

Fortran 90

KISTI 슈퍼컴퓨팅 센터



차 례

1. Fortran 90 개요
2. Numerical Robustness
3. Data Parallelism
4. 새로운 기능 사용법

제 I 장

Fortran 90 개요

Fortran 90에서 새로워진 기능을 소개
하고 Fortran 77, C/C++와 비교해 본
다.

Fortran

- ❖ **FORM**ula **TRAN**slation
- ❖ 1950년대 후반 IBM에서 개발
 - ✓ 가장 오래된 고급 언어
- ❖ 국제표준 채택(1980, ISO) → FORTRAN 77
- ❖ Fortran 90, Fortran 95, Fortran 2003

Fortran 77의 단점 (1/2)

❖ 고정 형식 소스코드 포맷

- ✓ 72열 까지만 허용
- ✓ 제 5열은 줄 번호
- ✓ 제 6열은 줄 연결 표시
- ✓ 주석은 1열에서 특정 문자로 시작, 프로그램 코드와 같은 줄에 올 수 없음
- ✓ 변수 길이 제한(6글자)

Fortran 77의 단점 (2/2)

- ❖ 병렬 연산 표현 불가능

- ❖ 동적 저장 기능 없음

- ❖ 수치적 이식성 부족

 - ✓ 시스템 간 정밀도 표현 차이

- ❖ 사용자 정의 자료구조 불가능

- ❖ 함수(루틴)의 재귀적 호출 불가능

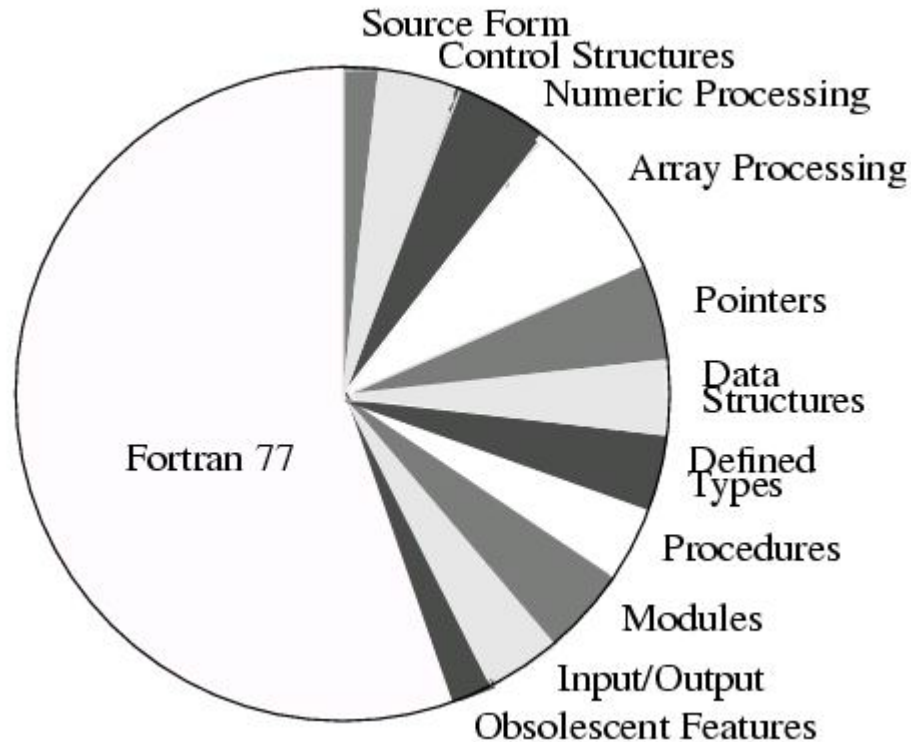
- ❖ COMMON 블록, EQUIVALENCE 문 등의 불안정성

➤ 새로운 기능 소개

- 개선된 문법
- Type Declaration과 attributing
- New control structure
- Numeric Processing
- Array Processing
- Pointers
- 데이터 타입과 연산자의 사용자 정의
- 프로시저
- 모듈
- I/O



Fortran 90



(<http://www.comphys.uni-duisburg.de/Fortran90/pl/pl.html>)

새로운 기능 소개 (1/12)

❖ 개선된 문법

- ✓ 자유형식 - 문장의 시작위치 제한 없음
- ✓ 한 줄에 132열까지 가능
 - IBM XL Fortran은 한 줄에 최대 6700문자 가능
- ✓ 한 줄에 여러 문장 가능 - 문장 분리 기호 ':'
- ✓ 문장 내에 주석 가능 - 주석 시작 기호 '!'
- ✓ 줄 바꿈 표시 - 문장 마지막에 '&'
- ✓ 변수 길이 31자까지 가능(반드시 문자로 시작)
 - IBM XL Fortran은 최대 250자 까지 가능
- ✓ 관계 연산자 표현
 - .LT., .LE., .EQ., .NE., .GE., .GT. → <, <=, ==, /=, ==>, >

새로운 기능 소개 (2/12)

```
PRINT*, "This line is continued &  
on the next line"; END ! of program
```

새로운 기능 소개 (3/12)

✓ Fortran 90에서 사용 가능한 문자 집합

- 문자-숫자 조합 : a-z, A-Z, 0-9, _ (underscore)

Symbol	Description	Symbol	Description
	Space	=	Equal
+	Plus	-	Minus
*	Asterisk	/	Slash
(Left parenthesis)	Right parenthesis
,	Comma	.	Period
'	Single quote	"	Double quote
:	Colon	;	Semi colon
!	Shriek	&	Ampersand
%	Percent	<	Less than
>	Greater than	\$	Dollar
?	Question mark		

새로운 기능 소개 (4/12)

❖ Type Declaration and Attributing

Fortran77	Fortran90
<pre>FUNCTION encode(pixels) INTEGER n PARAMETER(n=1000) INTEGER pixels(n, n), encode(n, n) REAL x(n), y(n) INTEGER*8 call call = 0 ... END</pre>	<pre>FUNCTION encode(pixels) IMPLICIT NONE INTEGER, PARAMETER :: n=1000 INTEGER, DIMENSION(n, n) :: pixels, encode REAL, DIMENSION(n) :: x, y INTEGER(8) :: call=0 ... END</pre>

새로운 기능 소개 (5/12)

❖ New Control Structure(1)

✓ SELECT-CASE

- IF 문과 같은 순차적 선택이 아닌 병렬적 선택 수행

```
SELECT CASE (expression)
CASE(value-list)
...
CASE(value-list)
...
END SELECT
```

새로운 기능 소개 (6/12)

❖ New Control Structure(2)

```
PROGRAM select
```

```
INTEGER :: n, k
```

```
PRINT *, 'Enter the value n = '
```

```
READ *, n
```

```
SELECT CASE(n)
```

```
  CASE (:0)
```

```
    k = -n
```

```
  CASE (10:)
```

```
    k = n+10
```

```
  CASE DEFAULT
```

```
    k = n
```

```
END SELECT
```

```
PRINT *, k
```

```
END
```

새로운 기능 소개 (7/12)

❖ New Control Structure(3)

✓ 인덱스 없는 DO 구문

- DO - END DO
- DO WHILE - END DO

```
DO
...
  IF (logical-expr) EXIT
...
END DO
```

```
DO WHILE (logical-expr)
...
END DO
```

새로운 기능 소개 (8/12)

❖ Numeric Processing

- ✓ 여러 가지 유용한 모델 값 리턴 하는 고유함수
- ✓ kind 값을 리턴 하는 고유함수 등 지원

❖ Array Processing

❖ Pointers

```
REAL, TARGET :: B(100,100)      ! 배열 B는 target 속성을 가진다.  
REAL, POINTER :: U(:, :), V(:, :), W(:, :)
```

 ! 3개의 포인터 배열 선언

...

```
U => B(I:I+2,J:J+2)              ! U는 B의 3x3 부분을 point  
ALLOCATE ( W(M,N) )              ! 크기가 MxN인 W를 동적할당  
V => B(:,J)                       ! V는 B의 J번째 열을 point  
V => W(I-1,1:N:2)                ! V는 W의 I-1번째 열의 일부를 point하도록 바뀜
```


새로운 기능 소개 (9/12)

❖ 사용자 정의 타입

```
TYPE RATIONAL                                ! This defines the type RATIONAL.  
  INTEGER :: NUMERATOR  
  INTEGER :: DENOMINATOR  
END TYPE RATIONAL  
...  
TYPE (RATIONAL) :: X, Y(100,100) ! X and Y are variables of type RATIONAL.  
...  
X%NUMERATOR = 2; X%DENOMINATOR = 3
```

❖ 사용자 정의 연산자

✓ 함수 + 연산자 인터페이스

```
INTERFACE OPERATOR(+)  
  FUNCTION RAT_ADD(X, Y)  
    TYPE (RATIONAL) :: RAT_ADD  
    TYPE (RATIONAL), INTENT(IN) :: X, Y  
  END FUNCTION RAT_ADD  
END INTERFACE
```

새로운 기능 소개 (10/12)

❖ 자료구조

- ✓ 배열 이외의 유용한 자료구조 정의 가능
 - 동적 연결 구조

```
TYPE LIST
  REAL                      :: DATA
  TYPE (LIST), POINTER :: PREVIOUS, NEXT  !Recursive Components
END TYPE LIST
```

❖ 프로시저

- ✓ 배열 또는 다른 자료구조를 리턴
- ✓ Recursive 호출 가능
- ✓ 명시적인 프로시저 인터페이스 정의 가능
- ✓ 내장 프로시저 가능
- ✓ 인수에 대해 OPTIONAL, INTENT 등의 속성 사용

새로운 기능 소개 (11/12)

❖ 모듈(1)

- ✓ 실행 프로그램이 접근해 사용할 수 있는 정의 포함
 - 사용자 정의 타입 정의
 - 상수 또는 변수의 선언 및 초기화
 - 프로시저 정의
 - 외부 프로시저 인터페이스 정의
 - **Generic** 프로시저의 명명
 - 연산자 인터페이스 선언

새로운 기능 소개 (12/12)

❖ 모듈(2)

```
MODULE RATIONAL_ARITHMETIC

  TYPE RATIONAL
    INTEGER :: NUMERATOR
    INTEGER :: DENOMINATOR
  END TYPE RATIONAL

  INTERFACE OPERATOR (+)
    FUNCTION RAT_ADD(X,Y)
      TYPE (RATIONAL) :: RAT_ADD
      TYPE (RATIONAL), INTENT(IN) :: X, Y
    END FUNCTION RAT_ADD
  END INTERFACE

  ...           ! and other stuff for a complete rational arithmetic facility

END MODULE RATIONAL_ARITHMETIC
```

➤ Fortran 90의 객체 지향성

- Data abstraction
- Data hiding
- Encapsulation
- Inheritance and Extensibility
- Polymorphism
- Reusability



Fortran 90의 객체 지향성 (1/4)

