차 례

제 1 장 Fortran 90 의 개요

1950 년대 후반 IBM 에서 개발된 Fortran 은 가장 오래된 고급언어로, 처음에는 IBM Mathematical FORmula TRANslation System 이란 긴 이름을 가지고 있었으며, 이후 줄여서 FORmular TRANslation, 즉 Fortran 이라 불리게 되었다. Fortran 표준은 1970 년대 후반에 ANSI 에 의해 새로운 표준으로 업데이트 되었고, 1980 년도에 ISO 에 의해 국제표준으로 채택되었는데 이것이 지금까지 널리 사용되어지는 Fortran77 이다(초안이 1977 년에 완성 되었기 때문에 77 이란 이름이 붙었다).

Fortran90 은 Fortran77 을 획기적으로 발전시킨 것이지만 Fortran77 의 모든 것을 포함하고 있다. 그래서 Fortran77 표준을 따르는 어떤 프로그램도 Fortran90 프로그램으로 사용될 수 있다. Fortran90 은 Fortran77 의 모든 기능에 프로그램 작성을 보다 편리하게 하는 다수의 새로운 특징들을 포함하고 있으며, 특히 객체 지향의 프로그래밍이 가능하고 현재 개발된 프로그래밍 언어 중에서 배열을 다루기에 가장 편리한 고수준의 문법을 갖추고 있다.

본 기술서에서는 앞으로 3 회에 걸쳐 Fortran77 에서 새롭게 추가된 것들을 중심으로 계산과학 분야에 주로 사용되는 Fortran90 프로그래밍 언어의 특징과 기능들에 초점을 맞춰 소개할 것이다.

1.1 Fortran77 의 단점

Fortran77 은 다음과 같은 단점을 가지고 있다.

- Fortran77 은 "천공카드" 또는 "고정형식"의 소스코드 포맷을 가진다.

  Fortran77 프로그램의 각 줄은 72 열까지만 허용된다.

  각 줄의 첫 5 열은 줄 번호를 적는 곳이다.

  각 줄의 제 6 열은 줄 연결을 나타내는 것만으로 사용된다.

  주석은 반드시 첫 1 열에서 특정 문자로 시작되어야 하고 프로그램 코드가 쓰여진 줄에 삽입되어 같이 쓰여질 수 없다.

  변수 이름이 6 글자로 제한된다.

- 본질적으로 병렬 연산을 표현할 수 없다.

- 동적 저장을 지원하지 않는다.

- 수치적 이식성 부족 : Fortran77 코드를 다른 시스템으로 포팅하는 경우 각 시스템간의 정밀도 표현이 달라 문제가 발생한다.
- 새로운 자료구조를 사용자가 정의해 사용할 수 없다.
- 재귀적 호출(자기 자신을 호출해 사용)을 지원하지 않는다.
- COMMON 블록, EQUIVALENCE 문 등의 불안정성

1.2 새로운 기능 소개

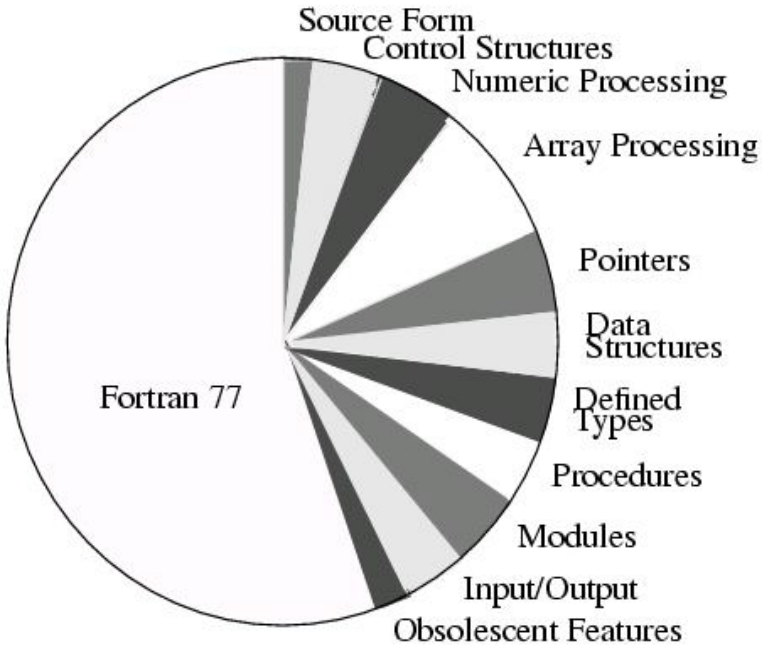


그림 1.1 Fortran90 (출처 :http://www.comphys.uni-duisburg.de/Fortran90/pl/pl.html)

그림 1은 Fortran90을 구성하는 주요 내용들을 나타내고 있다. Fortran90은 Fortran77에 새로운 기능들이 추가된 것이다. Fortran90에서 새롭게 추가된 특성을 여기서 간략히 소개한다. 소개되는 내용에 대한 보다 자세한 설명과 사용법, 예제 등은 3, 4, 5장에서 다루게 될 것이다.

1.2.1 개선된 문법

- 자유형식 - 어떤 위치에서 문장이 시작되어도 상관없다.
- 한 줄에 132열까지 사용 가능[1]
- 한 줄에 한 문장 이상 가능 - 문장 분리 기호 ';'
- 문장 내에 주석을 달 수 있음 - 주석 시작 기호 '!'
- 줄 바꿈 표시 기호 '&'
- 변수 이름 31자까지 사용 가능[2] (반드시 문자로 시작되어야 한다.)
- 관계 연산자 .LT., .LE., .EQ., .NE., .GE., .GT.를 각각 <, <=, ==, /=, =>, >로 대체 가능

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

※ Fortran90에서 사용 가능한 문자 집합

- 문자-숫자 조합 : a-z, A-Z, 0-9, _ (underscore)
- 기호(Fortran90에서 새로 추가된 기호는 굵게 표시)

1. IBM XLFortran 에서 한 문장의 최대 길이는 6700 문자
2. IBM XLFortran 은 250 자 까지 사용 가능

기 호	기 술	기 호	기 술
	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		

1.2.2 새로운 방식의 Type Declarations 과 Attributing

Fortran77	Fortran90
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	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

1.2.3 새로운 Control Structures

- 정수 또는 문자 표현식에 대한 SELECT-CASE문

  IF문 같은 순차적 선택이 아닌 병렬적 선택이 가능하도록 한다.

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

```
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
```

- 인덱스 없는 새로운 DO 구문

  DO - END DO [1]

  DO WHILE - END DO

- 인덱스에 의한 계산을 배열 연산으로 대체함으로써 데이터 병렬성을 가지는 계산에서 성능 이득을 얻을 수 있다.

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

1. END DO 는 Fortran77 표준이 아니다 .

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

1.2.4 Numeric Processing

여러 유용한 모델 값을 리턴하는 고유 함수들과 kind 값을 리턴하는 고유 함수들, 그리고 기타 일반적인 연산들에 대해 2장에서 다룰 것이다.

1.2.5 Array Processing

배열 연산은 Fortran90의 가장 중요한 부분이며 이는 4장에서 자세하게 다룰 것이다.

1.2.6 Pointers

Fortran90에서 포인터는 동적 자료 구조와 동적 배열이라는 두 가지 중요한 성능을 제공한다. Fortran90의 포인터는 포인터 target으로 명시적으로 선언된 데이터 object, 동적으로 생성된 object 또는 다른 포인터를 가리킨다(point to). 관련 내용은 5장에서 자세히 다룰 것이다.

```
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
```

1.2.7 사용자 정의 타입과 연산자

새로운 사용자 정의 타입은 기존의 데이터 타입을 이용해 정의되며, TYPE - END TYPE 구문을 사용한다. 예를 들어, 하나의 유리수를 분자와 분모로 표현하는 새

로운 타입을 정의하고 이러한 타입을 가지는 변수들을 다음과 같이 선언할 수 있다.

```
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
```

위와 같이 새롭게 정의된 타입을 가지는 변수들의 연산을 위해서 적절한 연산들을 정의할 필요가 있다. 사용자 정의 연산자는 사용자가 만든 함수와 이에 대한 연산자 인터페이스의 정의로 표현된다. 다음 예는 데이터 타입 RATIONAL로 정의된 변수 사이의 덧셈을 위해 확장된 "+" 연산자의 인터페이스를 정의한 것이다.

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

사용자 정의 데이터 타입과 연산자에 대한 보다 자세한 내용은 5장에서 다룰 것이다.

1.2.8 자료구조

배열은 과학계산 분야에서 가장 중요하고 가장 많이 사용되는 자료형이지만 동적 연결구조(Dynamically Linked Structures)와 같은 다른 형식의 자료구조 또한 필요하다. 다음 예는 동적연결구조를 가지는 데이터 타입을 정의한 것이다. Fortran90에서, 자료구조는 사용자 정의 타입을 가지는 객체(혹은 변수)로 선언되어 사용될 수 있다.

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

1.2.9 프로시저

Fortran90 프로시저는 다음과 같은 새로운 특성을 가진다.

- 함수의 결과 값이 배열 또는 다른 자료구조가 될 수 있다.
- 자기 자신을 호출하는 recursive 계산이 가능하다.
- 프로시저 인터페이스를 명시적으로 정의할 수 있다.
- 서브루틴내에 새로운 서브루틴을 두는 내장 프로시저 사용이 가능하다.
- 인수 사용에 대해 OPTIONAL, INTENT(IN) 등의 속성을 지정할 수 있다.

1.2.10 모듈

모듈은 Fortran90 에서 지원하는 프로그램의 새로운 단위로서 주 프로그램(main program) 또는 외부 부 프로그램(external subprogram)과 같은 실행 프로그램이 편리하게 접근해 사용할 수 있는 정의들을 포함한다.

- 사용자 정의 타입의 정의
- 상수 또는 변수의 선언 - 초기화도 가능하다.
- 프로시저 정의
- 외부 프로시저에 대한 인터페이스 정의
- generic 프로시저의 명명과 연산자 기호의 선언

```
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
```

모듈과 관련된 내용은 5장에서 다룰 것이다.

1.2.11 I/O

입출력과 관련된 Fortran90의 대표적인 특징으로 NAMELIST와 Non-advancing I/O 기능 등이 있으며 5장에서 다룰 것이다.

1.3 Fortran90의 객체지향성

최근의 가장 유행하는 프로그래밍 패러다임은 객체지향 프로그래밍이다. 객체지향 프로그래밍은 소프트웨어의 reusability, modularity를 좋게 하며, 어떤 타입을 가지는 객체의 처리를 위해 프로시저와 타입들에 대한 정의를 하나로 묶어(이 묶음이 하나의 객체가 된다) 오류 발생 조건을 사전에 제거한다. Fortran90은 C++와 같은 광범위한 개념의 객체지향 프로그래밍 능력을 가지고 있지는 못하지만 프로그래머가 사용하기에 충분한 객체지향성을 가지고 있다.

다음은 Fortran90이 가지는 객체지향성을 정리한 것이며, 각각의 내용들에 간단히 소개한다. 소개되는 내용에 대한 보다 자세한 설명과 사용법, 예제 등은 3, 4, 5장에서 다루게 될 것이다.

- data abstraction : 사용자 정의 타입
- data hiding : PRIVATE과 PUIBLIC 속성
- encapsulation : 모듈과 data hiding 특성

- inheritance와 extensibility : super-type, 연산자 overloading과 generic 프로시저

- polymorphism : generic overloading을 이용한 프로그래밍

- reusability : 모듈

1.3.1 Data Abstraction

Fortran90은 사용자 정의 타입으로 일정 수준의 데이터 abstraction 기능을 제공하며, 포인터를 이용해 복잡한 자료구조의 정의를 가능하게 한다. Fortran90의 포인터는 다른 언어들의 그것과는 좀 달리 구현되는데, C의 포인터보다 유연성 측면에서는 부족하지만 효율적인 면에서는 오히려 더 우수하다. Fortran90의 포인터는 강형(strongly typed)[1] 이며, 포인터의 target이 제한적이지만 이로 인해 보다 안전하고 빠른 코드 작성을 가능하게 한다. Fortran90은 열거형 타입을 지원하지는 않지만 모듈을 이용해 큰 불편 없이 구현이 가능하다.

