제 4 부 — 프로시저와 모듈

10장
함수와 서브루틴

Naming the Work

function · subroutine · contains · result · recursive

TWO KINDS

```
y = f(x)

call s(a, b, c)

contains
   function f(x) &
       result(y)
   subroutine s(...)
```

학습 목표

이 장에서 익히는 네 가지

01

프로시저 작성

함수와 서브루틴을 정의하고 호출한다.

02

프로시저 위치 구분

내부 · 외부 · 모듈 프로시저의 차이를 안다.

03

반환값 다루기

result 로 함수의 반환값을 명시한다.

04

재귀함수 익히기

recursive 함수로 계승(factorial) 문제를 푼다.

프로시저는 이름이 붙은 코드 묶음이다 — 한 번 정의해 두면 이름만 불러 같은 논리를 몇 번이고 재사용한다.

10.1 프로시저

함수와 서브루틴, 무엇이 다른가

함수

function

정확히 1개의 값만 반환한다.

```
y = f(x)
```

특정 값을 계산해 돌려주는 데 집중한다. 식 안에서 직접 사용한다.

서브루틴

subroutine

인수를 통해 0개 또는 여러 개의 값을 돌려준다.

```
call s(a, b, c)
```

여러 결과를 동시에 산출하거나 다양한 부수 작업을 처리한다.

가인수

dummy argument

프로시저를 정의할 때 쓰는 변수. 내부에서 실인수를 대신해 자리를 차지한다.

실인수

actual argument

프로시저를 호출할 때 실제로 전달하는 구체적인 값이나 변수.

매핑 원리

mapping principle

실인수는 가인수와 위치 순서에 따라 일대일로 매칭되어 전달된다.

10.1 [예제]

함수 — 섭씨를 화씨로 변환

```
[형] function 이름 ( [가인수목록] ) [result(결과변수)]
```

```
program temperature
   implicit none
   real :: celsius, fahr
   do i = 0, 100, 20
      celsius = real(i)
      fahr = to_fahrenheit(celsius)
   end do
contains
   function to_fahrenheit(c) result(f)
      implicit none
      real :: c
      real :: f
      f = c * 9.0 / 5.0 + 32.0
   end function to_fahrenheit
end program temperature
```

contains 아래에 둔다

주 프로그램 본체가 끝난 뒤 contains 키워드를 쓰고 , 그 아래에 함수를 정의한다. 이렇게 정의한 것을 내부 프로시저라 한다.

```
  celsius   fahrenheit
      0.0         32.0
     20.0         68.0
    100.0        212.0
```

호출은 식 안에서

fahr = to_fahrenheit(celsius) 처럼 대입문 우변이나 식의 일부로 쓴다.

10.1 호스트 연관

내부 프로시저는 바깥의 변수를 볼 수 있다

contains 아래에 정의된 내부 함수는 바깥쪽 주 프로그램의 자원을 공유할 권한을 가진다 — 이를 호스트 연관(host association)이라 한다.

함수 안의 변수 c, f

함수 내부에서 선언되었으므로 함수 바깥(주 프로그램)에서는 인식되지 않는 완전히 독립된 변수다.

밖 → 안 ×

주 프로그램의 변수 i, celsius, fahr

주 프로그램에 선언된 변수들은 함수 안에서 그대로 가져다 쓸 수 있다.

안 → 밖 ○

이 예제는 호스트의 변수를 쓰지 않고 함수 안에서 c 와 f 를 독립적으로 선언했다 — 예기치 않은 부작용을 막아 주는 안전하고 권장되는 방식이다.

[흔한 실수] 주 프로그램과 함수 안에 같은 이름의 변수를 선언하면, 함수 내부의 지역 변수가 바깥 변수를 가려 버린다(information hiding). 함수 안에서 값을 바꿔도 주 프로그램의 변수에는 아무 영향이 없다.

10.1 [예제]

서브루틴 — 넓이와 둘레를 한 번에

```
program circle_demo
   implicit none
   real :: r, area, circumference

   r = 2.5
   call circle(r, area, circumference)
contains
   subroutine circle(radius, a, c)
      implicit none
      real :: radius, a, c
      real, parameter :: pi = 3.14159265
      a = pi * radius**2
      c = 2.0 * pi * radius
   end subroutine circle
end program circle_demo
```

```
 radius = 2.5000     area = 19.6350     circumference = 15.7080
```

실인수와 가인수의 위치 매핑

r → radius

area → a

circumference → c

함수 한 번으로는 부족하다

함수는 값 하나만 반환하므로 넓이와 둘레를 각각 구하려면 두 번 호출해야 한다. 서브루틴은 여러 인수로 두 결과를 한 번에 돌려준다.