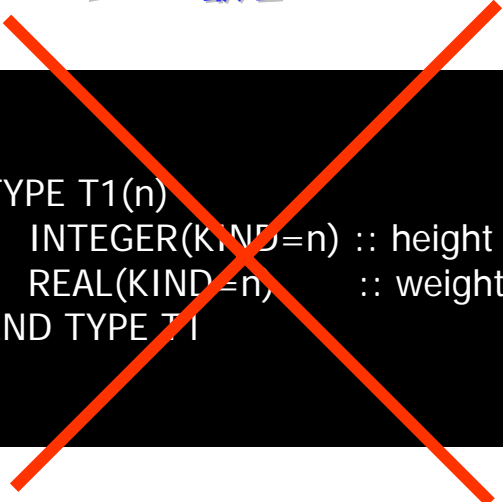
❖ Data Abstraction

- ✓ 객체나 프로시저로부터 공통된 특질을 추출하는 것
 - 예) 유사한 기능을 하는 두 함수를 하나의 함수로 합병

✓ Fortran 90에서는 일정수준만 가능

- 사용자 정의 타입, 포인터 등을 통해 지원
- 매개변수 정의타입과 서브타입 지원이 없음

```
TYPE T1
  INTEGER(KIND=1) :: height
  REAL(KIND=1)    :: weight
END TYPE T1
TYPE T2
  INTEGER(KIND=2) :: height
  REAL(KIND=2)    :: weight
END TYPE T2
```



```
TYPE T1(n)
  INTEGER(KIND=n) :: height
  REAL(KIND=n)    :: weight
END TYPE T1
```

Fortran 90의 객체 지향성 (2/4)

❖ Data Hiding

- ✓ PUBLIC, PRIVATE 속성을 이용해 모듈내의 객체 또는 프로시저에 대한 접근 제한

PRIVATE	! set default visibility
INTEGER :: pos, store, stack_size	! hidden
INTEGER, PUBLIC :: pop, push	! not hidden

Fortran 90의 객체 지향성 (3/4)

❖ Encapsulation

- ✓ 여러 관련된 기능이나 객체들을 하나의 라이브러리 또는 패키지로 정의
- ✓ 복잡한 내부구조에 대한 이해 없이 관련 기능 사용
 - data hiding, module/module procedure, use문 사용
 - module내 데이터 객체에 대한 rename
 - 특정 객체에 대한 선택적 접근

```
MODULE globals  
  REAL, SAVE : a, b, c  
  INTEGER, SAVE :: i, j, k  
END MODULE globals
```

USE globals	! allows all variables in the module to be accessed
USE globals, ONLY: a, c	! allows only variables a and c to be accessed
USE globals, r => a, s => b	! allows a and b to be accessed with local variables r and s

Fortran 90의 객체 지향성 (4/4)

❖ Inheritance and Extensibility

- ✓ 객체나 기능에 대한 계층상의 자동상속을 지원하지 않음
 - 다른 타입을 포함하는 super-type 지원
 - sub-type 없음

❖ Polymorphism

- ✓ Generic 프로시저를 통해 polymorphism 구현

❖ Reusability

- ✓ module에 포함된 내용은 언제든지 다른 프로그램 unit에 의해 접근 가능

➤ Fortran 77, C/C++와 Fortran 90의 비교



상대 순위 비교

- ❖ 계산과학을 위한 언어들의 상대적 순위 비교
([http://www.comphys.uni-duisburg.de / Fortran90/pl/pl.html](http://www.comphys.uni-duisburg.de/Fortran90/pl/pl.html))

Functionality	Fortran 77	C	C++	Fortran 90
Numerical Robustness	2	4	3	1
Data Parallelism	3	3	3	1
Data Abstraction	4	3	2	1
Object Oriented Programming	4	3	1	2
Functional Programming	4	3	2	1
Average	3.4	3.2	2.2	1.2

Fortran 77, C/C++와 비교 (1/5)

❖ Numerical Robustness

✓ 1위: Fortran90

- numeric polymorphism, kind type, 정밀도 선택, numeric environmental inquiry 등

✓ 2위: Fortran 77

- complex type 지원

✓ 3위: C++

- C보다 polymorphism 우월

✓ 4위: C

Fortran 77, C/C++와 비교 (2/5)

❖ Data parallelism

✓ 1위: Fortran90

- Fortran 90만 지원하고 있음

✓ Fortran 77, C, C++는 동률

Fortran 77, C/C++와 비교 (3/5)

❖ Data Abstraction

✓ 1위: Fortran90

- 사용하기 편리하고 효율적

✓ 2위: C++

- 계산 과학 분야에서 사용하기에 상대적으로 복잡

✓ 3위: C

- 다양한 자료구조의 지원 측면에서 Fortran 77 보다 우월

✓ 4위: Fortran 77

Fortran 77, C/C++와 비교 (4/5)

❖ 객체 지향 프로그래밍

✓ 1위: C++

- 자동상속 등의 탁월한 객체 지향성

✓ 2위: Fortran90

- polymorphic 특성의 수동상속 지원

✓ 3위: C

- 다양한 자료구조의 지원 측면에서 Fortran 77 보다 우월

✓ 4위: Fortran 77

Fortran 77, C/C++와 비교 (5/5)

❖ Functional Programming

✓ 1위: Fortran90

- lazy evaluation 기능: 필요할 때만 값을 계산

✓ 2위: C++

- polymorphism 지원

✓ 3위: C

✓ 4위: Fortran 77

- recursion, 자료구조 지원 부족

제 2 장

Numerical Robustness

계산과학 분야의 핵심이 되는 수치 계산을 위해 Fortran 90이 지원하는 풍부한 수치적 기능에 대해 알아본다.

Numerical Robustness

❖ 수치적 기능

- ✓ 단정도, 배정도 실수 타입
- ✓ (단정도)복소수 타입과 다양한 함수 라이브러리
- ✓ 배정도 복소수 타입과 4배정도 실수 타입 추가

➤ 현재의 수치적 구현 환경에 대한 추가적인 정보를 이용하거나 수치적 정밀도 향상을 통해 수치 계산 성능을 높일 수 있음

- type kind mechanism
- numeric approximation model
- environmental intrinsic function

Numeric Kind Parameterization (1/2)

❖ Fortran 77

✓ `"*size"`

- `REAL = REAL*4`
- `DOUBLE PRECISION = REAL*8`

❖ Fortran 90

- ✓ 고유 데이터 타입마다 kind 값 대응
- ✓ kind 값은 implementation에 의존
- ✓ `REAL`에 4, `DOUBLE PRECISION`에 8이 대응되면
 - `REAL = REAL(4) = REAL(kind=4)`
 - `DOUBLE PRECISION = REAL(8) = REAL(kind=8)`

Numeric Kind Parameterization (2/2)

❖ Intrinsic Function: **KIND**

- ✓ Implementation의 kind 값에 무관한 코드 작성 가능

```
INTEGER, PARAMETER :: SINGLE = KIND(1.0), &  
                      DOUBLE = KIND(1D0)  
REAL(KIND=SINGLE)    ... single precision variables ...  
REAL(KIND=DOUBLE)   ... double precision variables ...  
...
```

정밀도 선택

❖ Intrinsic Function: **SELECTED_REAL_KIND**

- ✓ 입력조건에 맞는 정밀도를 지원하는 최소의 실수 데이터 타입 kind 값을 리턴
- ✓ 입력: 사용자가 원하는 소수 정밀도와 지수 범위

```
INTEGER, PARAMETER :: P9 = SELECTED_REAL_KIND(9)
```

```
REAL(KIND=P9)      ... 9 digit (at least) precision real variables ...
```

```
...
```

✓ 실수 상수의 표현

1.41_SINGLE and 1.41 are the same,
1.41_DOUBLE and 1.41D0 are the same,
1.41_P9 is the appropriate 9+ digit representation of 1.41, and typically will be equivalent to 1.41_SINGLE(Cray) or 1.41_DOUBLE(IBM)

Numeric Polymorphism (1/5)

❖ Intrinsic Function: *Generic*

- ✓ $\text{SINGLE } x \rightarrow \text{COS}(x)$ is single
- ✓ $\text{DOUBLE } x \rightarrow \text{COS}(x)$ is double

❖ Fortran 90에서는 사용자가 정의한 프로시저도 *generic* 속성을 가질 수 있다.

- ✓ interface block 이용해 generic name 부여
- ✓ 기존 generic name에 새로운 프로시저 추가 가능

Numeric Polymorphism (2/5)

❖ integer와 real을 swap하는 외부 프로시저

```
SUBROUTINE swapint (a, b)
  INTEGER, INTENT(INOUT) :: a, b
  INTEGER :: temp
  temp = a; a = b; b = temp
END SUBROUTINE swapint
```

```
SUBROUTINE swapreal (a, b)
  REAL, INTENT(INOUT) :: a, b
  REAL :: temp
  temp = a; a = b; b = temp
END SUBROUTINE swapreal
```

Numeric Polymorphism (3/5)

❖ generic name "swap" 부여

```
INTERFACE swap                                ! generic name
  SUBROUTINE swapreal (a,b)
    REAL, INTENT(INOUT) :: a, b
  END SUBROUTINE swapreal
  SUBROUTINE swapint (a, b)
    INTEGER, INTENT(INOUT) :: a, b
  END SUBROUTINE swapint
END INTERFACE

!main program
INTEGER :: m,n
REAL :: x,y
...
CALL swap(m,n)
CALL swap(x,y)
```


Numeric Polymorphism (4/5)

INTERFACE SMOOTH

INTEGER FUNCTION SMOOTH_INT(AA)

INTEGER :: AA(:, :)

END FUNCTION SMOOTH_INT

INTEGER FUNCTION SMOOTH_SINGLE(AA)

REAL(SINGLE) :: AA(:, :)

END FUNCTION SMOOTH_SINGLE

INTEGER FUNCTION SMOOTH_DOUBLE(AA)

REAL(DOUBLE) :: AA(:, :)

END FUNCTION SMOOTH_DOUBLE

INTEGER FUNCTION SMOOTH_RATIONAL(AA)

TYPE(RATIONAL) :: AA(:, :)

END FUNCTION SMOOTH_RATIONAL

END INTERFACE

! SMOOTH is the generic name

! for procedures SMOOTH_INT

! SMOOTH_SINGLE

! SMOOTH_DOUBLE

! SMOOTH_RATIONAL

! AA is an assumed shape two-

! dimensional array in each case.

Numeric Polymorphism (5/5)

❖ 존재하는 *generic name*에 프로시저 추가

```
INTERFACE COS
  FUNCTION RATIONAL_COS(X)
    TYPE(RATIONAL) :: RATIONAL_COS
    TYPE(RATIONAL) :: X
  END FUNCTION RATIONAL_COS
END INTERFACE
```

! Extends the generic properties
! of COS to return results of
! type RATIONAL, assuming the
! argument is of type RATIONAL.