Fortran90에서 부족한 점은 매개변수 정의 타입(parameterized derived types)과 서브타입(subtype)에 대한 지원이 없다는 것이다. 매개변수 정의 타입은 사용자 정의 타입에서 각 타입 종류의 선택을 매개변수를 통해 할 수 있는 것을 의미한다. 예를 들어 Fortran90에서 다음에 있는 두 개의 사용자 정의 타입은 각각 독립적으로 존재하는 것이지만, 매개변수 정의 타입에서는 각 타입에 대한 skeleton만 정의하고 KIND와 같은 매개변수를 통해 선택적으로 사용할 수 있도록 한다.

```
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
```

서브타입은 부모 타입의 부분집합을 의미한다. 예를 들어, 기본 INTEGER 타입과 그 특성이 같지만 제한된 범위만 다른 자연수 타입을 생각해 볼 수 있다. 서브타입에 대한 구현은 어렵지만 변수의 제한범위 점검에 대한 안전성을 제공한다.

1. 모든 객체의 형을 정적으로 정하고 형 일치 및 그에 대한 규칙을 완벽하게 정의하여 번역시에 형 오류를 검출할 수 있는 프로그래밍 언어 .

1.3.2 Data Hiding

기본적으로 모듈 내의 모든 객체는 USE문을 가진 모든 프로그램 unit에서 접근할 수 있다. Fortran90에서는 PRIVATE 또는 PUBLIC 속성을 이용해 모듈 내의 객체나 프로시저에 대한 접근을 제한함으로써 data hiding을 지원하고 있다.

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

1.3.3 Encapsulation

Encapsulation은 여러 관련된 기능 혹은 객체들을 하나의 라이브러리 혹은 패키지로 정의함으로써 사용자가 복잡한 내부구조에 대한 이해 없이 관련 기능을 사용할 수 있도록 하는 것을 의미한다. Fortran90에서의 encapsulation은 data hiding, MODULE/MODULE PROCEDURE와 USE문의 사용, 그리고 모듈 내의 데이터 객체에 대한 재명명(rename), 특정 개체에 대한 선택적 접근 등에 의해 이루어 진다.

```
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 accessedUSE
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
```

1.3.4 Inheritance and Extensibility

Fortarn90은 super-type을 지원한다. 여기서 super-type이란 다른 정의된 타입을 포함하는 사용자 정의 타입을 말하며, 이렇게 하나의 계층적 구조가 형성된다. Fortran90은 서브타입을 지원하지 않기 때문에 객체나 기능에 대한 계층상의 상속이 이루어 지지는 않는다. 상속은 객체지향 프로그래밍에서 매우 중요한 개념이지만 Fortran90에서는 이것을 지원하지 않는다.

1.3.5 Polymorphism

Generic 프로시저에 의해 구현되는 Fortran90의 polymorphism은 다음 2장에 설명되어 있다. Fortran90의 generic 특성은 진보된 것이지만, 특정 프로시저를 항상 사용자가 작성해서 generic 인터페이스에 첨가해야 한다.

1.3.6 Reusability

모듈은 다른 모듈에 의해 사용 가능한 함수와 자료구조의 선언 등에 대한 라이브러리를 구성하며, 이렇게 모듈에 포함된 내용은 언제든지 다른 프로그램 unit에 의해 접근될 수 있고 이를 통해 사용자는 reusability가 좋은 코드를 손쉽게 작성할 수 있다.

1.4 Fortran77, C, C++와 Fortran90의 비교

탁월한 안정성과 다양한 라이브러리 지원 등으로 인해 Fortran은 계산과학 분야에서 가장 많이 사용되는 프로그램이었다. 그러나, 최근에 와서 동적 자료구조, 유닉스 워크스테이션, 대화식 가시화 능력, 병렬구조 등에 대한 중요성이 대두 되면서 C 또는 C++와 같은 새로운 언어가 계산과학 분야에서 많은 관심을 받고있다. 이에 Fortran은 Fortran90으로 진화 하면서 현대 계산과학 분야에서 필요로 하는 여러 가지 능력들을 갖추게 되었는데 다음 표는 Fortran77, C, C++와 Fortran90을 몇 가지 측면에서 비교한 것을 순위로 나타내 요약한 것이다.

Functionality	Fortran77	C	C++	Fortran90
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

표 1. 계산과학을 위한 언어들의 상대적 순위 (출처 :http://www.comphys.uni-duisburg. de / Fortran90/pl/pl.html)

1.4.1 Numerical Robustness

Numeric polymorphism, kind 타입, 정밀도 선택, numeric environmental inquiry 등으로 인해 Fortran90이 1위가 됐으며, 복소수 타입의 지원으로 Fortran77이 2위, C++는 C보다 polymorphism 측면에서 우월해 그 다음 순위가 되었다. Fortran90의 numerical robustness는 2장에서 상세히 다룰 것이다.

1.4.2 Data Parallelism

Data parallel은 Fortran90만 지원하고 있으며 이에 대해서는 3장에서 다루게 될 것이다.

1.4.3 Data Abstraction

Fortran90은 사용하기 편리하고 효율적인 data abstraction 능력을 가지고 있다. C++는 object oriented programming의 한 부분으로서 data abstraction 능력을 가지며, 이는 계산과학 분야에서 사용하기에 Fortran90보다 복잡하다. C는 자료구조의 지원 측면에서 Fortran77보다 나은 능력을 가진다.

1.4.4 Object Oriented Programming

Fortran90은 자동상속을 지원하지 않아 C++보다는 못하지만 polymorphic 특성의 수동상속이 가능해 Fortran77이나 C보다는 좋은 순위를 가진다. 이 부분에서도 C는 자료구조의 지원 측면에서 Fortran77보다 더 앞선다.

1.4.5 Functional Programming

Recursion과 자료구조 지원의 부족으로 Fortran77은 최하위가 되었으며, Fortran90은 필요할 때만 값을 계산하는 lazy evaluation 기능으로 최고점을 받았고 lazy evaluation은 지원하지 않지만 polymorphism을 지원하는 C++가 C보다 좋은 점수를 받았다.

제 2 장 Numerical Robustness

많은 계산과학 분야의 문제는 수학적 모델을 수치적으로 전산실험 하는 것이므로 수치 계산은 계산과학 분야에서 핵심적인 부분이며, 따라서 수치적 기능은 매우 중요하다. 수치적인 기능은 단정도와 배정도 실수 타입, 복소수(단정도) 데이터 타입과 다양한 수치계산 문제에서 사용할 수 있는 풍부한 함수 라이브러리 등으로 구성되며 때때로, 배정도 복소수 타입과 4 배정도 실수 데이터 타입과 같은 기능이 추가되기도 한다. 대부분의 과학계산 문제는 이러한 수치적 기능만으로 충분하다.

그러나, 경우에 따라서 위에서 언급된 수치적 기능만을 사용하는 것보다 현재 수치적 implementation 환경에 대한 더 많은 정보를 사용하거나 수치적 정밀도를 더 높여줌으로써 수렴성에 대한 보장이나 결과의 정밀도를 높이는 등의 수치계산 성능을 높일 수 있다. 여기서는 "type kind" 기구를 이용한 수치적 정밀도 선택과 수치적 implementation 환경에 대한 정보를 얻는 numeric approximation model, environmental intrinsic function 등에 대해 알아본다.

2.1 Numeric Kind Parameterization

Fortran90 의 KIND mechanism 은 Fortran77 에서 공통적으로 사용되던 "*size" 형식을 일반화, 정식화, 표준화 시킨 것이다. "*size" 문법은 서로 다른 종류의 실수 타입을 변수 size 를 이용해 형식화 한 것으로 Fortran77 에서 단정도 REAL 은 REAL*4 와 같고 DOUBLE PRECISON 은 REAL*8 과 동일하다. Fortran90 에서는 고유의 데이터 타입마다 정수의 kind 값을 대응시키는 타입 kind 개념을 정식화 했으며 이러한 kind 값들은 implementation dependent 하게 설정 되어진다. 만약 단정도 REAL 에 4 를 DOUBLE PRECISION 에 8 을 대응 시켰다면 REAL 과 REAL(4), REAL(KIND=4) 는 동일한 표현이며, DOUBLE PRECISION 과 REAL(8), REAL(KIND=8) 은 동일한 표현이다. Fortran90 의 REAL 타입 선언 문법은 다음과 같다.

```
REAL [([KIND= ] kind-value)]
```

어떤 시스템에서는 단정도 REAL 의 크기가 4 바이트가 아닐 수 있으며, 혹은 단정도 REAL 을 표현하는 방식도 하나 이상이 될 수도 있다. 이런 경우, Fortran90 은 다른 타입의 실수 표현에는 다른 kind 값을 대응시킬 뿐이다. 모든 시스템이나 implementation 에 적합한 명확한 kind 값의 집합은 없으며 따라서, kind 값은 implementation dependent 하게 결정된다.

고유함수 KIND를 사용하면 코드작성을 implementation의 kind 값 설정에 무관하도록 할 수 있다. KIND함수는 입력으로 들어온 인수에 대해 현재 implementation에서의 kind 값을 정수로 리턴해 주는 Fortran90의 고유함수 이다.

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

2.2 정밀도 선택

Fortran90의 고유함수 SELECTED_REAL_KIND는 최소의 소수점 정밀도와 지수 범위를 알려줌으로써 사용자가 원하는 정밀도의 변수를 사용할 수 있도록 해준다. SELECTED_REAL_KIND는 사용자가 원하는 소수 정밀도와 지수범위를 나타내는 두 개의 선택적인 정수 인수를 가지는데 적어도 하나는 있어야 한다. SELECTED_REAL_KIND는 현재의 implementation에서 입력 조건에 맞는 정밀도를 지원하는 최소의 실수 데이터 타입 kind 값을 리턴 해준다. 아래의 예에서 P9은 소수 9번째 자리까지의 정밀도를 나타낼 수 있는 정수의 kind 값을 함수 SELECTED_REAL_KIND로부터 리턴 받은 것이다.

```
INTEGER, PARAMETER :: P9 = SELECTED_REAL_KIND(9)
REAL(KIND=P9)      ...9 digit (at least) precision real variables...
.....
```

IBM 시스템에서 P9은 DOUBLE PRECISION 혹은 앞 절의 DOUBLE과 동일하다. 그러나 Cray 시스템에서는 단정도 실수 혹은 앞 절의 SINGLE과 동일하다.

실수 상수들은 어떤 종류라도 다음과 같이 "_"에 kind 값을 붙여 나타낼 수 있다.

```
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)
```

2.3 Numeric Polymorphism

Fortran90의 모든 계산관련 고유함수 들은 implementation에 의해 제공되는 모든 타입에 대해 generic 하다. 예를 들자면, 고유함수 COS(X)의 결과는 인수 X가 SINGLE, DOUBLE 또는 P9이냐에 따라 각각 SINGLE, DOUBLE, P9으로 리턴된다. Fortran77에서도 고유함수는 역시 generic 하지만 사용자나 혹은 다른 벤더에 의해 제공되는 프로시저는 generic 하지 않다. Fortran90은 이러한 Fortran77의 결점을 보완하고 있다. 여기서 말하는 Polymorphism은 사용자 정의 프로시저까지 확장된 Fortran90의 generic 속성을 의미한다.

사용자가 만들어 제공하는 프로시저 집합에 대해 generic 이름을 붙이거나 기존의 generic 이름에 새로운 프로시저를 추가하기 위해 interface block을 사용한다. 서로 다른 타입의 인수를 가지는 프로시저들을 generic interface로 묶어 사용하고 generic으로 묶인 프로시저를 프로그램에서 호출할 때는 하나의 generic 이름으로 호출하지만 입력 인수 타입에 의해 각 프로시저가 구분된다.

- INTEGER와 REAL을 swap하는 external subroutine

```
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
```

- generic swap routine "swap"을 만들고 프로그램에서는 generic 이름 "swap"으로 호출한다. 이때 인수 타입이 generic interface의 어떤 프로시저를 호출하는지를 결정한다.

```
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)
```

다음의 예는 interface block을 이용하여 네 개의 프로시저를 묶어 새로운 generic 이름(SMOOTH)를 정의한 것이다. 인수 AA는 각 경우마다 타입이 다르지만 함수의 결과는 모두 INTEGER로 동일하다. 그렇지만, 함수의 결과는 모든 경우 혹은 일부분에 대해 다른 타입을 가질 수도 있다.

```
INTERFACE SMOOTH                          ! SMOOTH is the generic name
  INTEGER FUNCTION SMOOTH_INT(AA)         ! for procedures   SMOOTH_INT
    INTEGER :: AA(:,:)                    ! SMOOTH_SINGLE
  END FUNCTION SMOOTH_INT                 ! SMOOTH_DOUBLE
                                          ! SMOOTH_RATIONAL
  INTEGER FUNCTION SMOOTH_SINGLE(AA)
    REAL(SINGLE) :: AA(:,:)               ! AA is an assumed shape two-
  END FUNCTION SMOOTH_SINGLE              ! dimensional array in each case.

  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
```

이미 존재하는 generic 이름에 새로운 프로시저를 추가할 수도 있다. 다음 예는 고유함수 COS에 인수 타입 RATIONAL에 대한 연산 프로시저를 추가한 것이다.

```
INTERFACE COS                              ! Extends the generic properties
  FUNCTION RATIONAL_COS(X)                 !  of COS to return results of
    TYPE(RATIONAL) :: RATIONAL_COS         !  type RATIONAL, assuming the
    TYPE(RATIONAL) :: X                    !  argument is of type RATIONAL.
  END FUNCTION RATIONAL_COS
END INTERFACE
```

2.4 Numeric Approximation Model

이식 가능하고 수치적으로 robust 한 프로그램 개발을 위해 implementation 의 numerical 특성에 동적으로 접근할 수 있도록 하는 것이 중요하다. Fortran90 은 실수 근사 모델과 모델 관련 값을 알아볼 수 있는 16 개의 고유함수를 제공해서 그러한 접근이 가능하도록 하고 있다.

