous reference)

문제 — only 없이 두 모듈을 통째로 가져오면

```
module geo_mod
   real, parameter :: pi = 3.14159265
end module geo_mod

module phys_mod
   real, parameter :: pi = 3.1415927
end module phys_mod

program clash
   use geo_mod
   use phys_mod
```

```
Error: Name 'pi' at (1) is an ambiguous
  reference to 'pi' from module 'geo_mod'
```

컴파일러는 이것이 geo_mod 에서 온 상수인지 phys_mod 에서 온 상수인지 결정하지 못한다.

해결 — 선택 가져오기와 => 로 지역 이름을 분리한다

```
program clash_fixed
   use geo_mod,  only: geo_pi  => pi
   use phys_mod, only: phys_pi => pi
   implicit none
   print *, geo_pi, phys_pi
end program clash_fixed
```

```
 3.14159274        3.14159274
```

교훈

두 상수가 프로그램 문맥 안에서 geo_pi 와 phys_pi 라는 독립된 지역 이름으로 분리되어 컴파일러가 대상을 명확히 식별한다. 애초에 use ... only: 로 필요한 것만 가져오는 습관이 이런 충돌을 예방한다.

참고

서브모듈 — 인터페이스와 구현의 분리

Fortran 2008 에서 도입된 서브모듈(submodule)은 모듈에 인터페이스만 두고, 실제 본문은 별도의 서브모듈에 두는 구조다. 구현을 고쳐도 모듈 자체는 다시 컴파일되지 않으므로, 대규모 프로젝트의 빌드 시간을 크게 줄인다.

모듈 — 인터페이스만

```
module math_mod
   interface
      module function area(r) result(a)
         real, intent(in) :: r
         real :: a
      end function area
   end interface
end module math_mod
```

서브모듈 — 본문

```
submodule (math_mod) math_impl
contains
   module procedure area
      a = 3.141592 * r**2
   end procedure area
end submodule math_impl
```

구현은 긴 형태(module function … end function) 또는 짧은 module procedure 형태로 작성한다. 이 책에서는 참고 수준으로만 다룬다.

12장 정리

요약

모듈과 가져오기

- 모듈 정의: module 이름 … end module 이름. 그 자체로 실행되지 않는 재사용 부품이다.
- 가져오기: use 모듈(전부), use 모듈, only: a, b(선택), use 모듈, only: 새이름 => 원래이름(이름 바꾸기).
- 순서 규칙: use 는 implicit none 보다 먼저 온다.
- 재사용할 모듈은 단독 파일로 분리하고 -c 로 먼저 컴파일한 뒤 링크한다.
- 정밀도: 상수·수치 루틴은 iso_fortran_env 의 real64 와 _real64 리터럴로 적는다.

캡슐화와 구조

- 접근성: 기본은 공개다. private 로 기본을 닫고 public :: … 로 공개할 것만 연다. 이것이 캡슐화의 핵심이다.
- 모듈 프로시저는 명시적 인터페이스를 자동으로 가지므로 인터페이스 블록이 필요 없다.
- contains 는 명세부와 프로시저부의 경계다. 위쪽의 모듈 변수는 모든 프로시저가 공유하고 수명이 길다(암묵적 save).
- 서브모듈(Fortran 2008, 참고): 모듈에 인터페이스만 두고 본문은 submodule (부모) 이름 에 둔다.

한 줄 요약: 한 번만 정의해 여러 곳에서 가져다 쓰고, 기본은 닫아 두고 필요한 것만 연다.

12장 마무리

연습 문제

1. 단위 환산 상수를 담은 모듈 units_mod 를 만들고, use ..., only: 로 한 상수만 가져와 환산값을 출력하라.

2. circle_const 와 sphere_const 가 모두 pi 를 공개할 때, => 로 이름을 구분해 충돌 없이 출력하라.

3. 카운터 모듈에 1 감소 서브루틴 decrement 를 추가해 공개하고, 증가·감소를 섞어 호출해 최종값을 출력하라.

4. 카운터의 count 를 public 으로 바꾸면 무엇이 가능해지는지, 왜 캡슐화 관점에서 바람직하지 않은지 서술하라.

5. 모듈 변수 call_count 를 두고, 어떤 프로시저가 호출될 때마다 1 씩 증가시켜 횟수를 조회하는 함수를 공개하라.

6. 상수 모듈에 euler_e(자연상수 e, real64)를 추가하고, exp(1.0_real64) 와 비교 출력하라.

7. 벡터 모듈 vec_mod 를 만들어 내적과 2-노름 함수를 공개하고, 내장 dot_product 와 값이 같은지 확인하라.

8. 누적 통계 모듈에 최솟값·최댓값 추적 기능을 추가하라(모듈 변수 둘과 조회 함수 공개).

9. running_stats_mod 를 재사용해 두 데이터 집합의 평균·분산을 구하라. 두 번째 집합 전에 반드시 초기화하라.

10. calculus_mod 를 재사용해 $1/(1+x^2)$ 을 누적 적분하고, Python 으로 그려 arctan(x) 에 가까운지 확인하라.

제4부가 끝났다 — 다음 부에서는 포인터와 파생형 등 고급 주제로 넘어간다.