Numeric Approximation Model (1/3)

❖ Implementation의 numerical 특성 접근

- ✓ 실수 근사 모델과 관련 값을 알아볼 수 있는 16개의 intrinsic function 제공
 - Environmental inquiry: 9개
 - Numeric manipulation: 7개

Numeric Approximation Model (2/3)

❖ 실수 근사 모델

$$x = sb^e \sum f_i b^{-i} \quad i = 1, \Lambda, p$$

where

x is the real value

s is (the sign of the value)

b is the radix (base) and is usually 2; b is constant for a given real kind

p is the base b precision; p is constant for a given real kind

e is the base b exponent of the value

f_i is the i^{th} digit, base b , of the value; $0 < f_i < b$;

f_i may be 0 only if all f_i are 0

Numeric Approximation Model (3/3)

❖ 실수 근사 모델

✓ IEEE arithmetic

- $b = 2$: binary representation
- $p = 24$: single precision
- $p = 56$: double precision
- $-127 < e < 127$
- $e = -127$: 0과 NaNs(illegal or out-of-range value) 표현에 사용

✓ IBM 370 real arithmetic

- $b = 16$: non-binary representation
- $p = 6$: single precision
- $p = 14$: double precision
- $-127 < e < 127$

Environmental Inquiry (1/2)

Characteristic values of a real kind	Intrinsic function name
the decimal precision	PRECISION
the decimal precision range	RANGE
the largest value	HUGE
the smallest value	TINY
a small value compared to 1; b^{1-p}	EPSILON
the base b	RADIX
the value of p	DIGITS
the minimum value of e	MINEXPONENT
the maximum value of e	MAXEXPONENT

Environmental Inquiry (2/2)

```
REAL :: a
```

```
PRINT *, 'PRECISION =', PRECISION(a)
```

```
PRINT *, 'HUGE =', HUGE(a)
```

```
PRINT *, 'RADIX =', RADIX(a)
```

```
PRINT *, 'DIGITS', DIGITS(a)
```

```
PRINT *, 'MINEXPONENT =', MINEXPONENT(a)
```

```
...
```

```
$a.out
```

```
PRECISION = 6
```

```
HUGE = 0.3402823466E+39
```

```
RADIX = 2
```

```
DIGITS 24
```

```
MINEXPONENT = -125
```

(KISTI IBM system)

Numeric Manipulation (1/2)

Values related to the argument value	Intrinsic function name
exponent value of the number, e	EXPONENT(X)
fractional part of the number, $s \sum f_i b^{-i}$	FRACTION(X)
returns the nearest representable number(second argument specifies direction)	NEAREST(X,S)
returns the inverted value of the distance between the two nearest possible numbers	RRSPACING(X)
multiplies X by the base to the power I (change e by value of the second argument)	SCALE(X,I)
returns the number that has the fractional part of X and the exponent I (set e to value of second argument)	SET_EXPONENT(X,I)
the distance between the two nearest possible numbers	SPACING(X)

Numeric Manipulation (2/2)

```
PRINT*, 'EXPONENT =', EXPONENT(10.2)
PRINT*, 'FRACTION =', FRACTION(10.2)
PRINT*, 'NEAREST =', NEAREST(3.0, 2.0)
PRINT*, 'SPACING =', SPACING(3.0)
...
```

\$a.out

```
EXPONENT = 4
FRACTION = 0.6374999881
NEAREST = 3.000000238
SPACING = 0.2384185791E-06
```

(KISTI IBM system)

제 III 장

Data Parallelism

**데이터 병렬성을 이용한 배열 연산에 탁
월한 기능을 제공하는 Fortran 90의 특
징들을 살펴 본다.**

➤ 배열 연산

- 정합성 요구
- 배열 작성자
- 마스크된 배열 할당 - WHERE 문
- Assumed-Shape Dummy Arguments



배열 연산 (1/2)

❖ Fortran 90의 데이터 병렬 지원

- ✓ 대부분 표현에 배열을 직접 사용할 수 있음
- ✓ 대부분 연산은 배열을 피연산수로 가질 수 있고 그 결과도 배열이 될 수 있음

Fortran 77	Fortran 90
REAL a(100), b(100)	REAL, DIMENSION(100) :: a, b
DO 10 i = 1, 100	a=2.0
a(i) = 2.0	b=b*a
10 b(i) = b(i)*a(i)	

배열 연산 (2/2)

- ☑ A, B, C, P, Q, R : 2차원 배열
- ☑ Z, V : 1차원 배열
- ☑ S, X : Scalar

```
C(:, :) = A(:, :) + B(:, :)  
PRINT* , P(:, :) * Q(:, :) - R(:, :), S  
CALL T3(X, Q(:, :), Z(:) - V(:))
```

||

```
C = A+B  
PRINT* , P*Q-R, S  
CALL T3(X, Q, Z-V)
```

정합성 요구 (1/2)

✂ 배열 연산에 참여하는 배열의 모양은 일치해야 한다.

$$A = \begin{pmatrix} 2 & 3 & 5 \\ 1 & 7 & 4 \end{pmatrix}, \quad B = \begin{pmatrix} 5 & 4 & 1 \\ 2 & 3 & 2 \end{pmatrix}$$

$$A + B = \begin{pmatrix} 7 & 7 & 6 \\ 3 & 10 & 6 \end{pmatrix}, \quad A \times B = \begin{pmatrix} 10 & 12 & 5 \\ 2 & 21 & 8 \end{pmatrix}$$

$$B + 2 = \begin{pmatrix} 5 & 4 & 1 \\ 2 & 3 & 2 \end{pmatrix} + \begin{pmatrix} 2 & 2 & 2 \\ 2 & 2 & 2 \end{pmatrix} = \begin{pmatrix} 7 & 6 & 3 \\ 4 & 5 & 4 \end{pmatrix}$$

정합성 요구 (2/2)

✂ 계산이 수행되는 배열은 우변 계산이 완료된 후 저장된다.

$$G(P,:) = G(P,:) / G(P,K)$$

- $G(P,:)$: 행렬 G 의 P 번째 행
- $G(P,K)$: 행렬 G 의 P 번째 행, K 번째 열 성분

배열 작성자

❖ 1차원 배열의 명시적 구성에 이용

- ✓ 스칼라 표현 : (/1,2,3,4/)
- ✓ 암시적 DO 구문
- ✓ 배열 표현

❖ 2차원 이상의 배열 구성 → RESHAPE 함수

암시적 DO 구문

(expression-list, index-variable=first-value,
last-value[, increment])

✓ (/1,2,3, ..., n/) = (/ (k, k=1,n) /)

✓ (/1,0,1,0, ..., /) = (/ (1,0, j=1,500000) /)

! Implied DO-lists:

(/ ((i + j, i = 1, 3), j = 1, 2) /) != (/ 2, 3, 4, 3, 4, 5 /)

! Arithmetic expressions:

(/ (1.0 / REAL(i), i = 1, 6) /)

!= (/ 1.0/1.0, 1.0/2.0, 1.0/3.0, 1.0/4.0, 1.0/5.0, 1.0/6.0 /)

!= (/ 1.0, 0.5, 0.33, 0.25, 0.20, 0.167 /)

배열 표현

❖ 임의 차원 배열 이용한 배열 작성자 구성

☑ A 가 1000×1000 의 2차원 배열일 때

☑ $(/A+1.3/)$

- 배열 A 의 각 원소에 1.3을 더한 값(100만개)을 원소로 가지는 1차원 배열 작성자
- 기본적으로 열 우선 순으로 나열 됨
 - $(/ ((A(j,k)+1.3, j=1,1000), k=1,1000) /)$
- 행 우선 순 나열
 - $(/ ((A(j,k)+1.3, k=1,1000), j=1,1000) /)$

RESHAPE 함수 (1/2)

❖ 배열 작성자(1차원)를 이용해 원하는 모양의 배열 구성

RESHAPE(array-creator, shape-vector)

✓ Shape-vector

- 원하는 모양에 대한 각 차원의 크기를 (/.../)로 지정

```
REAL, DIMENSION (3,2) :: ra
```

```
ra = RESHAPE( (/ ((i+j, i=1,3), j=1,2) /), SHAPE=(/3,2/) )
```

RESHAPE 함수 (2/2)

❖ 배열 상수의 정의

✓ 1000X1000 크기의 실수타입 단위 행렬 정의

```
REAL, PARAMETER, DIMENSION(1000,1000) :: Ident_1000 =      &  
RESHAPE( (/ (1.0,(0.0, k=1,1000), j=1,999),1.0 /), (/1000,1000/))
```

마스크된 배열 할당 – WHERE 문 (1/3)

- ❖ 배열 연산이 일부 원소에만 적용되도록 logical 타입의 마스크 사용

WHERE(mask) array-assignment-statement

- ✓ WHERE 구문의 mask가 .TRUE.일 때 배열 원소에 값이 할당

마스크된 배열 할당 – WHERE 문 (2/3)

❖ WHERE (C .NE. 0) A = B/C

$$B = \begin{pmatrix} 1.0 & 2.0 \\ 3.0 & 4.0 \end{pmatrix}, \quad C = \begin{pmatrix} 2.0 & 0.0 \\ 0.0 & 2.0 \end{pmatrix}$$

$$A = \begin{pmatrix} 0.5 & - \\ - & 2.0 \end{pmatrix}$$

마스크된 배열 할당 – WHERE 문 (3/3)

❖ WHERE (C .NE. 0)

$$A = B/C$$

ELSE WHERE

$$A = B$$

END WHERE

$$A = \begin{pmatrix} 0.5 & 2.0 \\ 3.0 & 2.0 \end{pmatrix}$$

Assumed-Shape Dummy Arguments

❖ Fortran 프로시저의 dummy 배열 사용

- ✓ Explicit-Shape (F77, F90)
- ✓ Assumed-Size (F77, F90)
- ✓ Assumed-Shape (F90)

Explicit-Shape

- ❖ dummy 배열의 모양과 크기가 명시적으로 결정
- ❖ 프로시저의 쓰임새가 한정적

```
SUBROUTINE S1(A, B, C, k, m, n)  
  
REAL:: A(100, 100)    !static  
REAL:: B(m, n)        !adjustable  
REAL:: C(-10:20, k:n) !adjustable  
...
```

Assumed-Size

- ❖ dummy 배열 index의 마지막 값을 지정하지 않고 사용
- ❖ 마지막 index를 제외한 나머지는 명시적 선언 필요

```
SUBROUTINE S2(A, B, C, k, m)

REAL:: A(100, 100)      !static
REAL:: B(m, *)          !assumed-size
REAL:: C(-10:20, k:*)   !assumed-size
...
```

Assumed-Shape (1/3)

- ❖ **dummy 배열은 대응되는 실제 배열의 정보를 전달받아 사용**
- ❖ **dummy 배열과 실제 배열은 차원과 형식이 일치해야 함**

```
SUBROUTINE S3(A, B, C, k)

REAL:: A(100, 100)    !static
REAL:: B(:, :)        !assumed-shape
REAL:: C(-10:20, k:)  !assumed-shape
...
```

Assumed-Shape (2/3)

- ❖ 외부 프로시저가 **assumed-shape dummy arguments**를 사용할 경우 명시적인 정의 필요
 - ✓ 호출 프로그램이 인터페이스 블록 사용해 정의

Assumed-Shape (3/3)

... ! calling program unit

INTERFACE

SUBROUTINE sub (ra, rb, rc)

REAL, DIMENSION (:, :) :: ra, rb

REAL, DIMENSION (0:, 2:) :: rc

END SUBROUTINE sub

END INTERFACE

REAL, DIMENSION (0:9,10) :: ra ! Shape (/ 10, 10 /)

CALL sub(ra, ra(0:4, 2:6), ra(3:7, 5:9))

...

END

SUBROUTINE sub(ra, rb, rc) ! External

REAL, DIMENSION (:, :) :: ra ! Shape (/10, 10/)

REAL, DIMENSION (:, :) :: rb ! Shape (/ 5, 5 /) = REAL, DIMENSION (1:5, 1:5) :: rb

REAL, DIMENSION (0:, 2:) :: rc

! Shape (/ 5, 5 /) = REAL, DIMENSION (0:4, 2:6) :: rc

...