Fortran90 에서 각 종류의 실수들은 다음과 같이 모델링 되어 진다.

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

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_1 may be 0 only if all f_i are 0

각 실수 타입에 대한 주요 특징은 b 와 p 그리고 e 의 범위에 의해 결정되는데 대부분 implementation 에서 서로 다른 실수 종류는 p 에 의해 결정된다.

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

2.5 Environmental Inquiry

각 종류의 실수를 특징짓는 미리 정의된 9 개의 모델 관련 값이 있다. 사용자는 environmental inquiry 고유함수를 이용해 데이터 objects의 선언이나 기타의 계산에서 필요한 경우 언제든지 이 값들에 접근할 수 있다. 각 함수는 단 하나의 인수를 가지는데 이 인수는 상수이거나 스칼라 변수 또는 배열이 될 수 있다.

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

표 2.1 Environmental intrinsic functions

다음은 위의 코드를 KISTI IBM 시스템에서 실행한 결과이다.

```
$a.out
PRECISION = 6
HUGE = 0.3402823466E+39
RADIX = 2
DIGITS 24
MINEXPONENT = -125
```

7개의 numeric manipulation 함수를 이용해 사용자는 인수로 주어진 수의 종류와 관련된 모델 값에 접근할 수 있다. 7개 중 3개는 2개의 인수를 나머지 4개는 하나의 인수를 가진다.

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)

표 2.2 Numerical manipulation functions

다음은 위의 코드를 KISTI IBM 시스템에서 실행한 결과이다.

```
$a.out
EXPONENT = 4
FRACTION = 0.6374999881
NEAREST = 3.000000238
 SPACING = 0.2384185791E-06
```

다음은 Newton 방법에서 최적의 정밀도를 얻기 위해 SPACING 함수를 이용한 예이다.

```
.......
DO
DX = F(X)/DF(X)                ! Compute the next delta-X
X = X-DX
IF (DX<2*SPACING(X)) EXIT      ! Stop if near the spacing limits of
ENDDO                          ! that kind in this region
.........
```

제 3 장 Data Parallelism

Grand Challenge 문제와 같이 현재 계산과학 분야의 많은 문제들은 컴퓨팅 시스템의 보다 빠른 속도를 필요로 하고 있다. 이러한 요구에 대해 컴퓨팅 시스템의 성능 향상은 단일 프로세서의 속도를 증가시키는 방향이 아니라 프로세서를 추가로 장착해서 시스템의 계산 용량을 증가시키는 쪽으로 나아가고 있다. 이러한 까닭으로 앞으로는 계산량이 많은 응용 프로그램들이 필연적으로 병렬성을 많이 사용하게 될 것이다.

병렬성을 크게 '데이터 병렬성(data parallelism)'과 '프로세스 병렬성(process parallelism)'으로 나누어 볼 때, 응용 프로그램들은 이 두 가지 병렬성 중 하나 혹은 둘 모두를 가질 수 있다. 어떤 프로그램이 데이터 병렬성을 가진다는 것은 프로그램내에 여러 데이터에 유사한 계산을 동시에 수행할 수 있는 계산영역이 있다는 것을 말한다. 예를 들어 주어진 값으로 배열의 모든 원소를 나누어 주어야 하는 경우처럼 배열의 모든 원소에 동시에 같은 연산을 수행해야 하는 경우를 들 수 있다. 여기서 언급하고자 하는 데이터 병렬성은 특별히 배열원소에 대한 동시 연산을 의미한다.

프로세스 병렬성에서 프로세스는 임의의 일련의 계산을 의미하며 예를 들어, 하나의 서브루틴이 하나의 프로세스가 될 수 있다. 프로세스 병렬성을 가지는 프로그램은 서로 다른 프로세스들이 병렬로 수행될 수 있는 계산영역을 가지게 된다. 프로세스 병렬성을 이용함으로써 한 프로시저 호출 내에서 두 개의 서로 다른 인수값을 동시에 구하는 것이 가능해 진다.

프로그램의 병렬화 과정에서 프로세스 병렬성이 중요한 역할을 하지만, 특히 과학계산 프로그램에서는 배열과 연관된 계산이 워낙 많이 나타나므로 상대적으로 데이터 병렬성이 중요하게 다루어지고, 또 자주 사용된다. Fortran90은 데이터 병렬성을 지원하는 풍부한 연산들을 제공하고 있다. 이 장에서는 Fortran90이 제공하는 다양한 병렬 배열 연산들의 기능과 그 편리성에 대해 알아본다.

3.1 배열 연산

Fortran90에서는 반드시 스칼라로 제한되어 사용되는 소수의 연산을 제외하고는 어떤 연산도 배열을 피연산수로 가질 수 있으며 그 결과도 배열이 될 수 있다. Frotran77에서는 오직 스칼라 표현만 허용하지만, Fortran90에서는 거의 모든 표현에 배열을 직접 사용할 수 있도록 함으로써 데이터 병렬을 지원하고 있다. 다음의 두 표현을 비교해 보자. 물론 똑 같은 내용을 각각 Fortran77과 Fortran90으로 작성한 것이다.

Fortran77	Fortran90
REAL a(100), b(100) DO 10 i = 1, 100 a(i) = 2.0 10 b(i) = b(i)*a(i)	REAL, DIMENSION(100) :: a, b a=2.0 b=b*a

위에서와 같이 배열 연산을 위해 Fortran77의 스칼라 표현은 IF문의 제어 조건, DO 루프 인덱스 등과 같은 제어 구문을 필요로 하지만, Fortran90의 배열 연산은 제어 구문 없이 서로 대응되는 원소간의 직접적인 연산이 가능하다. 다음은 Fortran90 배열 기반 연산의 사용 예로서 사용된 변수는 모두 배열이다.

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

사용된 변수가 배열로 선언되었다는 것만 다를 뿐 스칼라 표현과 똑 같은 모습을 하고 있지만 위와 같은 배열 표현은 병렬 계산을 내포하고 있다. 만약 A, B, C, P, Q와 R이 2차원 배열이고 Z와 V가 1차원 배열이라면 위의 예는 다음과 같이 배열 연산으로 분명하게 표현될 수 있다.

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

사용자 함수에서도 배열을 피연산수로 가지는 배열 표현을 사용할 수 있으며 함수의 결과도 배열이 될 수 있다. 이에 관련된 내용은 뒤에서 다루게 될 것이다.

배열의 대응 원소끼리 연산을 수행하는 Fortran90의 데이터 병렬 배열 연산 표현은 Fortran77의 스칼라 표현식에 피연산수를 스칼라 대신 배열을 사용한 것으로 이는 기존의 스칼라 표현을 확장 시키고 일반화 시킨 것이라 할 수 있다.

3.1.1 정합성 요구

배열 연산에서의 정합성 요구는 연산에 참여하는 배열들의 모양이 서로 일치해야 함을 의미한다. 배열 연산에서 사용되는 피연산수 즉 배열은 서로 같은 차원을 가져야 하며 각 차원마다 원소의 개수가 같아야 한다는 것이다. 물론 연산의 결과역시 연산에 참여한 배열들과 같은 모양을 가지게 될 것이다.

2 x 3 배열 A와 B를 아래와 같이 정의했을 때,

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

A+B와 A x B의 계산 결과는 다음과 같다.

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

이와 같은 연산의 정의로 인해 연산에 참여하는 각 배열의 모양이 같아야 한다는 정합성 요구를 이해할 수 있다. 그러나, 한가지 예외 사항이 있는데 피연산수 중 하나로 스칼라가 올 때이다. 배열 B에 대해 B+2는 유효한 연산이며 글 결과는 다음과 같다.

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

다음과 같이 배열을 초기화할 목적으로 또는 배열을 스케일(scale) 하기위해 배열 연산에서 스칼라를 주로 사용한다.

```
A = 0          ! sets each element of A to zero
B = (B+1)/2    ! add 1 to each element of B then take half the result
```

Fortran90의 배열 연산에서 계산이 수행되는 배열은 반드시 우변의 계산이 모두 완료된 후 저장된다. 즉 우변에 있는 배열의 모든 원소가 병렬로 또는 주어진 순서대로 계산이 완료된 후 그 결과가 저장된다는 것이다. 다음 예는 이와 같은 규칙의 중요성을 보여준다.

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

배열 영역에 대해서는 다음 장에서 다루겠지만, 위의 예에서 배열 영역 G(P,:)는 행렬 G의 P번째 행을 나타내며, G(P,K)는 P번째 행, K번째 열의 성분을 나타낸다. 위의 계산은 행렬 G의 P번째 행 G(P,:)를 G(P,K)로 규격화 하는 것을 나타내며 이 계산의 결과로 G(P,K)는 최종적으로 1의 값을 가지게 될 것이다. 여기서 우변의 계산이 완료되기 전 G(P,K)가 1로 변경되어 저장된다면, G(P,K)를 이용한 P번째 행의 규격화는 제대로 수행되지 않게 된다는 사실에 주의하자. 이는 루프를 이용하는 순차 프로그램에서 흔하게 발생하는 오류이다. Fortran90의 배열 연산은 내부적으로 루프를 이용해 각 배열 원소에 대해 순차적으로 연산을 수행하는 것이 아니다. 배열 연산은 종합적으로 수행되는 병렬 계산으로 생각해야 한다.

3.1.2 배열 작성자

사용자는 배열 작성자(array constructor)를 이용해 1차원 배열을 명시적으로 구성할 수 있다. 만약 2차원 이상의 배열 작성을 원한다면 내부함수 RESHAPE을 이용하면 된다. 배열 작성자는 (/1, 2, 3, 4/)와 같이 (/…/)내에 콤마로 배열의 원소를 구분해 나열한 것이다. 일반적으로 배열 작성자의 각 원소는 어떤 스칼라 표현도 가능하다. 만약 작성자의 모든 원소가 상수라면 그 작성자는 배열 상수가 되는데 Fortran90에서 배열 작성자는 RESHAPE 함수와 같이 사용되어 배열 상수를 표현하는 수단이 된다.

암시적 DO 구문

스칼라 표현과 더불어 배열 작성자의 원소를 나타내는 방법으로 암시적 DO 구문과 배열 표현이 있다. 우선 암시적 DO 구문은 다음과 같은 형식을 가진다.

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

표현식 (/(k, k=1,n)/)는 원소가 1,2,3,…,n으로 주어지는 1차원 벡터를 생성하며 n=4라면 (/1,2,3,4/)가 된다. 또 하나의 예로 1과 0이 반복되는 100만개의 원소로 구성되는 벡터는 다음과 같이 표현할 수 있다.

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

다음은 암시적 DO 구문을 대응되는 스칼라 표현으로 나타낸 예이다.

```
! 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

배열 표현은 임의 차원의 배열을 이용해 배열 작성자를 구성하는 것이다. 배열 A가 1000 x 1000의 2차원 배열일 때 (/A+1.3/)은 100만개의 원소를 가지는 배열 작성자가 된다. 각 원소는 배열 A의 각 원소에 1.3을 더한 값이 되며, 1열 1000개, 2열 1000개, … 와 같이 열-우선 순으로 나열돼 하나의 배열 작성자를 구성한다.

```
(/A+1.3/) = (/ ( ( A(j,k)+1.3, j=1,1000 ), k=1,1000 ) /)
```

여기서 A+1.3의 원소를 열-우선 순으로 나열하지 않고 행-우선 순으로 나열된 배열 작성자를 원한다면 다음과 같이 암시적 DO구문을 이용한다.