가인수에 대입하면 밖이 바뀐다

서브루틴 내부에서 a 와 c 에 대입한 결과가, 호출한 쪽의 area 와 circumference 에 그대로 반영된다.

[흔한 실수] 호출 방식 혼동

함수를 call 로 부르거나, 서브루틴을 식 한가운데(y = s(x) + 1)에 쓰면 컴파일 오류가 난다.

10.2 프로시저의 위치

내부 · 모듈 · 외부

내부 프로시저

internal

주 프로그램이나 다른 프로시저 안의 contains 문 아래에 정의된다.

호스트 연관 · 명시적 인터페이스

모듈 프로시저

module

독립된 모듈 안의 contains 문 아래에 정의된다.

명시적 인터페이스 (12장)

외부 프로시저

external

프로그램·모듈·프로시저 어디에도 속하지 않은 독립된 형태로 정의된다.

묵시적 인터페이스 – 주의 필요

인터페이스(interface)란

프로시저를 호출하는 데 필요한 형식적 규약과 구조 — 인수 개수, 자료형, 반환값 등을 말한다. 프로시저는 실제 동작을 수행하는 코드 단위의 실체이고, 인터페이스는 그 실체를 어떻게 불러 쓰는지에 대한 약속이다.

이 절에서는 내부와 외부 프로시저를 다루고, 모듈 프로시저는 12장에서 설명한다.

10.2 [예제]

내부 프로시저 — 호스트 결합 구조

```
program host_assoc
   implicit none
   integer :: counter

   counter = 0
   call bump()
   call bump()
   call bump()
contains
   subroutine bump()
      implicit none
      counter = counter + 1      ! 호스트의 counter
```

```
counter = 3
```

내부 프로시저의 장점은 바깥 프로그램(호스트)에 선언된 변수를 인수로 넘기지 않아도 그대로 읽고 쓸 수 있다는 것이다.

시작 **counter = 0**

정수형 변수 counter 가 0 으로 초기화된다.

1회 **counter = 1**

인수가 없는 bump() 를 호출하면, 호스트 결합으로 주 프로그램의 counter 를 직접 인지해 1 로 증가시킨다.

2·3회 **counter = 3**

연달아 호출하며 counter 가 2, 3 으로 누적된다. 인수를 하나도 넘기지 않았는데 값이 바뀐다.

10.2 [예제]

외부 프로시저 — 인터페이스에 주의

```
program external_demo
   implicit none
   real :: triple        ! 외부 함수의 형을 반드시 선언
   print '(f0.2)', triple(4.0)
end program external_demo

function triple(x)
   implicit none
   real :: x
   real :: triple
   triple = 3.0 * x
end function triple
```

```
12.00
```

real :: triple 선언을 빠뜨리면

```
  Error: Return type mismatch of function 'triple' ... (UNKNOWN/REAL(4))
```

묵시적 인터페이스 (implicit)

외부 프로시저는 호출하는 주 프로그램과 분리되어 있어 컴파일러가 그 내부 구조를 자동으로 인지하지 못한다. 그래서 반환 타입을 호출하는 쪽에서 직접 선언해 주어야 한다.

그래서 내부·모듈 프로시저를 기본으로

컴파일러가 함수의 구조와 타입을 자동으로 검증할 수 있는 내부 프로시저나 모듈 프로시저를 쓰는 것이 안전하다. 외부 프로시저를 안전하게 쓰는 명시적 인터페이스는 11장에서 다룬다.

10.2 [흔한 실수]

프로시저에도 implicit none 을 빠뜨리지 말 것

implicit none 은 프로시저마다 각각 따로 써야 한다 — 주 프로그램에 썼다고 내부 프로시저까지 적용되지 않는다.

implicit none 이 없으면

```
function area(r) result(a)
   real :: r, a
   ar = 3.14159 * r * r   ! 오타: a 대신 ar
   a = a                  ! a 는 대입된 적이 없다
end function area
```

ar 이 새 변수로 묵인된다. 계산 결과는 엉뚱한 변수에 들어가고, 정작 돌려줄 a 는 값이 정해지지 않은 채 쓰레기값(garbage value)을 반환한다.

implicit none 을 넣으면

```
function area(r) result(a)
   implicit none
   real :: r, a
   ar = 3.14159 * r * r
end function area
```

```
Error: Symbol 'ar' ... has no IMPLICIT type;
  did you mean 'a'?
```

컴파일러가 오타를 잡아내고 올바른 이름까지 제안한다.