END SUBROUTINE sub

➤ 배열 부분(Array Sections)

- 배열 부분
- 동적 배열
- Array-Valued 함수
- Assumed-Shape Dummy Arguments



배열 부분

❖ 배열 부분의 표현 방식

- ✓ Simple Subscript
- ✓ Subscript Triplet
- ✓ Vector Subscript

Simple Subscript

❖ 배열의 원소 하나를 표현

array-name(subscript-1, subscript-2, ...)

$$B = \begin{pmatrix} 1.0 & 2.0 \\ 3.0 & 4.0 \end{pmatrix} \Rightarrow B(1,2) = 2.0$$

Triplet Subscript (1/3)

❖ 배열 부분을 세 개의 정수(start index, end index, stride)로 표현

- ✓ stride가 생략된 경우 기본값은 1
- ✓ start index가 생략된 경우 기본값은 대응 차원의 lower bound
- ✓ end index가 생략된 경우 기본값은 대응 차원의 upper bound

Triplet Subscript (2/3)

$$Q = \begin{pmatrix} 13 & 11 & 25 & 2 & 1 & 9 \\ 9 & 3 & 31 & 14 & 52 & 27 \\ 16 & 45 & 54 & 36 & 15 & 20 \\ 7 & 20 & 18 & 19 & 8 & 19 \\ 37 & 56 & 54 & 66 & 77 & 90 \end{pmatrix}$$

$$Q(1:5,2) = \begin{pmatrix} 11 \\ 3 \\ 45 \\ 20 \\ 56 \end{pmatrix} = Q(:,2)$$

$$Q(1:5:2,2) = \begin{pmatrix} 11 \\ 45 \\ 56 \end{pmatrix} = Q(:,2,2)$$

$$Q(1:2,5:6) = \begin{pmatrix} 1 & 9 \\ 52 & 27 \end{pmatrix} = Q(:,2,5:) \quad Q(5,4:6) = (66 \quad 77 \quad 90) = Q(5,4:)$$

Triplet Subscript (3/3)

Fortran 77	Fortran 90
<pre>REAL A(10,10),B(10,10) DO 1 J=1,8 DO 2 I=1,8 A(I,J) = B(I+1,J+1) 2 CONTINUE 1 CONTINUE</pre>	<pre>REAL, DIMENSION(10,10) :: A,B A(1:8,1:8) = B(2:9,2:9)</pre>

Vector Subscript (1/4)

❖ 배열의 subscript를 (/.../)로 묶은 1차원 배열로 표현

```
REAL, DIMENSION :: ra(6), rb(3)  
INTEGER, DIMENSION (3) :: iv
```

```
iv = (/ 1, 3, 5 /)      ! rank 1 integer expression (vector subscript)  
ra = (/ 1.2, 3.4, 3.0, 11.2, 1.0, 3.7 /)  
rb = ra(iv)             ! iv is the vector subscript  
! = (/ ra(1), ra(3), ra(5) /)  
! = (/ 1.2, 3.0, 1.0 /)
```

Vector Subscript (2/4)

$$Q = \begin{pmatrix} 13 & 11 & 25 & 2 & 1 & 9 \\ 9 & 3 & 31 & 14 & 52 & 27 \\ 16 & 45 & 54 & 36 & 15 & 20 \\ 7 & 20 & 18 & 19 & 8 & 19 \\ 37 & 56 & 54 & 66 & 77 & 90 \end{pmatrix}$$

Triplet Subscript	Vector Subscript
$Q(1:5, 2) = Q(:, 2)$	$Q((/1,2,3,4,5/), 2)$
$Q(1:2, 5:6) = Q(:, 2, 5:)$	$Q((/1,2/), (/5,6/))$
$Q(5, 4:6) = Q(5, 4:)$	$Q(5, (/4,5,6/))$
$Q(1:5:2, 2) = Q(:, 2, 2)$	$Q((/1,3,5/), 2)$

Vector Subscript (3/4)

❖ 암시적 DO 구문 표현

$$Q(/1,2,3,4,5/), 2) = Q(/(k, k=1,5)/), 2)$$

$$Q(/1,2/), (/5,6/)) = Q(/(k, k=1,2)/), (/ (k, k=5,6)/))$$

$$Q(5, (/4,5,6/)) = Q(5, (/ (k, k=4,6)/))$$

$$Q(/1,3,5/), 2) = Q(/(k, k=1,5,2)/), 2)$$

Vector Subscript (4/4)

❖ subscript의 중복 사용 가능

✓ 차원의 크기가 주어진 전체 배열보다 큰 배열 정의 가능

$$Q((/ 4,1,2,3,4,2,5 /), (/1,4,4,3 /)) = \begin{pmatrix} Q_{41} & Q_{44} & Q_{44} & Q_{43} \\ Q_{11} & Q_{14} & Q_{14} & Q_{13} \\ Q_{21} & Q_{24} & Q_{24} & Q_{23} \\ Q_{31} & Q_{34} & Q_{34} & Q_{33} \\ Q_{41} & Q_{44} & Q_{44} & Q_{43} \\ Q_{21} & Q_{24} & Q_{24} & Q_{23} \\ Q_{51} & Q_{54} & Q_{54} & Q_{53} \end{pmatrix}$$

➤ 동적 배열

- Autonomic Array
- Allocatable Array
- Pointer Array



동적 배열

❖ 정적 메모리 할당(F77, F90)

- ✓ 컴파일 할 때 배열의 크기와 주소가 결정

❖ 동적 메모리 할당(F90)

- ✓ 프로그램 실행 중에 배열의 크기와 주소 결정
- ✓ 배열의 크기를 입력 값 또는 프로그램에서 계산되는 값으로 하는 것이 가능

Autonomic Array (1/2)

❖ 그 크기가 dummy 인수에 의해 결정되는 로컬 배열

```
SUBROUTINE sub(n, a)
  IMPLICIT NONE
  INTEGER :: n
  REAL, DIMENSION(n, n) :: a
  REAL, DIMENSION (n, n) :: work1
  REAL, DIMENSION (SIZE(a, 1)) :: work2
  ...
END SUBROUTINE sub
```

Autonomic Array (2/2)

❖ SIZE (A, n)

✓ 배열 A의 n번째 차원의 크기 리턴

```
FUNCTION F18(A,N)
  INTEGER :: N                ! A scalar
  REAL :: A(:, :)            ! An assumed shape array
  COMPLEX :: Local_1(N, 2*N+3)
                        ! Local_1 is an automatic array whose size is based on N.
  REAL :: Local_2(SIZE(A,1), SIZE(A,2))
                        ! Local_2 is an automatic array exactly the same size as A.
  REAL :: Local_3(4*SIZE(A,2))
                        ! Local_3 is a one-dimensional array 4 times the size of
                        ! the second dimension of A.

  ...
END FUNCTION F18
```

Allocatable Array (1/2)

❖ ALLOCATABLE attribute로 선언

- ✓ 배열 이름과 차원은 미리 결정, 그 크기는 입력 값이나 계산 값에 의해 결정

```
PROGRAM simulate
```

```
IMPLICIT NONE
```

```
INTEGER :: n
```

```
INTEGER, DIMENSION(:, :), ALLOCATABLE :: a    ! 2D
```

```
PRINT *, 'Enter n:'
```

```
READ *, n
```

```
IF(.NOT. ALLOCATED(a)) ALLOCATE( a(n,2*n) )
```

```
...
```

```
DEALLOCATE(a)
```

```
...
```

Allocatable Array (2/2)

❖ 배열 할당 상태 확인: STAT

ALLOCATE(allocate_object_list [, STAT= status])
DEALLOCATE(allocate_obj_list [, STAT= status])

```
INTEGER, DIMENSION(:), ALLOCATABLE :: ages  ! 1D
REAL, DIMENSION(:, :), ALLOCATABLE :: speed ! 2D
...
READ*, isize
ALLOCATE(ages(isize), STAT=ierr)
IF (ierr /= 0) PRINT*, "ages : Allocation failed"
ALLOCATE(speed(0:isize-1, 10), STAT=ierr)
IF (ierr /= 0) PRINT*, "speed : Allocation failed"
...
```

Pointer Array

❖ POINTER attribute로 선언

- ✓ Allocatable문 이용해 할당
- ✓ TARGET으로 선언된 다른 배열이나 배열 부분에 대한 aliasing(point to)을 위해 사용

```
REAL, TARGET :: B(100,100)  
REAL, POINTER :: U(:, :), V(:, :), W(:, :)
```

! 배열 B는 target 속성을 가진다.
! 3개의 포인터 배열 선언

```
...  
U => B(I:I+2,J:J+2)  
ALLOCATE ( W(M,N) )  
V => B(:,J)  
V => W(I-1,1:N:2)  
...
```

! U는 B의 3x3 부분을 point
! 크기가 MxN인 W를 동적할당
! V는 B의 J번째 열을 point
! V는 W의 I-1번째 열의 일부를
! point하도록 바꿈

➤ Array-valued 함수

- Transformational 함수
- Elemental 함수
- 사용자 정의 array-valued 함수



Transformational 함수 (1/4)

❖ 배열이나 스칼라를 입력으로 받아 다른 모양의 배열이나 스칼라로 'transform'하여 결과 출력

✓ 42개의 Fortran 90 고유함수 존재

- array 함수(21)
- environmental inquiry 함수(9)
- 기타(12)

Transformational 함수 (2/4)

Transformational Intrinsic Function	Comment
environmental inquiry function (9)	←
array function (21)	→
ASSOCIATED	check association of pointer
BIT_SIZE	number of bits of integer
DOT_PRODUCT	mathematical dot product of two vectors
KIND	←
LEN	length of a character string
MATMUL	mathematical matrix product
PRESENT	check presence of an optional argument
REPEAT	replicate a character string
SELECTED_INT_KIND	←
SELECTED_REAL_KIND	←
TRIM	remove trailing blanks from a string
TRANSFER	transfer bit pattern to a different type

Transformational 함수 (3/4)

Array Intrinsic Function	Comment
ALL	true if all of the element values are true
ANY	true if any of the element values are true
ALLOCATED	check if array is allocated
COUNT	number of elements having the value true
CSHIFT	circularly shift an array along a dimension
EOSHIFT	end-off shift an array along a dimension
LBOUND	lower bound of an array
MAXLOC	location of maximum element in an array
MAXVAL	maximum element value in an array
MERGE	merge two arrays, under a mask
MINLOC	location of minimum element in an array
MINVAL	minimum element value in an array
PACK	gather an array into a vector, under a mask
PRODUCT	product of all the elements of an array
SHAPE	shape of an array
SIZE	total size of an array
SPREAD	spread an array by adding a dimension
SUM	sum of all of the elements of an array
TRANSPOSE	matrix transpose of a 2-dimensional array
UBOUND	upper bound of an array
UNPACK	scatter a vector into an array, under a mask

Transformational 함수 (4/4)

REAL :: a(1024), b(4,1024)

scalar = SUM(a)	! sum of all elements
a = PRODUCT(b, DIM=1)	! product of elements in first dim
scalar = COUNT(a == 0)	! gives number of zero elements
scalar = MAXVAL(a, MASK=a .LT. 0)	! largest negative element

LOGICAL a(n)

REAL, DIMENSION(n) :: b, c

IF (ALL(a))	...	! global AND
IF (ALL(b == c))	...	! true if all elements equal
IF (ANY(a))	...	! global OR
IF (ANY(b < 0.0))	...	! true if any element < 0.0

Elemental 함수

❖ 스칼라 dummy 인수로 정의되지만, 실제 인수가 배열이 되면 그 배열과 같은 모양의 배열을 결과로 리턴

- ✓ 대부분의 Fortran 계산 함수
- ✓ 배열의 원소 각각에 대해 함수 적용
- ✓ 총 108개의 elemental 함수
 - ex) $\text{COS}(X)$, $\text{SIN}(X)$, $\text{SQRT}(A)$, ...