```
(/ ( ( A(j,k)+1.3, k=1,1000 ), j=1,1000 ) /)
```

RESHAPE 함수

1차원으로 정의되는 배열 작성자를 이용해 원하는 모양의 배열을 구성하기 위해 Fortran90의 내부함수 RESHAPE를 사용한다. RESHAPE 함수는 다음과 같은 형태로 사용한다.

```
RESHAPE (array-constructor, shape-vector)
```

여기서 shape-vector는 원하는 배열 모양에 대한 각 차원의 크기를 지정하는 값을 (/…/)로 묶어 나타낸다.

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

다음은 1000 x 1000크기의 실수타입 단위 행렬을 배열 작성자와 RESHAPE 함수를 이용해 Ident_1000이라는 이름을 가지는 배열 상수로 정의한 것이다.

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

3.1.3 마스크된 배열 할당 - WHERE 문

마스크된 배열 연산은 배열의 각 원소에 병렬로 수행되는 연산이 일부 원소에만 적용되도록 logical 타입의 마스크를 사용하는 것이다. 마스크를 통하여 배열에 값을 할당하기 위해 WHERE 문을 사용한다.

```
WHERE (mask) array-assignment-statement
```

WHERE 구문의 마스크 원소가 .TRUE. 일 때 이에 대응되는 배열 원소에 값이 할당되며, .FALSE. 가 되면 값이 할당되지 않는다. 다음은 마스크를 이용해 배열의 일부 원소에 값을 할당하는 예이다.

```
WHERE (C .NE. 0) A = B/C
```

여기서 배열 A, B, C는 같은 모양이어야 하며, 이때 논리식 C .NE. 0 가 나타내는 마스크 배열도 A, B, C와 똑같은 모양이 된다. 만약 배열 B와 C가 다음과 같이 주어졌다면,

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

위의 WHERE 문에서 마스크된 배열 A의 원소는 다음과 같이 할당된다.

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

여러 개의 할당에 하나의 마스크를 적용할 수 있으며 이때 WHERE 문은 다음과 같이 블록화 된다.

```
WHERE (mask)
array-assignment-1
array-assignment-2
.........
END WHERE
```

또한 블록화된 WHERE 문에 ELSE WHERE 옵션을 덧붙여 사용하면 마스크 원소가 .FALSE. 일 때도 배열에 값을 할당하도록 할 수 있다.

```
WHERE (mask)
```

```
array-assignment-1
array-assignment-2
........
ELSE WHERE
array-assignment-n+1
........
END WHERE
```

```
WHERE (C .NE. 0)
A = B/C
ELSE WHERE
A = B
END WHERE
```

앞서 주어진 배열 A, B, C에 대해 위와 같은 연산을 행하면 배열 A는 다음과 같이 할당된다.

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

3.1.4 Assumed-Shape Dummy Arguments

Fortran77에서 프로시저의 인수로 배열을 패싱시켜 사용할 때, 프로시저내에서 정의되는 dummy 배열은 그 모양이 명시적으로 선언 되어야 했다. 이를 위해 Fortran77에서는 배열의 각 차원의 크기를 프로시저내에서 선언해 사용하거나 호출 프로그램에서 프로시저의 인수로 필요한 값을 패싱 시켜야 했었지만, Fortran90에서는 assumed-shape dummy arguments를 이용해 dummy 배열에 대한 정보를 명시적으로 선언하지 않고 프로시저내에서 사용할 수 있게 되었다. Fortran 프로시저에서 dummy 배열은 explicit-shape(F77, F90), assumed-size(F77, F90), 그리고 assumed-shape(only F90)의 세 가지 형식으로 사용될 수 있다.

Explicit-Shape

dummy 배열의 모양과 크기가 명시적으로 결정된 값을 가진다. 모양과 크기가 고정된 배열에만 사용할 수 있으므로 프로시저의 쓰임새가 한정적이다.

```
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 인덱스의 마지막 값을 명시적으로 정하지 않고 사용하는 방법이다. 마지막 인덱스를 제외한 나머지 인수는 explicit-shape와 마찬가지로 명시적으로 선언해 주어야 한다. assumed-size 방식은 거의 사용되지 않아 Fortran에서 사라져 가는 기능 중 하나이다.

```
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

dummy 배열의 모양과 크기를 결정하는 인수를 명시적으로 선언하지 않고 호출될 때 마다 대응되는 실제 배열의 정보를 내부적으로 전달 받아 사용한다. 이때 dummy 배열과 대응되는 실제 배열은 차원과 형식이 서로 일치해야 한다. assumed-shape dummy arguments는 각 차원마다 콜론으로 선언된다.

```
SUBROUTINE S3(A, B, C, k)
REAL:: A(100, 100)                    !static
REAL:: B(:,:)                         !assumed-shape
REAL:: C(-10:20, k:)                  !assumed-shape
.........
```

아래의 예에서 U는 2차원 assumed-shape 배열이고 V는 1차원 assumed-shape 배열이다. 서브루틴 CALC3을 호출할 때 실수 타입의 2차원 배열은 무엇이든 U로 패싱될 수 있다. 마찬가지로 실수 타입의 1차원 배열은 모두 V로 패싱될 수 있다.

```
SUBROUTINE CALC3(t, U, V)
REAL:: t, U(:,:), V(:)
```

```
.....
END SUBROUTINE CALC3
```

만약 assumed-shape dummy arguments를 사용하는 프로시저가 호출 프로그램 밖에 위치한 외부 프로시저[1] 라면 호출 프로그램은 이 외부 프로시저에 대해 명시적으로 정의한 인터페이스 블록[2] 을 가져야 한다.

```
... ! 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
```

1. Fortran90 의 프로시저는 외부 (external), 내부 (internal), 모듈 (module) 세 종류
2. 인터페이스 블록에 대해서는 5 장에서 다룰 것이다 .

제 4 장 Array Sections

앞선 3 장에서 데이터 병렬성을 지원하는 Fortran90 의 배열 연산에 대해 알아보았다 . 4 장에서 다루고자 하는 배열부분 (array section) 도 Fortran90 의 배열 처리 특성을 설명하는 것이고 데이터 병렬 처리의 연장선상에 있다는 것을 밝혀둔다 .

4.1 배열부분

앞서 나왔던 배열 연산 G(P,:)=G(P,:)/G(P,K) 은 2 차원 배열 G 에서 P 번째 행에만 연산을 행하는 것이다 . 이와 같이 배열 연산이 배열 전체에 대해서가 아니라 배열의 일부분에만 적용되는 경우가 자주 있다 . 배열 부분은 하나 이상의 원소를 포함하고 있는 배열의 일부분을 의미하는데 Fortran90 에서 이러한 배열 부분을 표현하는 방식은 세가지 (simple subscript, subscript triplet, vector subscript) 가 있다 .

Simple Subscript

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

배열의 원소 하나를 나타내는 방식이다. 스칼라 subscript를 이용해, 나타내고자 하는 원소의 위치를 표현한다. subscript의 개수는 배열의 차원과 같다.

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

Triplet Subscript

```
array-name (start index : end index : stride)
```

Triplet subscript 방식은 배열 부분을 세 개의 정수 start index, end index, stride 로 표현한다 . stride 가 1 이면 stride 는 생략 가능하다 . start index 또는 end index 도 생략될 수 있으며, 만약 생략 되었다면 대응되는 차원의 하한(lower bound) 이 start index 로 상한 (upper bound) 이 end index 로 가정된다 . 다음은 주어진 배열 Q 에 대해 몇 가지 배열 부분들을 triplet subscript 로 표현한 것이다 .

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

$Q(1:5,2) = \begin{pmatrix} 11 \\ 3 \\ 45 \\ 20 \\ 56 \end{pmatrix} = Q(:,2)$	$Q(1:2,5:6) = \begin{pmatrix} 1 & 9 \\ 52 & 27 \end{pmatrix} = Q(:2,5:)$
$Q(5,4:6) = \begin{pmatrix} 66 & 77 & 90 \end{pmatrix} = Q(5,4:)$	$Q(1:5:2,2) = \begin{pmatrix} 11 \\ 45 \\ 56 \end{pmatrix} = Q(::2,2)$

다음 예는 Fortran90 에서의 배열정의가 Fortran77 과 비교해 얼마나 편리한가 보여주고 있다 .

Fortran77	Fortran90
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	REAL, DIMENSION(10,10) :: A,B A(1:8,1:8) = B(2:9,2:9)

```
Fortran90
REAL, DIMENSION(1024) :: a
        a(1:1024) = 1.0       ! same as a = 1.0
        a(:1024:2) = 2.0      ! a(1)=a(3)=...= 2.0
```

Vector Subscript

Vector subscript 는 배열의 subscript 값을 (/…/) 로 묶어놓은 1 차원 배열이다 . vector subscript 를 구성하는 원소는 전체 배열 subscript 의 범위 내에서 순서에 무관하게 사용될 수 있다 .

```
Fortran90
REAL, DIMENSION :: ra(6), rb(3)
INTEGER, DIMENSION (3) :: iv
iv = (/ 1, 3, 3 /)                  ! 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 /)
```

다음은 triplet subscript에서 예시한 배열 Q의 배열 부분을 vector subscript로 나타낸 것이다.

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 는 또한 다음과 같이 implied DO 루프 형식으로도 표현할 수 있다 .

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)

Fortran90
INTEGER, DIMENSION(1024) :: a a = (/ (i, i=1,1024) /) ! a(1)=1, a(2)=2, … a = (/ (i, i=1,4096,4) /) ! a(1)=1, a(5)=5, …

subscript 는 전체 배열의 크기 범위 내에서 중복 사용될 수 있다 . 이러한 중복 사용이 가능하므로 vector subscript 는 전체 배열의 크기보다 더 많은 원소를 가질 수 있다 . 따라서 다음과 같이 차원의 크기가 주어진 전체 배열보다 더 큰 배열을 정의할 수 있다 .

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

4.2 동적배열

Fortran77 에서 배열에 대한 메모리 할당은 컴파일 동안에 그 크기와 주소가 결정되는 정적할당 방식이다 . 이에 반해 Fortran90 이 지원하는 동적배열은 프로그램 실행 중에 메모리가 할당되며 배열의 크기를 프로그램에서 계산되는 값 또는 입력값으로 결정할 수 있다 . Fortran90 에서는 automatic array, allocatable array, 그리고 pointer array 이렇게 세 가지 형태의 동적배열을 지원한다 .

4.2.1 Automatic Arrays

Automatic array 는 크기가 dummy 인수에 의해 결정되는 지역 배열이다 . 프로시저 내에서 자동 생성되고 프로시저의 종료와 함께 자동으로 소멸된다 . 크기가 프로시

저의 dummy 인수에 의해 결정되므로 프로시저가 호출될 때 마다 그 크기는 달라질 수 있다 .

다음 예에서 배열 work1 과 work2 는 automatic array 이다 .

```
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
```

다음 예에서 사용되고 있는 함수 SIZE(A, n) 은 배열 A 의 n 번째 차원의 크기를 리턴 해주는 Fortran90 의 inquiry 함수 이다 .

```
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
```

4.2.2 Allocatable Arrays

Fortran90 에서는 ALLOCATE 문을 이용해 메모리를 동적으로 할당할 수 있다 . Allocatable array 는 다음과 같이 ALLOCATABLE attribute 로 선언된다 .

```
PROGRAM simulate
IMPLICIT NONE
INTEGER :: n
INTEGER, DIMENSION(:,:), ALLOCATABLE :: a    ! 2D
```

배열의 이름과 차원은 위와 같이 미리 결정되어 있어야 하지만 그 크기는 다음과 같이 입력 값으로부터 받아오거나 프로그램 실행 중에 계산되는 값에 의해 결정된다 .

```
PRINT *,'Enter n:'
READ *,n
IF(.NOT.ALLOCATED(a)) ALLOCATE( a(n,2*n) )
.....
DEALLOCATE(a)
```

Allocatable array 는 프로시저 내에서 지역적으로 사용될 수도 있고 프로그램 전체에서 global 하게 사용될 수도 있다 . allocatable array 를 더 이상 사용할 필요가 없다면 DEALLOCATE 문을 이용해 해제 시켜야 한다 . 그렇지만 , 프로시저 내에서 지역적으로 사용된 allocatable array 는 SAVE 로 명시해 두지 않은 경우 프로시저의 종료와 함께 자동으로 소멸된다 .

다음과 같이 내부함수 STAT 을 ALLOCATE 또는 DEALLOCATE 문에서 사용하면 할당 상태를 확인할 수 있다 . 여기서 status 값이 0 이면 할당 또는 할당된 배열의 해제가 성공적으로 이루어진 것이고 0 이 아니면 실패한 것이다 .