10.3 반환값과 result

두 가지 표기 방식

방식 A

함수 이름에 대입

전통적 방식

함수명 자체를 내부 변수처럼 취급해, 본문 안에서 함수명에 직접 최종 계산값을 대입한다. 함수 헤더에 반환할 자료형을 직접 명시한다.

```
real function square(a)
   implicit none
   real :: a
   square = a * a
end function square
```

방식 B

result 절 사용

현대적 방식

함수 선언부 뒤에 result(결과변수) 를 붙여 반환값을 담을 독립된 변수를 따로 정의한다. 결과 변수의 자료형은 선언부에서 정한다.

```
function cube(a) result(y)
   implicit none
   real :: a
   real :: y
   y = a * a * a
end function cube
```

[흔한 실수] 함수 이름과 result 변수는 동일한 자료형·종류를 공유한다 — 두 곳 중 단 한 곳에만 타입을 선언해야 하며, 양쪽에 모두 쓰면 중복 선언 오류가 난다.

10.3 [예제]

두 가지 방식을 나란히 써 보기

```
x = 3.0
print '(a, f6.2)', "square(x) = ", square(x)
print '(a, f6.2)', "cube(x)   = ", cube(x)

contains

   ! 방식 A — 함수 이름으로 반환
   real function square(a)
      implicit none
      real :: a
      square = a * a
   end function square

   ! 방식 B — 결과 변수로 반환
   function cube(a) result(y)
      implicit none
```

```
square(x) =    9.00        cube(x)   =   27.00
```

타입을 어디에 적는가

방식 A 는 real function square(a) 처럼 헤더에 반환 타입을 직접 적는다. 방식 B 는 헤더에 타입을 적지 않고, result(y) 로 지정한 y 의 타입을 선언부(real :: y)에서 정한다.

방식 B 의 장점

결과 변수에 함수 이름과 전혀 다른 직관적인 이름을 붙일 수 있어, 복잡한 함수 본문을 훨씬 읽기 쉽게 쓸 수 있다.

재귀에서는 B 가 필수

함수 이름이 자기 자신을 호출하는 식의 일부로 쓰이므로, 반환값을 담을 별도의 변수가 반드시 필요하다.

10.4 재귀 함수

자기 자신을 부르는 프로시저

큰 문제를 동일한 형태의 더 작은 문제로 쪼개어 해결한다 — 코드가 간결하고 논리 구조가 깔끔해지지만, 호출 횟수가 많아지면 메모리와 실행 시간이 추가로 소모된다.

1

키워드 명시

재귀적으로 작동한다는 사실을 컴파일러에 알리기 위해 recursive 키워드를 명시한다.

2

result 절의 필수화

함수 이름은 자기 자신을 호출하는 식의 일부로 쓰이므로, 반환값을 담을 별도의 결과 변수를 result 절로 지정해야 한다.

3

종료 조건 (base case)

더 이상 자신을 호출하지 않고 제어를 마치는 조건이 반드시 있어야 한다. 없으면 무한 호출로 스택 오버플로우가 발생한다.

세 표기 방식은 모두 문법적으로 동일하다

방식 1	`integer recursive function` 이름(가인수) `result`(결과변수)	
방식 2	`recursive integer function` 이름(가인수) `result`(결과변수)	
방식 3	`recursive function` 이름(가인수) `result`(결과변수)	**이 책의 표기**

10.4 [예제]

계승(factorial) 계산

```
print '(a, i0)', "fact(5) = ", fact(5)
do n = 0, 8
   print '(i2, a, i7)', n, "! = ", fact(n)
end do

contains

   recursive function fact(n) result(n_fact)
      implicit none
      integer :: n
      integer :: n_fact
      if (n <= 1) then
         n_fact = 1
      else
```

```
fact(5) = 120
----------------
 0! =       1     3! =       6     6! =     720
 1! =       1     4! =      24     7! =    5040
 2! =       2     5! =     120     8! =   40320
```

종료 조건이 먼저

if (n <= 1) 이 참이 되는 순간 fact(1) 은 조건 없이 1 을 반환한다. 이 지점이 재귀의 바닥이다.