사용자 정의 array-valued 함수

❖ Transformational 함수

❖ 결과가 되는 배열의 모양은 동적으로 결정

```
FUNCTION Partial_sums(P)
  REAL :: P(:)                ! Assumed-shape dummy array
  REAL :: Partial_sums(SIZE(P)) ! The partial sums to be returned
  INTEGER :: k
  Partial_sums = (/SUM(P(1:k), k=1,SIZE(P))/)
    ! This is functionally equivalent to
    ! DO k=1,SIZE(P)
    !   Partial_sums(k) = SUM(P(1:k))
    ! END DO
    ! but the do loop specifies a set of sequential
    ! computations rather than parallel computations
END FUNCTION Partial_sums
```

제 IV 장

새로운 기능 사용법

**Fortran 90에 새롭게 추가된 다양하고
편리한 기능들을 알아보고 그 사용법을
익힌다.**

➤ 인터페이스 블록

- 프로시저와 인터페이스
- 인터페이스 블록
- 내부 프로시저
- INTENT Attribute



프로시저와 인터페이스

❖ 프로시저(함수, 서브루틴)

✓ 외부 프로시저(F77, F90)

- 메인 프로그램과 분리된 독립적인 프로그램 단위
- 지역적으로 선언된 것에 접근 불가
- 정보 전달을 위해 함수 이름과 인수 사용
- 암시적 인터페이스

✓ 내부 프로시저(F90)

- 프로그램 단위 내에 포함
- 프로시저 참조에 대한 각종 오류를 컴파일러가 점검 가능
- 명시적 인터페이스
- 모듈 프로시저
 - 모듈에 포함

인터페이스 블록 (1/5)

❖ 외부 프로시저에 대한 명시적 인터페이스 제공

- ✓ 컴파일러에게 프로시저 인수의 여러 가지 속성을 알려줘 프로시저 참조에 대한 오류 점검 가능하게 함

```
INTERFACE  
    interface_body  
END INTERFACE
```

- ✓ **interface_body**
 - FUNCTION/SUBROUTINE header
 - Dummy 인수 선언
 - 지역변수 선언
 - END FUNCTION/END SUBROUTINE 문

인터페이스 블록 (2/5)

❖ 외부 프로시저 호출이 있으면, 항상 인터페이스 블록 사용을 권장

```
PROGRAM stress

INTERFACE
  SUBROUTINE squash(a,n)
    REAL :: a(n)
  END SUBROUTINE
END INTERFACE

INTEGER, PARAMETER :: m = 100
REAL :: q(m)

q=71
CALL squash(q,m)

...
```

인터페이스 블록 (3/5)

❖ 암시적 인터페이스

```
PROGRAM test
  INTEGER :: i=3, j=25
  PRINT *, 'The ratio is ', ratio(i, j)
END PROGRAM test
```

```
REAL FUNCTION ratio(x, y)
  REAL :: x, y
  ratio=x/y
END FUNCTION ratio
```

```
$ xlf90 nointface.f -qextchk
** test   === End of Compilation 1 ===
** ratio  === End of Compilation 2 ===
1501-510  Compilation successful for file nointface.f.
ld: 0711-197 ERROR: Type mismatches for symbol: .ratio
ld: 0711-345 Use the -bloadmap or -bnoquiet option to obtain more information.
```

인터페이스 블록 (4/5)

❖ 명시적 인터페이스

```
PROGRAM test
```

```
INTERFACE
```

```
  REAL FUNCTION ratio(x, y)
```

```
    REAL::x, y
```

```
  END FUNCTION ratio
```

```
END INTERFACE
```

```
INTEGER :: i=3, j=25
```

```
PRINT *, 'The ratio is ', ratio(i,j)
```

```
END PROGRAM test
```

```
REAL FUNCTION ratio(x, y)
```

```
  REAL:: x, y
```

```
  ratio=x/y
```

```
END FUNCTION ratio
```

인터페이스 블록 (5/5)

❖ 명시적 인터페이스

```
$ xlf90 intface.f
"intface.f", line 9.34: 1513-061 (S) Actual argument attributes do not match
those specified by an accessible explicit interface.
** test   === End of Compilation 1 ===
** ratio  === End of Compilation 2 ===
1501-511  Compilation failed for file intface.f.
```

내부 프로시저 (1/2)

- ❖ 프로그램 단위(호스트)의 맨 마지막 부분, **CONTAINS**문 다음에 위치
- ❖ END문 다음에 FUNCTION 또는 SUBROUTINE을 반드시 넣어야 함
- ❖ 호스트에서 선언된 변수는 내부 프로시저에서 접근가능, 재정의도 가능

내부 프로시저 (2/2)

❖ 명시적 인터페이스

```
PROGRAM test

INTEGER :: i=3, j=25
PRINT *, 'The ratio is ', ratio(i,j)

CONTAINS
  REAL FUNCTION ratio(x, y)
    REAL :: x, y
    ratio=x/y
  END FUNCTION ratio

END PROGRAM test
```

```
$ xlf90 intproc.f
"intproc.f", line 3.31: 1513-061 (S) Actual argument attributes do not match
those specified by an accessible explicit interface.
** test   === End of Compilation 1 ===
1501-511  Compilation failed for file intproc.f.
```

INTENT Attribute (1/3)

❖ 프로시저 내의 인수들에 대한 이용 계획 명시

✓ 효율적인 컴파일, 프로그램 안정성 증가

✓ **in, out, inout**

- **INTENT(in)**: 들어와서 나갈 때까지 값의 변화가 없는 인수
- **INTENT(out)**: 값을 새로 할당 받을 때까지 사용되지 않는 인수
- **INTENT(inout)**: 프로시저에 들어와 사용되고 값을 새로 할당 받아 그 결과를: 호출 프로그램에 되돌려 주는 인수

INTENT Attribute (2/3)

```
PROGRAM intent_test
```

```
REAL :: x, y
```

```
y = 5.
```

```
CALL mistaken(x, y)
```

```
PRINT *, x
```

```
CONTAINS
```

```
  SUBROUTINE mistaken(a, b)
```

```
    IMPLICIT NONE
```

```
    REAL, INTENT(in) :: a
```

```
    REAL, INTENT(out) :: b
```

```
    a = 2*b
```

```
  END SUBROUTINE mistaken
```

```
END PROGRAM intent_test
```

INTENT Attribute (3/3)

```
$xlf90 intent.f
```

```
"intent.f", line 14.2: 1516-055 (S) The INTENT(IN) attribute specifies that a dummy argument, or a subobject of a dummy argument, must not be redefined or become undefined during the execution of the procedure.
```

```
** intent_test === End of Compilation 1 ===
```

```
1501-511 Compilation failed for file intent.f.
```

➤ 유도 타입(Derived Type)

- 유도 타입
- Supertypes
- 유도 타입 할당



유도 타입 (1/2)

❖ TYPE/END TYPE을 이용해 새로운 타입 정의

```
TYPE type_name  
  Declarations  
END TYPE type_name
```

- ✓ 기본 타입과 같은 *attribute*를 가질 수 있으나
PARAMETER *attribute*를 가질 수 없음

유도 타입 (2/2)

❖ 3차원 좌표(COORDS_3D)의 정의와 사용 예

```
TYPE COORDS_3D  
  REAL :: x, y, z  
END TYPE COORDS_3D
```

```
TYPE(COORDS_3D) :: pt
```

```
TYPE(COORDS_3D), DIMENSION(10, 20), TARGET :: pt_arr
```

Supertypes

- ❖ 이미 정의된 유도 타입 이용해 새로운 유도 타입 정의
- ❖ 현재 정의하는 타입을 내부에 다시 사용할 수 있음 → recursive 자료 구조 형성

```
TYPE SPHERE  
  TYPE (COORDS_3D) :: center  
  REAL              :: radius  
END TYPE SPHERE
```

(SPHERE는 COORDS_3D의 supertype)

유도 타입 할당 (1/2)

❖ 유도 타입에 값을 할당하는 두 가지 방법

- ✓ Component by component
- ✓ As an object

❖ 유도 타입의 특정 항 표현: '%' 연산자 이용

```
TYPE (SPHERE) :: bubble, ball
```

```
bubble%radius = 3.0
```

```
bubble%center%x = 1.0
```

```
bubble%center%y = 2.0
```

```
bubble%center%z = 3.0
```

유도 타입 할당 (2/2)

❖ 유도 타입 작성자 이용

```
bubble%center = COORDS_3D(1.0,2.0,3.0)
```

❖ SPHERE 작성자 이용

```
bubble = SPHERE(bubble%center, 3.0)
```

```
bubble = SPHERE(COORDS_3D(1.0,2.0,3.0), 3.0)
```

```
bubble = (1.0,2.0,3.0,3.0) (X)
```

```
ball = bubble
```


➤ 모듈(Module)

- 모듈
- 모듈:Global 데이터
- 모듈:프로시저
- 모듈:Generic 프로시저
- 모듈:Public, Private 객체
- 모듈:컴파일과 링크



모듈 (1/2)

❖ 새로운 프로그램 단위, 모든 프로그램 단위는 USE문을 이용해 모듈을 첨부할 수 있음

✓ Global 객체 선언

- Global 데이터 설정. COMMON, INCLUDE 대신 사용

✓ 인터페이스 선언

✓ 프로시저 선언

✓ Controlled 객체 선언

- 접근 구문 이용해 변수, 프로시저 등의 노출 정도 제어

✓ Packaged of Whole Sets of Facilities

- 유도 타입, 프로시저, 인터페이스, 연산자 등을 하나로 묶어 객체 지향성 제공

✓ Semantic Extension

- 프로그램 단위에 첨부돼 Fortran 90의 한 부분으로 기능

모듈 (2/2)

❖ 기본 모양

```
MODULE <module name>  
  <declarations and specifications statements>  
  [CONTAINS  
    <definitions of module procedures>]  
END [MODULE [<module name>]]
```

✓ Use-association

- 모듈 안에 USE문을 이용해 다른 모듈 첨부 가능

모듈: Global 데이터

❖ Global 변수를 모듈에 선언

✓ Fortran 77의 **COMMON**문 대체

☑ **USE** 문은 프로그램 단위에서 가장 위에 위치

```
MODULE globals  
  REAL :: a, b, c  
  INTEGER :: i, j, k  
END MODULE globals
```

USE globals	! allows all variables in the module ! to be accessed
USE globals, ONLY: a, c	! allows only variables a and c ! to be accessed
USE globals, r => a, s => b	! allows a and b to be accessed with ! local variables r and s

모듈: 프로시저 (1/2)

❖ 모듈에 포함되는 내부 프로시저

- ✓ CONTAINS문 다음에 위치
- ✓ END문 다음에 FUNCTION 또는 SUBROUTINE이 있어야 함
- ✓ 유도 타입과 관련 함수들을 하나로 묶어 관리하는데 유용