```
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"
```

4.2.3 Pointer Arrays

pointer array 는 ALLOCATE 문을 이용해 명시적으로 할당되고 실행 중에 결정되는 크기를 가지며 DEALLOCATE 문을 이용해 명시적으로 할당 해제 된다는 점에서

allocatable array 와 유사하다 . pointer array 는 target 으로 명시적으로 선언된 다른 배열과 배열 부분에 대한 aliasing 을 위해 사용된다 .

```
REAL, TARGET :: B(100,100)        ! 배열 B 는 target 속성을 가진다 .
REAL, POINTER :: U(:,:),V(:),W(:,:)   ! 3 개의 포인터 배열 선언
 ...
U => B(I:I+2,J:J+2)               ! U 는 B 의 3X3 부분을 point
ALLOCATE ( W(M,N) )               ! 크기가 MXN 인 W 를 동적할당
V => B(:,J)                       ! V 는 B 의 J 번째 열을 point
V => W(I-1,1:N:2)                 ! V 는 W 의 I-1 번째 열의 일부를 point 하도록 바꿈
```

- allocatable 로 선언하는 대신 pointer 로 선언
- 다른 배열 또는 배열 부분으로의 aliasing(point to) 을 위해 사용
- allocatable array 의 모든 기능을 포함하지만 상대적으로 복잡하여 효율면에서 좋지않음

지역적으로 계산되는 값에 의존하는 크기를 가지는 동적배열의 정의를 위해 일반적으로 allocatable array 를 많이 사용한다 . 그러나 알고리즘에서 동적 aliasing 이 필요한 경우에 pointer array 를 사용하게 된다 . 관련하여 보다 자세한 내용은 5 장에서 포인터와 같이 다룰 것이다 .

4.2.4 Array-Valued 함수

Fortran90 함수들은 결과를 배열로 리턴할 수 있다 . 다수의 내부함수들은 항상 배열로 결과를 리턴하며 또한 대다수의 내부함수들은 배열을 결과로 리턴할 수 있다 . 또한 아래와 같이 사용자가 정의하는 함수도 배열을 결과로 가질 수 있다 . 함수들이 배열을 연산에 사용할 수 있고 결과로 리턴할 수 있으므로 해서 데이터 병렬 계산이 가능해지며 이로 인해 사용자는 데이터 병렬성과 프로세스 병렬성을 자연스럽게 같이 사용할 수 있게 된다 .

```
FUNCTION add_vec (a, b, n)
IMPLICIT NONE
INTEGER, INTENT(IN) :: n
REAL, DIMENSION (n), INTENT(IN) :: a, b
REAL, DIMENSION (n) :: add_vec
INTEGER :: i
DO i = 1, n
add_vec(i) = a(i) + b(i)
END DO
END FUNCTION add_vec
```

Transformational 함수

Fortran90 에서의 array-valued 함수는 크게 "transformational 함수 " 와 "elemental 함수 " 로 나누어 볼 수 있다 . transformational 함수는 배열을 입력으로 받아 그것을 다른 모양의 배열이나 스칼라로 'transform' 하여 출력으로 내어 놓는다 . 스칼라를 입력으로 받아 배열을 결과로 내어놓을 수도 있다 . Fortran90 에서 transformational 함수에 해당하는 내부함수는 다음 표에 정리해 둔 것과 같이 모두 42 개가 있다 .

표 4.1 Transformational intrinsic functions

Transformational Intrinsic Function	Comment
environmental inquiry function (9)	2장. 5절의 표2 참조
array function (21)	표 2. 참조
ASSOCIATED	check association of pointer
BIT_SIZE	number of bits of integer
DOT_PRODUCT	mathematical dot product of two vectors
KIND	2장. 1절 참조
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	2장. 2절 참조
SELECTED_REAL_KIND	2장. 2절 참조
TRIM	remove trailing blanks from a string
TRANSFER	transfer bit pattern to a different type

Fortran90 에는 21 개의 내부 배열 함수가 있다 . 이들 배열 함수들은 배열을 작성하거나 주어진 배열에 관한 정보를 알아내기 위해 사용하며 , 모두 transformational 함수이다 .

표 4.2 array intrinsic functions

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

- SIZE(array [,dim]) 함수의 사용 예

```
SUBROUTINE swap(a,b)
IMPLICIT NONE
REAL, DIMENSION(:) :: a,b
REAL, DIMENSION(SIZE(a)) :: work
work = a
a = b
b = work
```

```
END
```

- 모든 원소를 포함하는 Global 연산의 사용 예

```
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 reduction, ANY 와 ALL 의 사용 예

```
LOGICAL a(n)
REAL, DIMENSION(n) :: b, c
IF ( ALL(a) )       ...            ! global AND
IF ( ALL(b == c) ) ...             ! true if all elts equal
IF ( ANY(a) ) ...                  ! global OR
IF ( ANY(b < 0.0) ) ...            ! true if any elt < 0.0
```

Elemental 함수

Elemental 내부함수는 대부분 스칼라 dummy 인수로 정의되는 것들이다. 이런 함수들은 배열이 실제 인수로 주어지면 , 그 배열과 같은 모양의 배열을 결과로 내어 놓는다 . 결과로 나온 배열의 원소들은 함수의 실제 인수로 들어간 배열의 원소 하나 하나에 함수가 각각 적용된 결과이다 . 따라서 elemental 함수는 실제 인수의 각 원소에 병렬계산의 개념으로 작용하게 된다 . Fortran 의 계산 함수들은 모두 elemental 함수 이다 . 예를 들어 내부함수 COS(X) 를 보면 X 가 스칼라 이면 결과가 스칼라로 X 가 배열이면 그 결과도 X 와 모양이 같은 배열로 나온다 . Fotran90 에는 위에 정리된 42 개 transformational 함수를 제외하고 모두 108 개의 elemental 내부함수가 있다 .

사용자 정의 Array-valued 함수

배열을 결과로 가지는 함수를 사용자가 직접 만들어 사용할 수 있으며 , 이런 함수는 모두 transformational 함수이다 . 결과로 주어지는 배열의 모양은 인수로 주어지

는 배열의 특성과 같은 인수에 의해 동적으로 결정되어 진다 . automatic array 를 유용하게 사용할 수 있지만 , 명시적인 모양을 가지는 배열이나 allocatable array, pointer array 등 어떤 방식의 배열이든 함수의 결과로 사용할 수 있다 .

다음은 배열을 결과로 가지는 함수를 정의한 예이다 . 함수 Partial_sums 는 일차원 배열을 입력으로 받아들여 첫 원소부터 n 번째 원소까지의 합을 n 번째 원소로 가지는 새로운 일차원 배열을 결과로 내어 놓는다 . 입력으로 들어가는 배열과 결과로 나오는 배열은 그 크기가 동일하다 .

```
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 5 장 기 타

자유형식 소스코드 등 코딩의 편의성, 배열 기반의 연산, 동적 메모리 할당, 사용자 정의 데이터 타입과 연산자, 모듈, SELECT CASE, CYCLE 과 EXIT 등의 새로운 제어 구조, Recursive 함수, 프로시저에 대한 generic 명명 등이 Fortran 이 77 에서 90 으로 오며 새로워진 대표적인 기능들이다. 앞선 두 번의 기술서 에서는 numerical robustness 와 데이터 병렬성에 초점을 맞추어 Fortran90 의 새로운 기능에 대해 소개하였다. 이제 과학계산을 위한 Fortran90 의 기능을 소개하는 마지막 내용으로 본 기술서 에서는 Fortran90 의 중요한 특징이지만 그 동안 다루지 못했던 모듈, 인터페이스 블록, 포인터 등의 객체 지향적 요소들과 기타 몇 가지 중요한 기능들에 대해 소개할 것이다.

5.1 Interface Block

Fortran90 의 주 프로그램 단위는 메인 프로그램과 모듈이다. 메인 프로그램은 프로그램 실행이 시작되고 종료 되는 곳으로 다수의 프로시저를 포함할 수 있다. 모듈은 프로시저와 선언을 포함할 수 있는 프로그램 단위로 C++ 의 클래스와 유사하며 Fortran77 의 COMMON, INCLUDE, BLOCK DATA 와 같은 불안정한 객체 지향적 요소들을 대신할 수 있다. 모듈에 대해서는 다음 장에서 다룰 것이다.

5.1.1 프로시저와 인터페이스

서브루틴과 함수를 말하는 프로시저는 메인 프로그램에 포함돼 있는 내부 프로시저와 메인 프로그램이나 모듈에 포함되지 않고 독립적으로 존재하는 외부 프로시저[1], 그리고 모듈 프로시저가 있다.

메인 프로그램과 외부 프로시저는 서로 분리된 독립적인 프로그램 단위이다. 메인 프로그램과 프로시저는 다른 프로시저에서 지역적으로 선언된 아이템들에 접근할 수 없으며 마찬가지로 외부 프로시저는 메인 프로그램에 선언된 아이템에 접근할 수 없다. 한 프로그램 단위에서 다른 단위로 정보를 전하기 위해서 인수와 함수 이름을 사용할 수 있을 뿐이다.

1. Fortran77 에는 외부 프로시저만 있다 .

내부 프로시저는 하나의 프로그램 단위에 포함돼 있으므로 프로시저에 대한 참조에 대해서 인수의 개수와 타입 등의 속성들과 함수의 리턴 값 등이 바르게 사용되었는지를 컴파일러가 점검할 수 있다. 비슷하게 모듈에 포함되는 모듈 프로시저 역시 모듈내의 문장으로 참조되거나 모듈에 대한 USE문을 사용함으로써 컴파일러가 인수와 결과 등이 올바른지 점검할 수 있도록 한다. 이와 같이 모듈 또는 내부 프로시저는 기본적으로 '명시적 인터페이스(explicit interface)'를 가져 인수의 사용과 리턴 값 등에 대해 컴파일러가 오류를 점검할 수 있다. 그러나 외부 프로시저는 메인 프로그램이나 다른 프로그램 단위와 분리되어 독립적으로 존재하므로 컴파일러가 프로시저에 대한 참조의 오류 여부를 점검할 수 없다. 외부 프로시저는 기본적으로 '암시적 인터페이스(implicit interface)'를 가진다.

5.1.2 외부 프로시저와 인터페이스 블록

Fortran90에서는 '인터페이스 블록(interface block)'을 이용하여 외부 프로시저에 대해서도 명시적 인터페이스를 제공해 주도록 하고 있다. 인터페이스 블록은 컴파일러에게 프로시저 인수의 모양, 형태 등의 속성을 알려 주어 프로시저에 대한 참조의 오류 여부를 점검할 수 있도록 해준다.

인터페이스 블록의 일반적인 형태는 다음과 같다.

```
INTERFACE
interface_body
END INTERFACE
```

interface_body에 해당하는 부분에는 다음과 같은 것들이 들어갈 수 있다.

- FUNCTION/SUBROUTINE header
- Dummy argument declarations
- Local declarations
- END FUNCTION/END SUBROUTINE statements

인터페이스 블록 사용 예 1

다음은 외부 프로시저 squash를 호출해 사용하는 메인 프로그램 stress에서 인터페이스 블록을 이용해 명시적 인터페이스를 설정해 준 것이다.

```
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)
.....
```

이와 같이 외부 프로시저에 대한 호출이 있는 경우 항상 인터페이스 블록을 사용하는 것이 좋다.

인터페이스 블록 사용 예 2 : No Interface(External Procedure: Implicit Interface)

메인 프로그램에서 외부 프로시저에 대한 명시적 인터페이스를 지정하지 않으면 이 외부 프로시저는 기본적으로 암시적 인터페이스를 가지게 된다.

```
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
```

위의 예는 메인 프로그램과 외부 프로시저의 인수 타입이 일치되지 않아 오류가 있는 프로그램이다. 암시적 인터페이스를 가지는 외부 프로시저에 대해 컴파일러는 별 문제점을 찾지 못하고 컴파일을 완료하지만 프로그램의 실행결과는 엉터리 값이 나올 것이다.

IBM XL Fortran 컴파일러에서 -qextchk 옵션을 주고 컴파일을 수행하면 다음과 같은 링크 에러가 발생한다.