0! 과 1! 은 모두 1

수학적 정의에 따라 두 경우 모두 종료 조건에 해당해 1 이 된다.

빠르게 커진다

2! = 2, 3! = 6 을 거쳐 8! 은 40320 이 된다. 조금만 더 키우면 32비트 정수의 한계를 넘는다 (5장).

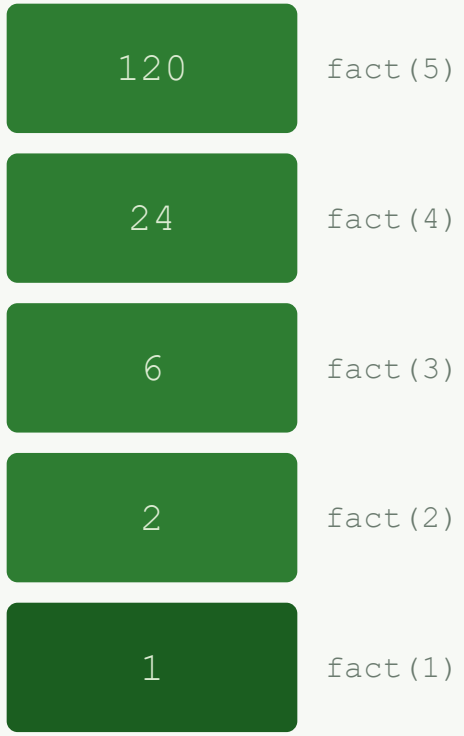
10.4 알아두기

fact(5) 가 풀려 나가는 순서

① 쌓인다 (호출)

```
fact(5)  = 5 × fact(4)
fact(4)  = 4 × fact(3)
fact(3)  = 3 × fact(2)
fact(2)  = 2 × fact(1)
fact(1)  = 1   ← 종료 조건
```

② 풀린다 (반환)



fact(5) 를 만나면 컴퓨터는 즉시 결과를 내지 못하고 5 × fact(4) 라는 식을 메모리에 쌓아 둔다. 인수가 1 에 도달하는 순간 쌓여 있던 연산이 2×1=2, 3×2=6, 4×6=24, 5×24=120 의 역순으로 풀려나가며 최종 결과를 완성한다.

10.4 [흔한 실수]

recursive 와 result 를 빠뜨리면

```
Error: ... cannot be called recursively,
  as it is not RECURSIVE
```

두 가지를 함께 확인한다

- 함수 머리에 recursive 를 붙였는가
- 반환을 위한 result 절을 선언했는가
- 종료 조건(base case)이 있는가

Fortran 2018 부터는 필수가 아니다

Fortran 2018 부터 프로시저가 기본적으로 되부름 가능해져 recursive 키워드가 반드시 필요하지는 않다. 다만 이 코드가 재귀적으로 동작한다는 사실을 읽는 사람에게 분명히 알리기 위해, 이 책에서는 명시적으로 붙인다.

```
recursive function fact(n) result(n_fact)
```

종료 조건이 없으면 — 스택 오버플로우

자기 자신을 무한히 호출하게 되고, 결국 메모리 허용 범위를 초과해 시스템이 프로그램을 강제로 중단시킨다. 재귀를 쓸 때는 "언제 멈추는가"를 가장 먼저 적는다.

10.4 [예제]

내부 프로시저끼리 협력하기

```
program plot_function
   implicit none
   call sample_to_csv(400, "curve.csv")
contains
   function damped(x) result(y)
      implicit none
      real :: x, y
      y = exp(-0.1 * x) * sin(x)
   end function damped

   subroutine sample_to_csv(npoints, filename)
      implicit none
      integer :: npoints, i, u
      character(len=*) :: filename
      real :: x
      real, parameter :: xmax = 20.0
      open(newunit=u, file=filename, status="replace", action="write")
      do i = 0, npoints
```

```
 0.00000, 0.000000      0.05000, 0.049730      0.10000, 0.098840
```

서브루틴이 함수를 부른다

sample_to_csv 안의 반복문에서 damped(x) 를 직접 호출한다. 같은 호스트에 속한 내부 프로시저들은 별도의 연결 과정 없이 서로를 자유롭게 부를 수 있다.