모듈: 프로시저 (2/2)

```
MODULE point_module
```

```
  TYPE point
```

```
    REAL :: x, y
```

```
  END TYPE point
```

```
CONTAINS
```

```
  FUNCTION addpoints (p, q)
```

```
    TYPE (point), INTENT(IN) :: p, q
```

```
    TYPE (point) :: addpoints
```

```
    addpoints%x = p%x + q%x
```

```
    addpoints%y = p%y + q%y
```

```
  END FUNCTION addpoints
```

```
END MODULE point_module
```

```
PROGRAM point_sum
```

```
!A program unit would contain:
```

```
USE point_module
```

```
TYPE (point) :: px, py, pz
```

```
px = point(1., 2.)
```

```
py = point(2., 5.)
```

```
pz = addpoints(px, py)
```

```
PRINT*, ' pz =', pz
```

```
END
```

모듈: Generic 프로시저 (1/6)

❖ Generic 인터페이스

- ✓ 유사한 기능을 하는 프로시저를 묶어 하나의 이름으로 사용

```
INTERFACE generic_name  
    Specific_interface_body  
    Specific_interface_body  
    ...  
END INTERFACE
```

- ✓ 함수와 서브루틴을 같이 묶어 정의할 수 없음
- ✓ 모듈 내에 정의해서 편리하게 사용

모듈: Generic 프로시저 (2/6)

☑ 정수와 실수를 SWAP하는 서브루틴

```
SUBROUTINE swapreal(a, b)
  REAL, INTENT(inout) :: a, b
  REAL :: temp
  temp = a
  a = b
  b = temp
END SUBROUTINE swapreal
```

```
SUBROUTINE swapint(a, b)
  INTEGER, INTENT(inout) :: a, b
  INTEGER :: temp
  temp = a
  a = b
  b = temp
END SUBROUTINE swapint
```


모듈: Generic 프로시저 (3/6)

☑ Generic SWAP 인터페이스 정의

```
INTERFACE swap                ! generic name
```

```
  SUBROUTINE swapreal(a, b)  
    REAL, INTENT(inout) :: a, b  
  END SUBROUTINE swapreal
```

```
  SUBROUTINE swapint(a, b)  
    INTEGER, INTENT(inout) :: a, b  
  END SUBROUTINE swapint
```

```
END INTERFACE
```

모듈: Generic 프로시저 (4/6)

☑ Generic 프로시저 SWAP 호출

```
...  
INTEGER :: m, n  
REAL :: x, y  
...  
CALL swap(m, n)  
CALL swap(x, y)  
...
```

모듈: Generic 프로시저 (5/6)

```
MODULE genswap
  TYPE point
    REAL :: x, y
  END TYPE point

  INTERFACE swap ! generic interface
    MODULE PROCEDURE swapreal, swapint, swaplog, swappoint
  END INTERFACE

  CONTAINS
    SUBROUTINE swappoint (a, b)
      TYPE (point), INTENT(INOUT) :: a, b
      TYPE (point) :: temp
      temp = a; a=b; b=temp
    END SUBROUTINE swappoint
    ... ! swapint, swapreal, swaplog procedures are defined here
END MODULE genswap
```

모듈: Generic 프로시저 (6/6)

```
PROGRAM main
```

```
USE genswap
```

```
INTEGER :: m, n
```

```
REAL :: x, y
```

```
TYPE(point) :: a, b
```

```
...
```

```
CALL swap(m, n)
```

```
CALL swap(x, y)
```

```
CALL swap(a, b)
```

```
...
```

```
END PROGRAM main
```

모듈: Public, Private 객체

❖ 일부 프로그램의 특정 객체에 대한 접근 제한

✓ PRIVATE 문 또는 PRIVATE attribute 사용

```
PRIVATE :: pos, store, stack_size ! hidden  
PUBLIC  :: pop, push              ! not hidden
```

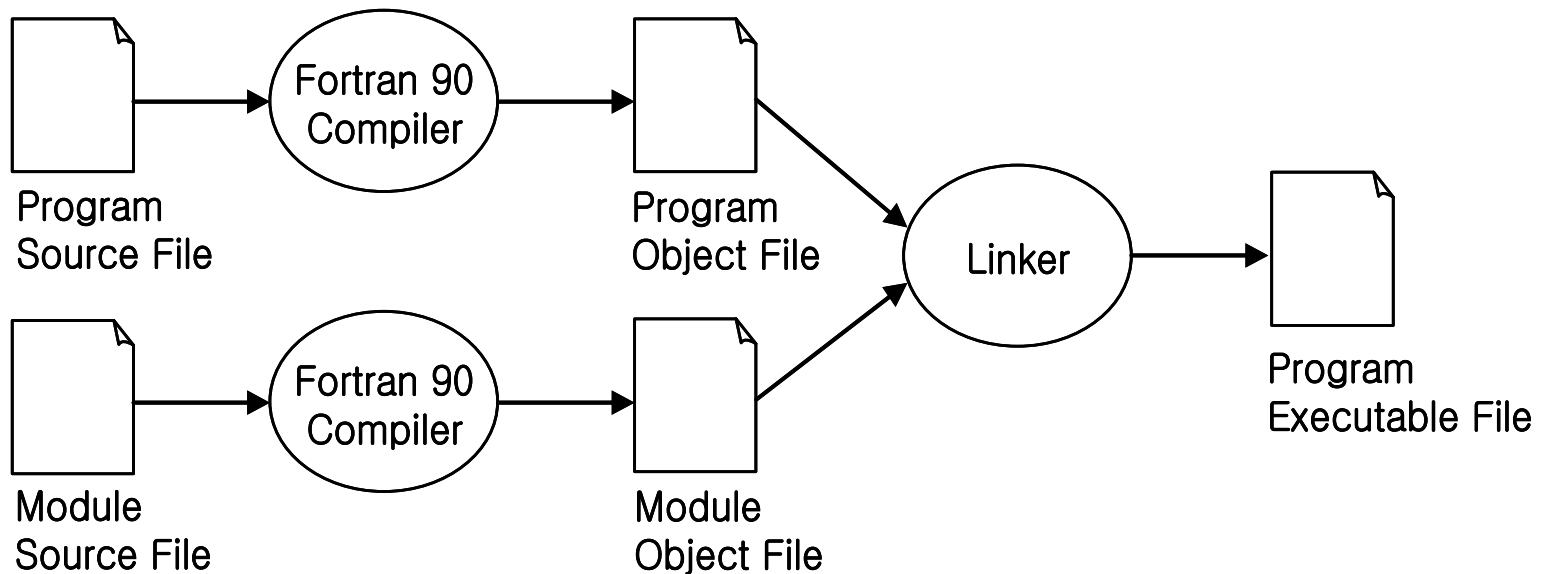
```
MODULE blimp  
  IMPLICIT NONE  
  PRIVATE                               ! set default visibility  
  ...  
END MODULE blimp
```

```
PUBLIC          ! set default visibility  
INTEGER, PRIVATE :: store(stack_size), pos  
INTEGER, PRIVATE, PARAMETER :: stack_size = 100
```

모듈: 컴파일과 링크

❖ 소스 프로그램 컴파일, 모듈 컴파일 → 링크

✓ 컴파일에 우선 순위는 없음



➤ 포인터

- Fortran 90의 포인터
- 포인터 상태
- 포인터 선언과 할당
- 포인터와 배열
- 동적 타깃
- 자동 Attributing
- 포인터 Disassociation
- 포인터 Valued 함수
- Linked List



Fortran 90의 포인터

- ❖ Fortran 90의 포인터 변수는 값을 저장할 수 없으며 단지 타겟에 대한 **Aliasing**의 기능만을 제공
- ❖ Fortran 90 포인터 변수의 기능은 참조되는 공간(타겟: target) 또는 그 공간에 저장된 값을 바꿀 수 있도록 하는 것
- ❖ Fortran에서 Linked list, tree 등의 동적 자료 구조 작성 가능

포인터 상태

❖ 포인터 변수의 상태는 3가지

- ✓ **undefined** : 포인터의 초기상태
- ✓ **associated** : 포인터가 가리키는 타깃이 있는 상태
- ✓ **disassociated** : 정의 되었지만 타깃이 없는 상태

❖ Logical 함수 ASSOCIATED 이용해 상태 확인 가능

포인터 선언과 할당 [1/6]

❖ POINTER Attribute로 선언

- ✓ 태그의 타입, kind, 차원 등이 고정
- ✓ 배열을 가리키는 포인터는 항상 deferred-shape
- ✓ 태그의 차원은 고정되나 모양은 변화 가능
- ✓ 포인터 변수는 문자를 가리킬 수 없음

```
REAL, POINTER :: ptor  
REAL, DIMENSION(:, :), POINTER :: ptoa
```

포인터 선언과 할당 (2/6)

❖ 타깃은 TARGET Attribute를 가지도록 선언

```
REAL, TARGET :: x, y  
REAL, DIMENSION(5,3), TARGET :: a, b  
REAL, DIMENSION(3,5), TARGET :: c
```

- ✓ x, y는 ptor과 associated 가능
- ✓ a, b, c는 ptoa와 associated 가능

포인터 선언과 할당 (3/6)

❖ 포인터에 값을 할당하는 두 가지 방법

✓ 포인터 할당 : '='>

- aliasing 기능, 포인터와 그 타겟은 동일 공간을 나타냄
- 포인터와 타겟, 포인터와 포인터 사이

✓ 표준(normal) 할당 : '='

- 포인터가 가리키는 공간에 저장되는 값 설정
- 포인터와 객체 사이, 이때 객체는 TARGET attribute를 가질 필요 없음

포인터 선언과 할당 (4/6)

```
ptor => y  
ptoa => b
```

- ✓ ptor은 y를 ptoa는 b를 aliasing
- ✓ y와 b의 값이 바뀌면 ptor와 ptoa의 값도 변화 단, 가리키는 공간은 변함 없음

```
ptor2 => ptor  
ptor2 => y
```

- ✓ ptor과 ptor2는 모두 y를 aliasing 하는 포인터

포인터 선언과 할당 (5/6)

```
x = 3.14159  
ptor => y  
ptor = x      ! y = x
```

✓ $x = 3.14159$

- x 는 값을 할당 받음

✓ $ptor \Rightarrow y$

- $ptor$ 은 y 를 aliasing

✓ $ptor = x$

- $ptor$ 이 가리키는 공간(y)을 x 의 값으로 설정
- $ptor$ 에 대한 모든 참조와 할당은 y 에 대한 참조와 할당

포인터 선언과 할당 [6/6]

```
REAL, POINTER :: p1, p2
REAL, TARGET :: t1 = 3.4, t2 = 4.5
p1 => t1
p2 => t2
PRINT *, t1, p1      ! 3.4 printed out twice
PRINT *, t2, p2      ! 4.5 printed out twice
p2 => p1           ! Valid: p2 points to the target of p1
PRINT *, t1, p1, p2  ! 3.4 printed out three times
```

```
REAL, POINTER :: p1, p2
REAL, TARGET :: t1 = 3.4, t2 = 4.5
p1 => t1
p2 => t2
PRINT *, t1, p1      ! 3.4 printed out twice
PRINT *, t2, p2      ! 4.5 printed out twice
p2 = p1           ! Valid: equivalent to t2=t1
PRINT *, t1, t2, p1, p2 ! 3.4 printed out four times
```