```
$ 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 사용 예 3 : With Interface(External Procedure: Explicit Interface)

이번에는 위의 예제 2의 코드에 다음과 같이 외부 프로시저에 대한 명시적 인터페이스를 넣어주었다.

```
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
```

다음과 같이 컴파일러는 명시적 인터페이스를 설정해준 위의 프로그램에 대해서 -qextchk 옵션 없이 오류를 찾아내 준다.

```
$ 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.
```

5.1.3 내부 프로시저

Fortran90에서는 프로그램 단위 내에 함수나 서브루틴 같은 프로시저가 올 수 있으며 이를 내부 프로시저라 한다. 모든 내부 프로시저는 그것을 포함하는 프로그램 단위[1] 의 맨 마지막 부분에서 CONTAINS문 다음에 위치해야 한다. 외부 프로시저와 그 형태는 동일하며 단지 END문 다음에 FUNCTION 또는 SUBROUTINE을 반드시 붙여 주어야 한다는 점이 다르다. 내부 프로시저를 포함하는 프로그램 단위에서 선언된 변수는 내부 프로시저에서 접근할 수 있으며 내부 프로시저에서 다시 정의할 수도 있다.

내부 프로시저를 이용한 프로그램 (Explicit Interface)

다음은 위의 예제 3의 코드를 인터페이스 블록을 사용하지 않고 내부 프로시저를 이용하도록 수정한 것이다.

```
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
```

내부 프로시저는 명시적 인터페이스가 기본이므로 위 예제의 오류는 다음과 같이 컴파일러에 의해 발견된다.

```
$ 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.
```

1. 내부 프로시저를 포함하는 프로그램 단위를 그 내부 프로시저의 ' 호스트 (host)' 라 한다 .

5.1.4 INTENT Attribute

Fortran90에서는 INTENT 속성(attribute)을 이용해 프로시저내의 더미 인수들에 대한 계획된 이용(in, out, 또는 inout)을 컴파일러에게 알려줌으로써 보다 효율적인 컴파일과 이로 인한 성능향상을 얻을 수 있도록 하고 있다.

- 프로시저에 들어와서 나갈 때까지 값이 변함없는 인수: INTENT(in)
- 프로시저 내에서 값을 할당 받을 때까지 사용되지 않는 인수: INTENT(out)
- 프로시저에 들어와 사용되고 값을 새로이 할당 받아 그 결과를 호출한 프로그램에 되돌려 주는 인수: INTENT(inout)

INTENT 속성 사용 예

```
SUBROUTINE neuron(a, b, c, m, n)
REAL, DIMENSION(n, n), INTENT(IN) :: a
REAL, DIMENSION(m, m) :: b, c
INTENT(out) :: b
INTENT(inout) :: c
c = SQRT(c)
b = c + SUM(a)
END
```

- INTENT(in) 인수 a는 프로시저 내에서 값이 변하지 않는다.
- INTENT(out) 인수 b의 값은 프로시저 내에서 값을 할당 받을 때까지 사용되지 않는다.
- INTENT(inout) 인수 c는 프로시저 내에서 사용되고 값이 새로 할당된다.

Fortran90 프로그램에서 INTENT 속성을 반드시 사용해야 하는 것은 아니다. 그렇지만 INTENT 속성을 사용함으로써 컴파일러는 코딩 오류를 점검하고 따라서 프로그램의 안정성을 높일 수 있다. 만약 INTENT(in) 속성의 인수가 값을 할당 받거나 INTENT(out) 속성의 인수가 새로운 값을 할당 받지 않는 다면 컴파일 단계에서 에러가 발생하게 될 것이다.

```
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 속성을 잘못 지정해준 위의 코드를 컴파일 하면 다음과 같이 컴파일 에러가 발생한다.

```
$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.
```

5.2 Derived Type

Fortran90에서는 C의 구조체와 같이 사용자가 새로운 데이터 타입을 정의해 사용할 수 있는 기능을 제공한다. 새로운 데이터 타입은 TYPE/END TYPE을 이용해 다음과 같은 형태로 정의된다.

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

예를 들면 3차원 좌표를 나타내는 새로운 데이터 타입 COORDS_3D를 다음과 같이 정의할 수 있다.

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

이와 같이 정의된 타입 COORDS_3D는 세 개의 실수 항 x, y, z를 가지고 있으며, 새롭게 정의된 COORDS_3D 타입의 객체를 다음과 같이 선언해 사용한다.

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

유도 타입 객체들은 기본 타입 선언과 마찬가지로 DIMENSION, ALLOCATABLE, TARGET[1] 등의 다양한 속성을 가질 수 있으며, 인수로 사용될 수 있고 INTENT, OPTIONAL 등의 속성도 가질 수 있다. 그러나 PARAMETER 속성을 가질 수는 없다.

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

5.2.1 Supertypes

새로운 유도 타입을 정의하는데 이미 정의된 유도 타입을 이용할 수 있다. 이미 정의돼 있는 유도 타입을 사용할 수 있을 뿐 아니라 현재 정의하는 타입 자체를 내부에서 다시 사용할 수도 있다(이로 인해 recursive 자료 구조 형성이 가능하다.).

중심점과 반지름에 의해 정의되는 구를 나타내는 새로운 유도 타입을 다음과 같이 정의할 수 있다.

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

5.2.2 Derived Type Assignment

유도 타입에 값을 할당하는 방법은 두 가지가 있다.

- Component by component
- As an object

유도 타입의 특정 항을 나타내기 위해 연산자 '%'를 사용한다. 위에서 정의된 SPHERE 타입을 가지는 객체 bubble을 가정하자. 이때 객체 bubble을 이루는 항은 일차적으로 center와 radius 두 개이고 center는 다시 세 개의 항을 가지게 된다.

---

1. TARGET 속성에 관해서는 포인터에서 다룰 것이다.

```
TYPE (SPHERE) :: bubble
```

다음과 같이 객체 bubble에 각 항 별로 값을 할당할 수 있다. 반지름이 3.0이고 중심의 좌표가 (1.0, 2.0, 3.0)인 구를 나타낸다.

```
bubble%radius = 3.0
bubble%center%x = 1.0
bubble%center%y = 2.0
bubble%center%z = 3.0
```

유도 타입을 가지는 객체에 값을 할당하기 위해 유도 타입 작성자(constructor)를 사용할 수도 있다. 작성자는 다음과 같이 유도 타입 이름과 이어지는 괄호에 값을 나열한 형태를 가진다.

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

객체 bubble을 SPHERE 작성자를 이용해 다음과 같이 설정할 수도 있다. SPHERE 작성자는 COORDS_3D 타입과 REAL 타입의 두 개의 항을 가진다.

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

bubble%center 자리에는 COORDS_3D(1.0, 2.0, 3.0)를 직접 넣어 줄 수도 있다.

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

그러나 다음과 같이 정의하는 것은 불가능하다.

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

같은 유도 타입을 가지는 두 개의 객체 bubble과 ball이 있어 ball에 bubble과 같은 값을 할당하기 위해서는 다음과 같이 쓸 수 있다.

```
ball = bubble
```

5.3 Module

모듈은 타입선언, 서브프로그램, 새로운 데이터 타입선언 등을 하나로 묶어 정의할 수 있는 프로그램 단위로서 Fortran90에 새롭게 추가된 기능이다. 사용자는 모듈을 사용해서 보다 신뢰성 있고, 재사용이 쉬우며 작성하기도 쉬운 객체 기반의 코드를 작성할 수 있다.

모든 프로그램 단위는 USE문을 이용해 모듈을 첨부할 수 있고 그렇게 해서 그 모듈이 제공하는 기능을 이용할 수 있다. 모듈은 다음과 같은 것을 포함할 수 있다.

Global Object Declaration

모듈은 Fortran77의 COMMON과 INCLUDE문을 대신해 사용된다. 글로벌 데이터 설정이 필요한 경우 인수를 패싱하지 않고 하나의 모듈에 그 데이터를 정의해 둔다. 이렇게 모듈에 정의된 데이터는 그 모듈을 첨부하는 어떤 프로그램 단위에서도 사용될 수 있으며 그 값은 각각의 모듈 첨부에 대해 일정하게 유지된다.

Interface Declaration

인터페이스에 대한 정의도 모듈에 해두고 명시적 인터페이스가 필요할 때마다 모듈을 첨부해 사용할 수 있다.

Procedure Declaration

프로시저를 모듈에 정의해서 그 모듈을 첨부한 어떤 프로그램 단위에서도 사용할 수 있도록 한다. 이렇게 하면 프로시저와 그 인터페이스가 모듈 내에서 명시적이므로 인터페이스 블록을 다시 작성할 필요가 없어진다.

Controlled Object Declaration

접근 구문을 이용해 모듈에서 선언된 변수, 프로시저, 연산자 등의 노출 정도(visibility)를 제어할 수 있다.

Packaged of Whole Sets of Facilities

사용자 정의 타입, 프로시저, 연산자, generic 인터페이스 등을 정의하고 하나로 묶을 수 있어 간단한 객체 지향성을 제공한다.

Semantic Extension

Semantic extension 모듈은 사용자 정의 타입 선언, 접근 루틴, overloading 연산자와 고유함수 등의 집합으로 프로그램 단위에 첨부되어 사용자가 마치 Fortran90 의 한 부분인 것처럼 사용할 수 있도록 하는 기능을 제공한다.

5.3.1 General Form

모듈 프로그램 단위의 기본 모양은 다음과 같다.

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

<*module name*> 은 USE 문에 사용되는 이름이고 파일명과 같을 필요는 없다.

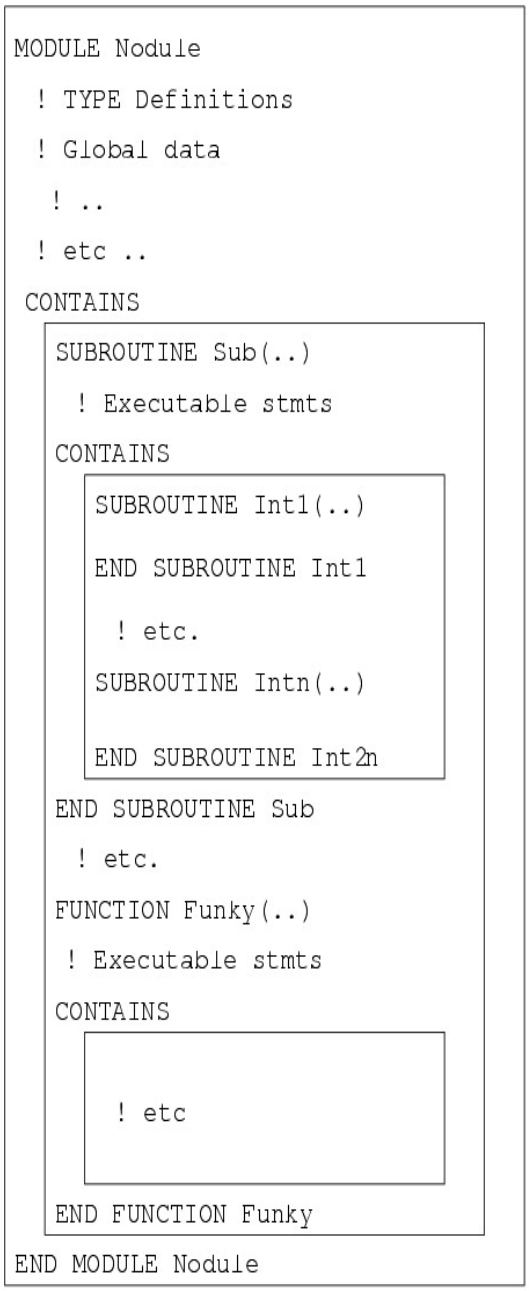


그림 5.1 Schematic Diagram of a Module Program Unit(The Liverpool Fortran90 Courses)

모듈 안에서 USE 문을 이용해 다른 모듈을 첨부할 수 있으며 이를 use-association 이라 한다.

5.3.2 Module: Global Data

Fortran77 의 COMMON 문을 대체하는 기능으로 글로벌 변수를 모듈에 정의해 두면 이를 필요로 하는 프로그램 단위는 USE 문을 이용해 모듈을 첨부해서 글로벌 변수에 접근할 수 있다.

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

위와 같이 정의된 모듈 globals 를 첨부시키기 위해 프로그램 단위는 다음과 같이 USE 문을 사용할 수 있다.

```
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
```

USE 문은 프로그램 단위에서 제일 위에 위치해야 한다.

5.3.3 Module: Procedure

모듈은 다른 프로그램 단위가 접근할 수 있는 프로시저를 포함할 수 있다. 모듈 내에 위치하는 모듈 프로시저도 내부 프로시저와 같이 다음과 같은 점을 제외하고는 외부 프로시저와 동일한 모양을 가진다.

- CONTAINS 문 다음에 위치한다.
- END 문 다음에 FUNCTION 또는 SUBROUTINE 이 있어야 한다.

다음 예와 같이 유도 타입과 이와 관련된 함수들을 하나로 묶어 관리하는데 유용하게 사용할 수 있다.