역할 분담

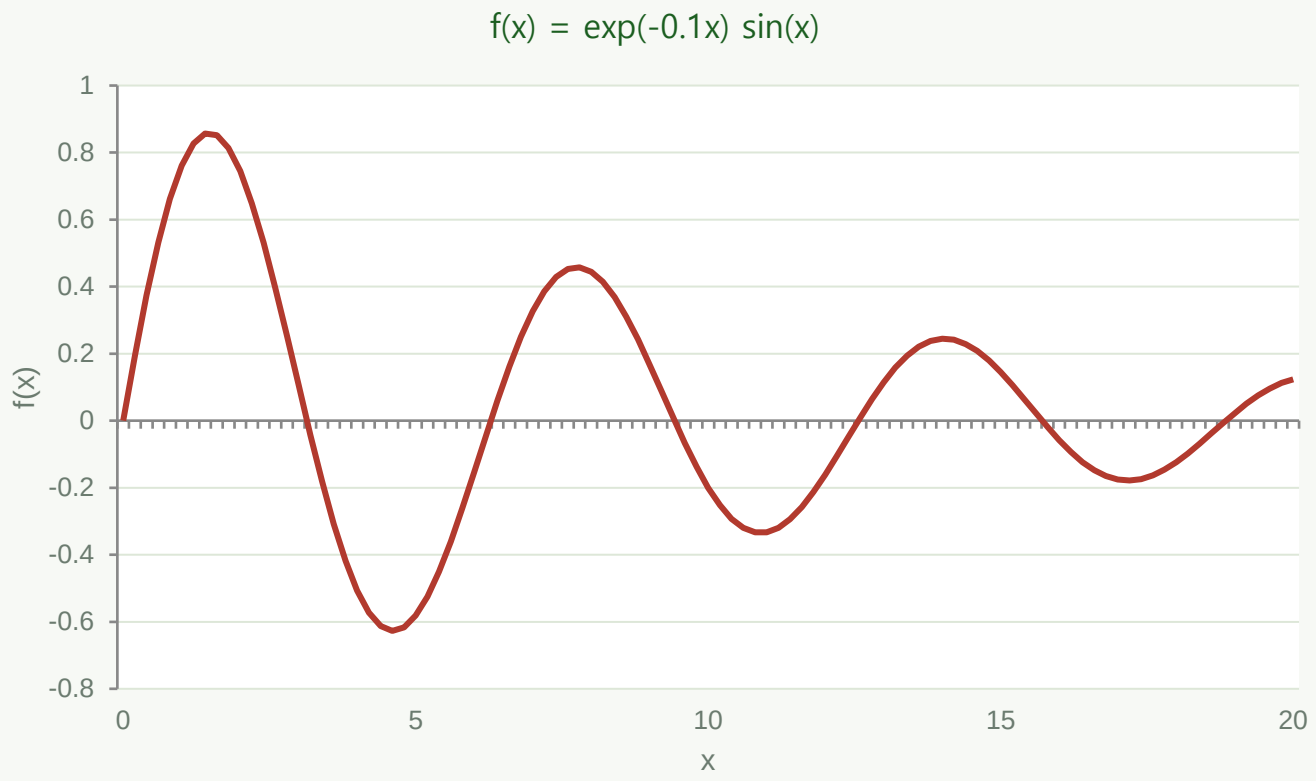
함수는 수식 하나를 계산하고, 서브루틴은 여러 지점에서 그 함수를 평가해 파일로 내보낸다.

모듈화의 이점

damped 함수 내부의 계산식 한 줄만 바꾸면, 데이터 생성 서브루틴이나 시각화 코드를 전혀 수정하지 않고도 다른 함수의 그래프를 즉시 얻는다.

10.4 [예제]

Fortran 함수, Python 그래프



감쇠 진동 (damped oscillation)

sin(x) 의 주기적 진동에 지수 감쇠 성분 exp(-0.1x) 가 곱해져, x 가 커질수록 진폭이 점차 줄어든다. 0 부터 20.0 까지 총 401 개 지점을 계산했다.

수식 한 줄만 바꾸면 된다

계산을 프로시저로 분리해 두었기 때문에, damped 안의 식만 고치면 데이터 생성과 시각화 코드는 그대로 두고 다른 함수의 그래프를 얻을 수 있다.

이것이 계산 프로그램을 프로시저로 모듈화했을 때 얻는 가장 강력한 구조적 이점이다.