포인터와 배열 (1/5)

❖ 차원, 타입 등이 맞으면

- ✓ 포인터는 배열 전체와 대응 가능
- ✓ triplet subscript로 표현된 배열 부분과 대응 가능

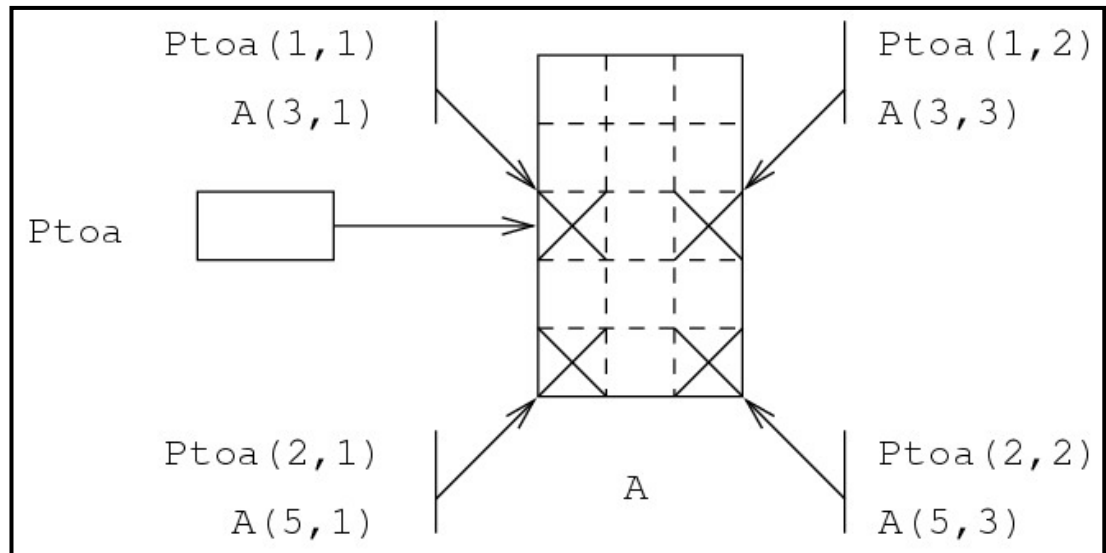
포인터와 배열 (2/5)

```
REAL, DIMENSION(:, :), POINTER :: Ptoa
```

```
Ptoa => A(3::2, ::2)
```

SIZE(Ptoa) = 4

SHAPE(Ptoa) = (/2,2/)



포인터와 배열 (3/5)

❖ 포인터와 배열의 대응

```
REAL, DIMENSION(:, :), POINTER :: Ptoa  
Ptoa => A(1:1, 2:2)
```

(O)

```
REAL, DIMENSION(:, :), POINTER :: Ptoa
```

```
Ptoa => A(1:1, 2)  
Ptoa => A(1, 2)  
Ptoa => A(1, 2:2)
```

(X)

```
REAL, DIMENSION(:, :), POINTER :: Ptoa
```

```
v = (/2,3,1,2/)  
Ptoa => A(v,v)
```

(X)

포인터와 배열 (4/5)

REAL, DIMENSION (:), POINTER :: pv1
REAL, DIMENSION (-3:5), TARGET :: tv1

pv1 => tv1 ! pv1 aliased to tv1



pv1 => tv1(:) ! aliased with section subscript



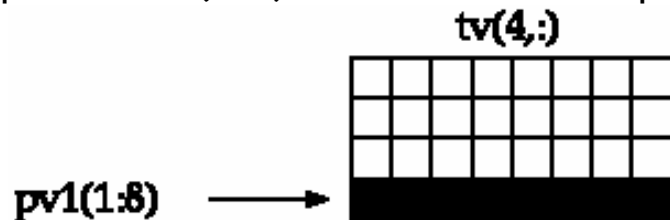
pv1 => tv1(1:5:2) ! aliased with section triplet



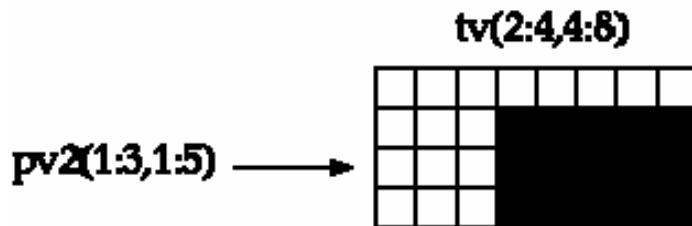
포인터와 배열 (5/5)

```
REAL, DIMENSION (:), POINTER :: pv1  
REAL, DIMENSION (:, :), POINTER :: pv2  
REAL, DIMENSION (4:8), TARGET :: tv
```

pv1 => tv(4, :) ! pv1 aliased to 4th row of tv



pv2 => tv(2:4, 4:8)



동적 타깃 (1/2)

❖ 동적 할당에 의해 생성되는 타깃 가능

```
ALLOCATE(ptor, STAT=ierr)  
ALLOCATE(ptoa(n*n, 2*k-1), STAT=ierr)
```

- ✓ 실수(ptor)와 2차원 배열(ptoa)을 위한 공간 할당,
이 공간은 포인터 ptor과 ptoa의 타깃이 됨

동적 타겟 (2/2)

- ❖ 포인터는 반드시 값 할당 이전에 Associated 공간을 가져야 한다.

```
ALLOCATE (ptor, STAT=ierr)  
ptor = 2.
```

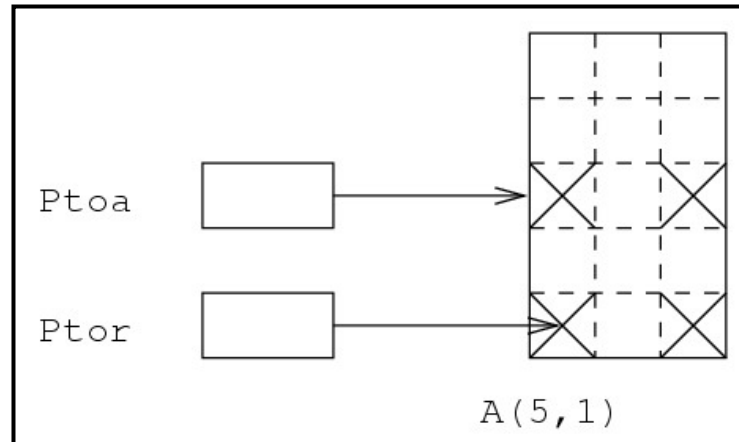
```
REAL, TARGET :: e  
ptor => e  
Ptor = 2.
```

자동 Attributing (1/2)

- ❖ 모든 포인터는 암시적인 TARGET attribute를 가지고 있어 다른 포인터에 associated 될 수 있다.

Ptoa => A(3::2, 1::2)

Ptor => Ptoa(2,1)



✓ ptor == A(5,1) == Ptoa(2,1)

자동 Attributing (2/2)

❖ Dangling 포인터

✓ 포인터 타겟의 경우 **dangling 포인터** 발생에 주의

```
REAL, POINTER :: p1, p2  
ALLOCATE (p1)
```

```
p1 = 3.4
```

```
p2 => p1
```

```
DEALLOCATE (p1)
```

! Dynamic variable p1 and p2 both pointed to is gone.
! Reference to p2 now gives unpredictable results

포인터 Disassociation (1/2)

❖ 포인터와 타깃의 Association 해제

✓ Nullification: **NULLIFY(ptor)**

- Association만 해제
- 타깃의 공간을 deallocate 하지 못함 → 메모리 낭비 위험

✓ Deallocation: **DEALLOCATE(ptoa, STAT=ierr)**

- ALLOCATE에 의해 association된 공간과 포인터의 연결을 해제
- 타깃 공간은 재사용 가능

포인터 Disassociation (2/2)

```
REAL, POINTER :: p           ! p undefined
REAL, TARGET :: t
PRINT *, ASSOCIATED (p)      ! not valid
NULLIFY (p)                  ! point at "nothing"
PRINT *, ASSOCIATED (p)      ! .FALSE.
p => t
PRINT *, ASSOCIATED (p)      ! .TRUE.
```

```
REAL, DIMENSION(:), POINTER :: p
ALLOCATE(p(1000))
NULLIFY(p)                   ! nullify p without first deallocating it!
                              ! big block of memory not released and unusable
```

포인터 Valued 함수 (1/3)

- ❖ 포인터를 결과로 주는 함수는 반드시 명시적 인터페이스를 가져야 한다.
- ❖ 결과의 크기가 수행되는 계산에 의존하는 경우 유용하게 사용될 수 있다.

포인터 Valued 함수 (2/3)

```
PROGRAM main
  IMPLICIT NONE
  REAL :: x
  INTEGER, TARGET :: a, b
  INTEGER, POINTER :: largest

  CALL RANDOM_NUMBER(x)
  a = 10000.0*x
  CALL RANDOM_NUMBER(x)
  b = 10000.0*x
  largest => ptr()

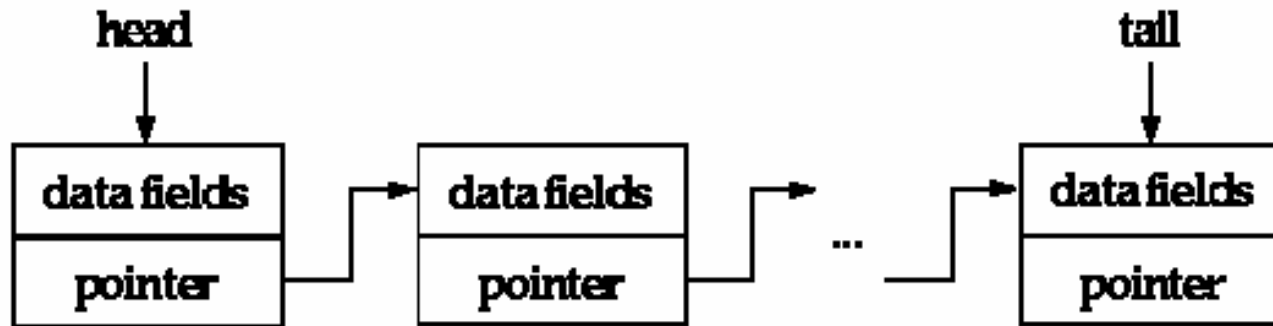
CONTAINS
  FUNCTION ptr()
    INTEGER, POINTER :: ptr
    IF (a .GT. b) THEN
      ptr => a
    ELSE
      ptr => b
    END IF
  END FUNCTION ptr
END PROGRAM main
```

포인터 Valued 함수 (3/3)

```
INTEGER, DIMENSION(100) :: x
INTEGER, DIMENSION(:), POINTER :: p
...
p => gtzero(x)
...
CONTAINS                                ! function to get all values .gt. 0 from a
  FUNCTION gtzero(a)
    INTEGER, DIMENSION(:), POINTER :: gtzero
    INTEGER, DIMENSION(:) :: a
    INTEGER :: n
    ...                                ! find the number of values .gt. 0 (put in n)
    ALLOCATE (gtzero(n))
    ...                                ! put the found values into gtzero
  END FUNCTION gtzero
...
END
```