```
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
```

5.3.4 Module: Generic Procedure

Fortran90에서는 프로시저에 대한 generic 인터페이스를 정의해서 유사한 기능을 하는 프로시저들을 묶어 하나의 이름으로 사용할 수 있다.

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

다음은 입력으로 두 개의 값을 받아들여 변수의 값을 서로 맞바꿔 주는 역할을 하는 외부 서브루틴이다. 기능은 두 개의 서브루틴이 동일하지만 각각 정수와 실수를 스왑 하는 점이 다르다.

```
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
```

똑 같은 기능을 하지만 정수를 스왑할 때는 swapint를 호출해야 하고 실수를 스왑할 때는 swapreal을 호출해 사용해야 한다. 이렇게 유사한 기능을 하는 두 개의 루틴은 다음과 같이 generic swap 인터페이스로 묶어 하나의 이름으로 정의해 사용할 수 있다.

```
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
```

위와 같이 generic 인터페이스를 정의해 두면 사용자는 정수와 실수의 구분 없이 프로그램 내에서 generic 프로시저 swap을 호출해 사용할 수 있다. 이때 입력으로 정수가 들어가면 swapint에 의해 결과가 리턴되고 실수가 들어가면 swapreal에 의해 결과가 리턴되게 된다.

```
......
INTEGER :: m, n
REAL :: x, y
......
```

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

사실 대부분의 고유함수들은 generic이어서 입력 인수에 의해 그 타입이 결정된다. 실제 예를 보면 고유함수 ABS(X)는 다음과 같이 동작한다.

- returns a real value if X is REAL
- returns a real value if X is COMPLEX
- returns a integer value if X is INTEGER

generic 인터페이스에는 함수와 서브루틴이 섞여 들어갈 수 없다. 모두 함수가 들어 가던지 아니면 모두 서브루틴만 들어가야 한다.

모듈은 이러한 generic 인터페이스를 정의하기 위해 많이 사용된다. generic 프로시저를 모듈 내에 정의하고 USE문을 이용해 첨부하면 generic 프로시저는 명시적 인터페이스를 가지게 된다.

```
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


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°¶
```

5.3.5 Module: Public and Private Object

기본적으로 모듈내의 모든 객체는 USE문을 이용해 그 모듈을 첨부한 모든 프로그램 단위에서 사용할 수 있다. 만약 일부 프로그램에서 특정 객체를 사용할 수 없도록 제한하고자 한다면 PRIVATE문이나 PRIVATE 속성을 사용할 수 있다.

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

임의의 모듈에 대한 기본적인 접근은 PUBLIC이다. 다음과 같이 IMPLICT NONE 다음에 PRIVATE문을 적어주면 기본적으로 모든 객체는 PRIVATE 속성을 가지게 된다.

```
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
```

5.3.6 Module: Compiling and Linking Programs and Modules

소스 프로그램으로부터 실행 프로그램을 생성하는 과정은 다음 두 단계로 이루어진다.

- Compilation: 소스 프로그램을 목적 프로그램(object program)이라 불리는 기계어 프로그램으로 변환하는 과정이다. (이때 생성되는 목적 파일의 확장자는 유닉스에서는 '.o', 도스에서는 '.obj' 가 된다.)

- Linking: 모듈에서 정의된 프로시저에 대한 참조를 모듈내의 정의와 연결(link)시키고 실행 프로그램을 생성한다.

모듈도 컴파일이 필요하기 때문에, 모듈을 첨부해 사용하는 프로그램의 변환은 다음과 같은 세 단계를 거치게 된다.

- 소스 프로그램을 컴파일 해서 목적 파일을 생성한다.

- 모듈을 컴파일 해서 또 다른 목적 파일을 생성한다.

- 프로그램의 목적파일에서 호출되는 함수와 모듈내의 함수 정의와 연결시키고 실행 프로그램을 생성한다.

소스 프로그램과 모듈의 컴파일에 우선 순위는 없다. 어떤 것을 먼저 컴파일 해도 상관 없지만, 링크가 수행되기 전에 모두 컴파일 되어 있어야 한다.

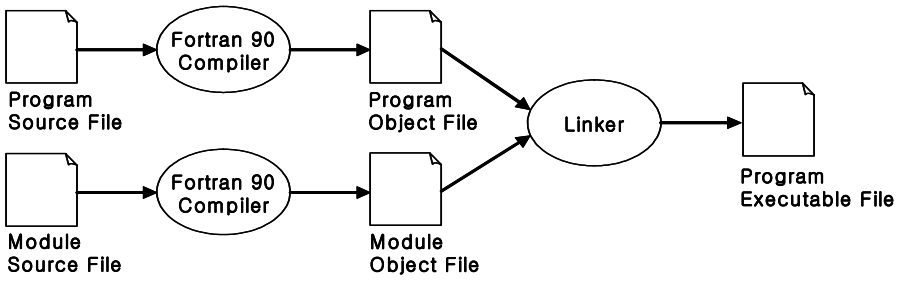


그림 5.2 Compiling and Linking Programs and Modules

5.4 Pointer

포인터 변수가 제공하는 기능은 그 변수에 의해 참조되는 공간을 바꿀 수 있도록 하고 동시에 그 공간에 저장된 값도 바꿀 수 있도록 하는 것이다. 여기서 포인터 변수가 가리키는(point to) 공간을 타깃(Target)이라 한다.

포인터는 변수에 간접적으로 접근할 수 있도록 하며, 사용자 정의 타입과 같이 사용되어 linked list, tree 등과 같은 동적 자료 구조 작성을 가능하게 한다. 포인터 변수는 값을 저장할 수 없으며 단지 데이터를 가지는 스칼라나 배열 변수들을 가리키는 역할만 하게 된다. Fortran90에서 제공하는 포인터는 C의 포인터와 같이 주소를 가지는 것도 아니다. C의 포인터 보다는 유용성 측면에서 떨어지지만 보다 최적화 되어 있다.

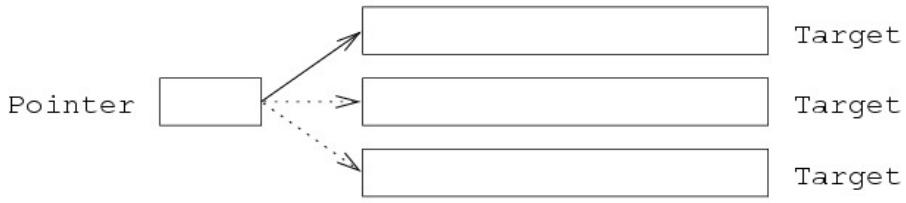


그림 5.3 The Relationship Between a Pointer and its Target(The Liverpool Fortran90 Courses)

5.4.1 Pointer Status

포인터 변수는 세 가지 상태를 가진다.

- undefined : the initial status of a pointer
- associated : the pointer has a target
- disassociated : the pointer has no target but is defined

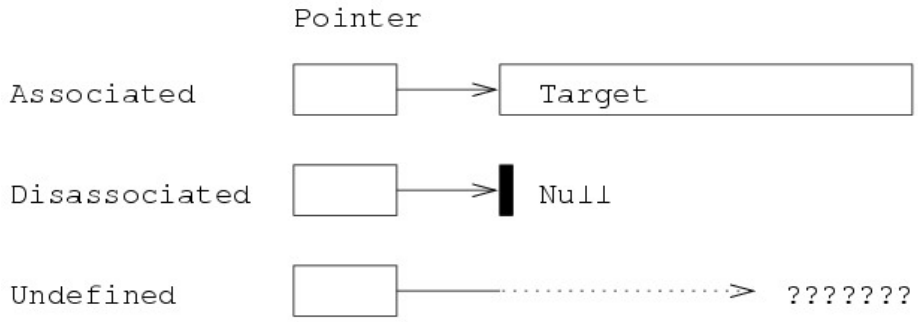


그림 5.4 The Three States of a Pointer(The Liverpool Fortran90 Courses)

ASSOCIATED는 포인터의 associated 상태를 리턴 해주는 LOGICAL 고유함수로 포인터의 상태가 associated 이면 .TRUE.를 리턴하고 disassociated이면 .FALSE.를 리턴 해준다.

5.4.2 Pointer Declaration

포인터 변수는 POINTER 속성으로 선언된다. 포인터 변수의 정적 타입, kind, 랭크(rank)[1] 등이 선언 단계에서 결정된다.

다음은 실수 스칼라 타깃을 가지는 포인터 변수 ptor과 랭크가 2인 실수 배열을 가리키는 포인터 변수 ptoa를 선언한 것이다. Fortran의 포인터는 문자를 가리킬 수 없다.

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

- 포인터 변수는 선언에 의해, 타깃의 타입, kind, 랭크 등이 고정된다.
- 배열을 가리키는 포인터 변수는 항상 deferred-shape으로 선언된다.
- 선언에 의해 타깃의 랭크는 고정되지만 그 모양은 변할 수 있다.

1. 랭크는 배열의 차원을 의미한다.

5.4.3 Target Declaration

포인터의 타깃이 되는 변수들은 선언될 때 TARGET 속성을 가져야 한다.

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

이와 같이 선언된 변수들은 앞서 선언된 포인터 변수 ptor과 ptoa에 대해

- x, y는 포인터 ptor과 대응(associated)될 수 있으며,
- a, b, c는 포인터 ptoa와 대응될 수 있다.

5.4.4 Pointer Manipulation

포인터에 값을 할당하는 다음 두 가지 방법은 그 결과가 매우 다르므로 사용에 주의가 필요하다.

'=>' pointer assignment

- 객체를 다른 이름으로 표현하는 대체(aliasing) 기능을 할 뿐이며, 이때 포인터와 그의 타깃은 동일한 공간을 나타낸다.
- 포인터와 타깃 또는 포인터와 포인터 사이에 올 수 있다.

'=', normal assignment

- 포인터 변수와 적절한 타입, kind, 랭크를 가지는 객체 사이에 올 수 있다. 이때, RHS의 객체는 TARGET 속성을 가질 필요가 없다.
- 포인터가 가리키는 공간에 저장되는 값을 설정한다. 여러 포인터가 가리키는 공간에 있는 값이 바뀌면 이를 가리키고 있는 모든 associated 포인터의 참조값이 바뀌게 된다.

포인터 할당은 포인터가 변수를 대체하여 동일 공간을 참조하게 만들며, 노멀 할당은 참조되는 그 공간에 저장된 값을 변경시키는 역할을 한다.

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

위는 포인터 ptor과 ptoa를 각각 타깃 y와 b에 대응 시키는 문장이다. 이때 포인터 ptor은 y를 대체(alias)하며, 포인터 ptoa는 b를 대체한다. 이제 프로그램에서 ptor과 y, 그리고 ptoa와 b는 서로 구분 없이 사용될 수 있다. 만약 y와 b의 값이 바뀌면 ptor과 ptoa의 값도 바뀌게 되지만, 가리키고 있는 공간의 위치에는 변함이 없다.

실수 스칼라를 타깃으로 가지는 새로운 포인터 ptor2에 대해 다음과 같은 포인터 할당이 가능하다. 이때 ptor2는 y를 가리키며 y를 대체하는 포인터는 두 개가 된다.

```
ptor2 => ptor
```

위 문장을 다음과 같이 표현할 수도 있다.

```
ptor2 => y
```

5.4.5 Pointer Assignment Example

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

- x = 3.14159에서 변수 x는 값을 할당 받는다.
- ptor => y에서 ptor는 y를 대체한다. 이때 y가 어떤 의미 있는 값을 가지고 있을 필요는 없다.
- ptor = x는 ptor이 가리키는 공간, 즉 y의 값을 변수 x의 값으로 설정한다. ptor에 대한 어떤 참조나 할당도 실제로는 y에 대한 참조나 할당이 되는 것이다.

이후 계속해서 x의 값이 변하게 되더라도 ptor과 y의 값은 변하지 않는다. 그러나 ptor = 5.0으로 설정하게 되면 y의 값은 5.0으로 바뀌게 된다.

```
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
```

5.4.6 Association with Arrays

배열 포인터는 타깃이 되는 배열 전체와 대응할 수 있고 랭크와 타입, kind 등이 맞는다면 타깃의 regular section[1] 과 대응할 수도 있다. Vector subscript 로 표현되는 non-regular 배열 부분은 배열 포인터와 대응할 수 없다.

앞서 정의 되었던 배열 포인터 ptoa에 대해 다음과 같은 표현은 유효하다.