오류 학습

인수 개수와 자료형이 어긋날 때

인수 개수 부족

```
call circle(r, area)    ! 인수는 세 개가 필요하다
```

```
Error: Missing actual argument
  for argument 'c' at (1)
```

세 번째 가인수 c 에 대응하는 실인수가 전달되지 않았다는 뜻이다. 둘레 값이 필요 없더라도 결과를 받아 줄 변수를 선언해 세 개를 모두 넘겨야 한다. call circle(r, area, circ) 로 고치면 정상 컴파일된다.

자료형 불일치

```
print '(f0.2)', to_fahrenheit(100)    ! 정수를 넘겼다
```

```
Error: Type mismatch in argument 'c' at (1);
  passed INTEGER(4) to REAL(4)
```

가인수 c 는 실수형인데 실인수로 정수형이 전달되었다는 뜻이다. 실인수를 100.0 으로 적거나 real(100) 처럼 명시적으로 변환해 넘겨야 한다.

두 오류 모두 실행 단계가 아닌 컴파일 단계에서 걸러졌다 — 내부 프로시저라 컴파일러가 인터페이스 구조를 정확히 알고 있기 때문이다. 외부 프로시저였다면 그대로 통과해 실행 중에 엉뚱한 값을 냈을 것이다.

10장 정리

요약

- 함수는 값 하나를 돌려주며 식 안에서 쓰고(y = f(x)), 서브루틴은 call 로 부르며 인수를 통해 결과를 0개~여러 개 돌려준다.
- 프로그램 본체 뒤 contains 아래에 두면 내부 프로시저가 되고, 호스트의 변수를 그대로 볼 수 있다(호스트 결합).
- 외부 프로시저를 안전하게 호출하려면 명시적 인터페이스가 필요하다(11장). 그전까지는 내부·모듈 프로시저를 기본으로 쓴다.
- 함수의 반환값은 함수 이름에 대입하거나 result 절로 돌려준다.
- 되부름 함수는 자기 자신을 호출하며, 종료 조건이 반드시 있어야 한다. recursive 를 명시하고 결과는 result 변수로 돌려준다.
- 프로시저마다 implicit none 을 따로 써야 하며, 모든 실행 경로에서 결과 변수에 값을 대입해야 한다.

함수 정의	`function name(args) result(r) … end function`
함수 정의 (이름 반환)	`real function name(args) … end function`
서브루틴 정의	`subroutine name(args) … end subroutine`
함수 호출	`y = name(args)`
서브루틴 호출	`call name(args)`
되부름 함수	`recursive function name(args) result(r)`
내부 프로시저	호스트의 `contains` 아래에 정의

한 줄 요약: 값을 돌려받아 식에 쓰면 함수, 호출해서 일을 시키면 서브루틴이다.

연습 문제

1. 화씨를 섭씨로 바꾸는 함수 to_celsius(f) 를 작성하고, 32°F~212°F 를 20도 간격으로 표로 출력하라.

2. result 절을 사용해 실수 x 의 세제곱을 돌려주는 함수 cube(x) 를 작성하라.

3. 정수 n 이 짝수면 .true., 홀수면 .false. 를 돌려주는 논리형 함수 is_even(n) 을 작성하라.

4. 두 정수 인수의 값을 맞바꾸는 서브루틴 swap(a, b) 를 작성하고, 호출 전후의 값을 출력하라.

5. 1~n 의 합을 되부름으로 구하는 sum_to(n) 을 작성하라. n = 10 에서 55 인지 확인하라.

6. 반지름 r 을 받아 원의 넓이만 돌려주는 circle_area(r) 를 작성하고 r = 1.0, 2.0, 3.0 에 대해 출력하라.

7. 반복문판과 되부름판 피보나치 함수를 각각 작성해 n = 0~15 를 비교하고, 되부름판이 왜 비효율적인지 설명하라.

8. deg2rad(deg) 를 작성하고, 0°~360° 를 5° 간격으로 사인값을 sine.csv 로 출력해 곡선을 그려라.

9. 이차방정식의 두 실근을 구하는 quadratic_roots(a, b, c, x1, x2, ok) 를 작성하라. 판별식이 음수면 ok 를 거짓으로.

10. 밑 base 와 지수 n 을 받아 거듭제곱을 반복 곱으로 계산하는 power(base, n) 을 작성하라 (** 사용 금지).

다음 장에서는 인수 전달 — intent 와 명시적 인터페이스, 선택적 인수를 다룬다.