Linked List (1/3)

❖ data field와 포인터 field로 구성



✓ 연결되는 객체들은

- 연속적으로 저장될 필요 없음
- 동적 생성, 동적 삭제 가능
- list내 임의 위치에 삽입 가능

Linked List (2/3)

TYPE node

INTEGER :: value ! data field

TYPE (node), POINTER :: next ! pointer field

END TYPE node

INTEGER :: num

TYPE (node), POINTER :: list, current

NULLIFY(list) ! initially nullify list (mark its

end)

DO

READ *, num ! read num from keyboard

IF (num == 0) EXIT ! until 0 is entered

ALLOCATE(current) ! create new node

current%value = num

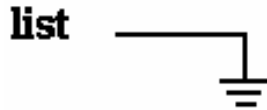
current%next => list ! point to previous one

list => current ! update head of list

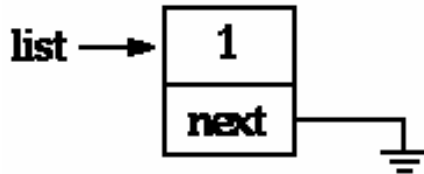
END DO

Linked List (3/3)

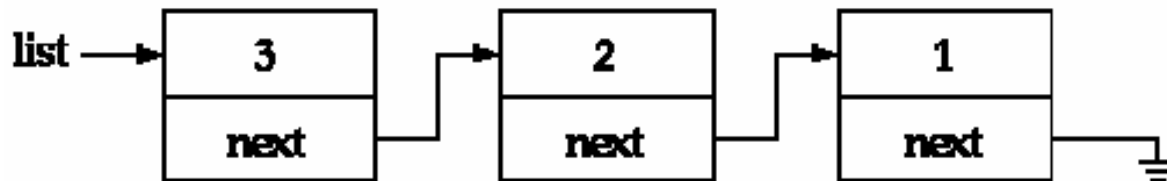
☑ NULLIFY(list)



☑ 첫 번째 num 읽은 후



☑ 세 개의 num을 모두 읽은 후



➤ 연산자 Overloading과 사용자 정의 연산자

- 연산자 Overloading
- 사용자 정의 연산자
- 할당 Overloading



연산자 Overloading (1/2)

❖ 고유 연산자를 추가적인 데이터 타입에 적용하도록 그 의미를 확장해 사용하는 것

- ✓ Generic 연산자 심볼 정의
 - INTERFACE OPERATOR문 사용
- ✓ Generic 인터페이스로 overload set 정의
- ✓ 연산 수행을 정의하는 모듈 프로시저 선언

```
INTERFACE OPERATOR (intrinsic_operator)
    interface_body
END INTERFACE
```

연산자 Overloading (2/2)

```
MODULE over
  INTERFACE OPERATOR (+)
    MODULE PROCEDURE concat
  END INTERFACE
  CONTAINS
    FUNCTION concat(cha, chb)
      CHARACTER (LEN=*), INTENT(IN) :: cha, chb
      CHARACTER (LEN=LEN_TRIM(cha) + LEN_TRIM(chb)) :: concat
      concat = TRIM(cha) // TRIM(chb)
    END FUNCTION concat
END MODULE over
```

```
PROGRAM testadd
  USE over
  CHARACTER (LEN=23) :: name
  CHARACTER (LEN=13) :: word
  name='Balder'
  word='convoluted'
  PRINT *, name // word
  PRINT *, name + word
END PROGRAM testadd
```

사용자 정의 연산자 [1/2]

❖ 새로운 연산자 표현: **.<name>.**

- ✓ 연산자 overloading과 같은 방법으로 정의
- ✓ 연산자 이름 <name>에는 문자만 가능
- ✓ 하나 또는 두 개의 INTENT(in) 인수가 있어 단항 또는 이항 연산을 표현
- ✓ 단항 연산은 이항 연산보다 우선 순위 높음

사용자 정의 연산자 (2/2)

```
MODULE distance_module
  TYPE point
    REAL :: x, y
  END TYPE point
  INTERFACE OPERATOR (.dist.)
    MODULE PROCEDURE calcdist
  END INTERFACE
  CONTAINS
    REAL FUNCTION calcdist (px, py)
      TYPE (point), INTENT(IN) :: px, py
      calcdist = SQRT ((px%x-py%x)**2 + (px%y-py%y)**2 )
    END FUNCTION calcdist
END MODULE distance_module
```

```
PROGRAM main
  USE distance_module
  TYPE (point) :: p1, p2
  REAL :: distance

  ...
  distance = p1 .dist. p2
  ...
END PROGRAM main
```

할당 Overloading (1/3)

❖ 서로 다른 유도 타입 또는 고유 타입과 유도 타입 사이의 할당을 위한 명시적 정의

- ✓ 연산자 Overloading과 동일
- ✓ 두 개의 인수를 가지는 서브루틴으로 정의
 - 첫 인수는 `INTENT(out)`, 실제 할당에서 `LHS`에 해당
 - 두 번째 인수는 `INTENT(in)`, 실제 할당에서 `RHS`에 해당

할당 Overloading (2/3)

```
MODULE assignoverload_module
  TYPE point
    REAL :: x,y
  END TYPE point
  INTERFACE ASSIGNMENT (=)
    MODULE PROCEDURE assign_point
  END INTERFACE
  CONTAINS
    SUBROUTINE assign_point(ax, px)
      REAL, INTENT(OUT)::ax
      TYPE (point), INTENT(IN)::px
      ax = MAX(px%x, px%y)
    END SUBROUTINE assign_point
END MODULE assignoverload_module
```

할당 Overloading (3/3)

```
PROGRAM main
USE assignoverload_module
REAL :: ax
TYPE (point) :: px
...
ax = px ! type point assigned to type real
        ! not valid until defined
...
END PROGRAM main
```


➤ 되부름(Recursive) 프로시저

▪ 되부름 프로시저



되부름 프로시저 (1/3)

❖ 되부름

- ✓ 프로시저가 직간접적으로 자기 자신 호출
- ✓ Fortran 90에서 명시적으로 지원
- ✓ 되부름 프로시저는 RECURSIVE로 선언
 - 되부름 함수는 RESULT로 선언된 변수를 통해 리턴

```
INTEGER RECURSIVE FUNCTION fact(n) RESULT(n_fact)
```

```
RECURSIVE INTEGER FUNCTION fact(n) RESULT(n_fact)
```

```
RECURSIVE FUNCTION fact(n) RESULT(n_fact)  
INTEGER n_fact
```

되부름 프로시저 (2/3)

```
PROGRAM main
```

```
IMPLICIT NONE
```

```
PRINT *, fact(5)
```

```
CONTAINS
```

```
  RECURSIVE FUNCTION fact(n) RESULT(n_fact)
```

```
    INTEGER, INTENT(in) :: n
```

```
    INTEGER :: n_fact
```

```
    ! also defines type of fact
```

```
    IF (n == 1) THEN
```

```
      n_fact = 1
```

```
    ELSE
```

```
      n_fact = n * fact(n - 1)
```

```
    END IF
```

```
  END FUNCTION fact
```

```
END PROGRAM main
```

되부름 프로시저 (3/3)

```
PROGRAM main
```

```
IMPLICIT NONE
```

```
INTEGER :: result
```

```
CALL factorial(5, result)
```

```
PRINT *, result
```

```
CONTAINS
```

```
  RECURSIVE SUBROUTINE factorial(n, n_fact)
```

```
    INTEGER, INTENT(in) :: n
```

```
    INTEGER, INTENT(inout) :: n_fact
```

```
    IF (n == 1) THEN
```

```
      n_fact = 1
```

```
    ELSE
```

```
      CALL factorial(n-1, n_fact)
```

```
      n_fact = n_fact * n
```

```
    END IF
```

```
  END SUBROUTINE factorial
```

```
END PROGRAM main
```

➤ Keyword/Optional 인수

- Keyword 인수
- Optional 인수



Keyword 인수 (1/2)

❖ 고유 함수 인수에 Keyword 사용 가능

```
READ(10,67,789) x, y, z
```

```
READ(UNIT=10,FMT=67,END=789) x, y, z
```

❖ 프로시저 인수에 Keyword 사용 가능

✓ 위치에 의한 인수 대응(F77, F90)

✓ Keyword에 의한 인수 대응(F90)

- 명시적 인터페이스를 가질 때 사용 가능
- Keyword = 인수 이름

Keyword 인수 (2/2)

```
SUBROUTINE axis(y0, y0, l, min, max, i)
  REAL, INTENT(in) :: x0, y0, l, min, max
  INTEGER, INTENT(in) :: l
  ...
END SUBROUTINE axis
```

☑ 위치 대응

CALL axis(0.0, 0.0, 100.0, 0.1, 1.0, 10)

☑ Keyword 대응

CALL axis(0.0, 0.0, max=1.0, min=0.1, l=100.0, i=10)

Optional 인수 (1/2)

❖ 프로시저 인수의 선택적 사용 가능

- ✓ OPTIONAL로 선언된 인수는 리스트에서 생략 가능
- ✓ 생략된 인수 다음은 모드 Keyword 대응
- ✓ 명시적 인터페이스를 가지는 프로시저에서 가능
- ✓ **PRESENT(argument_name)**
 - optional 인수의 상태 확인

Optional 인수 (2/2)

```
SUBROUTINE SEE(a, b)
  IMPLICIT NONE
  REAL, INTENT(in), OPTIONAL :: a
  INTEGER, INTENT(in), OPTIONAL :: b
  REAL :: ay; INTEGER :: bee
  ay = 1.0; bee = 1

  IF(PRESENT(a)) ay = a
  IF(PRESNET(b)) bee = b
  ...
```

```
CALL SEE()
CALL SEE(1.0, 1); CALL SEE(b=1, a=1.0)    ! same
CALL SEE(1.0);   CALL SEE(a=1.0)         ! same
CALL (b=1)
```

참고자료

1. The POWER4 Processor Introduction and Tuning Guide (<http://www.ibm.com/redboosk>)
2. Fortran90 and Computational Science ([http://www.comphys.uni-
duisburg.de/Fortran90/pl/pl.html](http://www.comphys.uni-duisburg.de/Fortran90/pl/pl.html))
3. The Liverpool Fortran90 Courses (<http://www.liv.ac.uk/HPC/F90page.html>)
4. Language Reference, XL Fortran for AIX, Version 8 Release 1
5. The Fortran90 Essentials: Discussion ([http://www.tc.cornell.edu/Services/Edu
/Topics/Fortran90/more.asp](http://www.tc.cornell.edu/Services/Edu/Topics/Fortran90/more.asp))
6. Fortran90 for the Fortran77 Programmer ([http://www.who.edu/CIS/training/
classes/f77to90/f77to90.html](http://www.who.edu/CIS/training/classes/f77to90/f77to90.html))
7. Larry R. Nyhoff, Sanford C. Leestma. *Fortran90 for Engineers and Scientists*. Prentice Hall. 1997.
8. Walter S. Brainerd, Charls H. Goldberg, Jeanne C. Adams. *Programmers Guide to Fortran90 3rd edition*. Springer. 1995.

기술지원

❖ Helpdesk

✓ www.hpcnet.ne.kr

❖ 교육센터 게시판

✓ webedu.hpcnet.ne.kr

❖ E-Mail

✓ consult@supercomputing.re.kr