```
ptoa => a(3::2, ::2)
```

이 포인터 할당은 배열의 특정 부분을 포인터 ptoa에 대체한 것이다. 여기서 ptoa(1,1)은 a(3,1)을 ptoa(2,2)는 a(5,3)을 나타낸다. SIZE(ptoa)는 4의 결과를 가지며 SHAPE(ptoa)는 (/2,2/)의 결과를 줄 것이다.

1. Linear function(subscript-triplet) 으로 정의되는 배열 부분

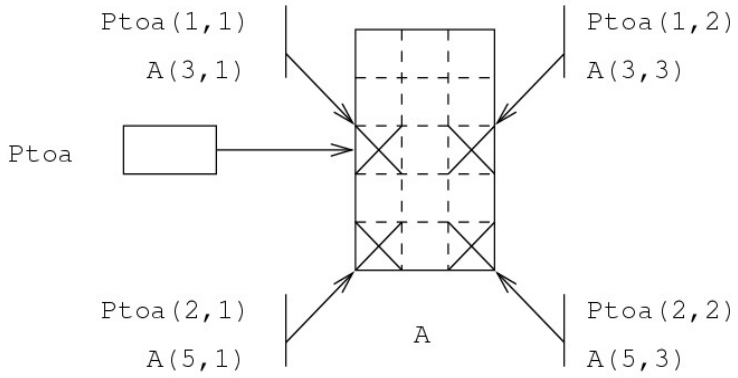


그림 5.5 A Pointer to a Section of an Array

다음은 포인터 ptoa가 1 X 1의 2차원 배열을 대체하는 것을 표현한 것이다. 이와 같이 비록 하나의 원소를 나타내더라도 랭크가 일치되면 배열 부분을 포인터로 대체할 수 있다.

```
ptoa => a(1:1, 2:2)
```

다음과 같은 표현들은 타깃의 랭크가 2가 아니므로 모두 틀린 표현들이 된다.

```
ptoa => a(1:1, 2)
ptoa => a(1, 2)
ptoa => a(1,2:2)
```

배열 포인터는 subscript에 의해 정의된 배열을 타깃으로 가질 수 없으므로 다음과 같은 포인터 할당은 잘못된 것이다.

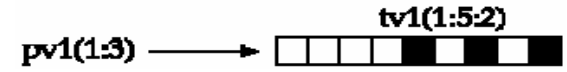
```
v = (/2,3,1,2/))
ptoa => a(v,v)
```

다음은 몇 가지 1차원과 2차원 배열 포인터에 의한 대체를 그림으로 나타낸 것이다.

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

pv1 => tv1 ! pv1 aliased to tv1

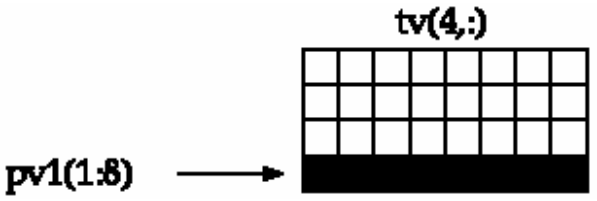
pv1=>tv1(:) ! aliased with section subscript



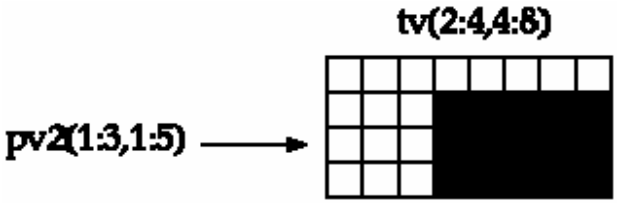
pv1=>tv1(1:5:2) ! aliased with section triplet

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

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



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



5.4.7 Dynamic Targets

포인터의 타깃은 동적 할당에 의해 생성될 수도 있다. 다음과 같이 ALLOCATE문을 이용해 포인터의 타깃이 되는 공간을 직접 예약할 수 있는데, 이때의 포인터는 다른 변수를 대체하는 역할 보다는 heap 영역의 이름없는 한 부분에 대한 참조의 역할을 하게 된다.

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

위의 두 문장에서 첫 문장은 실수, 두 번째 문장은 랭크가 2인 실수 배열을 위한 새로운 공간을 할당하게 되는데, 이 공간들은 포인터 ptor과 ptoa의 타깃으로 자동 생성되는 것이다.

프로시저에서 지역적으로 할당된 공간은 프로시저가 종료하기 전에 할당해제(deallocate) 되어야 한다. 그렇지 않으면 그 공간은 이 후 접근할 수 없는 낭비된 공간이 되어 버린다. 할당된 공간을 프로시저 호출 사이에서 계속 사용할 수 있도록 유지하려면 SAVE 속성을 가지도록 해야 한다.

ptor = 2. 과 같이 포인터에 어떤 값을 할당하려면 그 포인터는 ALLOCATE를 통해 예약된 공간을 가지거나 포인터 할당을 통해 대체 되어야 한다. 만약 그렇지 않다면 segmentation fault(core dump) 오류가 발생하게 된다.

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


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

5.4.8 Automatic Pointer Attributing

모든 포인터 변수는 암시적인 타깃 속성을 가지고 있어서 다른 포인터에 의해 대응될 수 있다.

```
ptoa => a(3::2, 1::2)
ptor => ptoa(2,1)
```

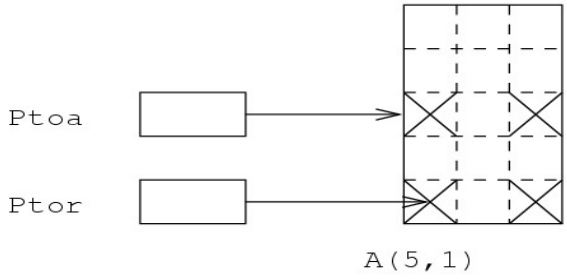


그림 5.6 Automatic Attributing of Arrays

위 그림에서 좌측 맨 아래 원소는 세 가지 이름을 가지게 된다.

```
ptor
a(5,1)
ptoa(2,1)
```

포인터를 타깃으로 할 때는 다음과 같은 상황(dangling pointer)이 발생하지 않도록 주의해야 한다.

```
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
```

5.4.9 Pointer Disassociation

포인터와 타깃의 대응을 해제하는 방법에는 두 가지가 있다.

Nullification: NULLIFY(ptor)

NULLIFY 문은 포인터와 타깃의 연결을 끊어 그 포인터의 상태를 대응 해제된 상태로 설정해 주지만 타깃의 공간을 deallocate 하지는 못한다. 이 공간은 다른 포인터에 대응되어 사용되지 않는 한 프로그램이 종료될 때까지 접근할 수 없는 공간이 된다.

```
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.
```

다음과 같은 상황은 메모리 공간의 낭비를 가져오게 되므로 피해야 한다.

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

Deallocation: DEALLOCATE(ptoa, STAT=ierr)

DEALLOCATE문은 ALLOCATE에 의해 대응되는 공간을 가지는 포인터와 타깃의 연결을 끊고 타깃의 공간을 재사용할 수 있도록 해준다. Allocate 되지 않았거나 대응해제 또는 정의되지 않은 상태를 가지는 타깃에 대한 deallocation은 오류를 발생시킨다.

5.4.10 Pointers to Arrays vs. Allocatable Arrays

일반적으로 ALLOCATABLE 배열은 POINTER 배열보다 성능이 우수해 많이 사용된다. 포인터의 타깃이 되는 배열은 대체되는 여러 다른 이름으로 참조될 수 있지만 ALLOCATABLE 배열은 오직 하나의 이름을 가지게 된다. 단, 대체(aliasing)가 분명하게 필요하다면, 포인터와 타깃이 사용되어야 한다.

ALLOCATABLE 배열이 가지는 다음과 같은 제이 있다.

- unallocated ALLOCATABLE arrays cannot be passed as actual arguments to procedures
- ALLOCATABLE arrays cannot be used as components of derived types

결론적으로 ALLOCATABLE 배열이 더 효율적이고, 포인터 배열은 더 유연하다.

할당여부와 상관없이 포인터는 프로시저의 인수로 사용될 수 있어 전체 배열을 패싱 하기에 편리하다. ALLOCATABLE 배열은 더미 인수로 사용될 수 없으며 단일 프로그램 단위에서 allocated, deallocated 되어야 한다.

프로시저가 포인터나 타깃을 더미 인수로 가지는 경우 그 프로시저의 인터페이스는 반드시 명시적이어야 한다. 더미 인수가 포인터라면 실제 인수는 그와 같은 타입, kind, 랭크를 가져야 한다. 포인터 더미 인수는 INTENT 속성을 가질 수 없다. 실제 인수는 포인터지만 더미 인수가 포인터가 아니라면 그 더미 인수는 포인터의 타깃과 대응된다.

```
INTERFACE              ! do not forget interface in calling unit
SUBROUTINE sub2(b)
REAL, DIMENSION(:, :), POINTER :: b
END SUBROUTINE sub2
END INTERFACE

REAL, DIMENSION(:, :), POINTER :: p
ALLOCATE (p(50, 50))
CALL sub1(p)           ! both sub1 and sub2
CALL sub2(p)           ! are external procedures
...
SUBROUTINE sub1(a)     ! a is not a pointer
REAL, DIMENSION(:, :) ::
...
END SUBROUTINE sub1

SUBROUTINE sub2(b)     ! b is a pointer
REAL, DIMENSION(:, :), POINTER :: b
DEALLOCATE(b)
END SUBROUTINE sub2
```

5.4.11 Pointer Valued Functions

함수의 결과는 포인터가 될 수 있다. 이때 포인터 함수의 인터페이스는 반드시 명시적 이어야 한다. 다음 함수 ptr은 a와 b를 비교하여 큰 값을 리턴 해주는 기능을 한다. FUNCTION문이 속성을 가질 수 없으므로 함수의 결과를 포인터로 처리하고 있다.

```
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
```

함수의 결과가 포인터가 될 수 있다는 사실은 다음과 같이 결과의 크기가 수행되는 계산에 의존하는 경우 유용하게 사용될 수 있다.

```
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
```

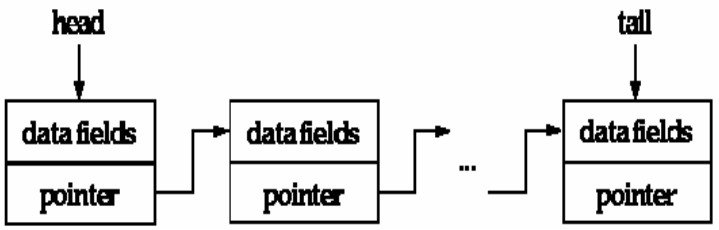
```
END FUNCTION gtzero
...
END
```

5.4.12 Practical Use of Pointer: Linked List

유도 타입이 가지는 포인터 항은 같은 타입의 객체를 대체할 수 있으며, 이를 이용해 다음과 같은 linked list를 만들 수 있다.

```
TYPE node
INTEGER :: value ! data field
TYPE (node), POINTER :: next ! pointer field
END TYPE node
```

일반적으로 Linked list는 data field와 list내에서 같은 타입을 가지는 다음 객체에 대한 pointer field로 구성되는 유도 타입으로 구성된다.



연결되는 객체들은

- 연속적으로 저장되어 있지 않아도 된다.
- 실행 중에 동적으로 생성될 수 있다.
- list 내의 임의의 위치에 삽입되어도 된다.
- 동적으로 삭제할 수 있다.

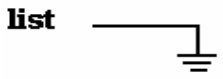
list의 크기는 프로그램이 실행되면서 임의의 크기로 증가될 수 있는데, 이와 같은 방식으로 트리 혹은 다른 동적 구조들을 생성할 수 있다.

Linked List Creation

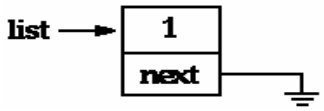
```
TYPE node
INTEGER :: value ! data field
TYPE (node), POINTER :: next        ! pointer field
END TYPE node

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
...
```

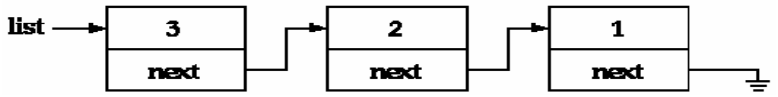
After NULLIFY(list)



After the first num is read



After all 3 numbers are read



5.5 Operator Overloading and User Defined Operator

여기서는 Fortran90에서 제공하는 고유 연산자의 의미를 확장해 사용할 수 있는 연산자 overloading과 사용자가 새로운 연산자를 정의해 사용하는 사용자 정의 연산자에 대해 다룰 것이다.

5.5.1 Overloading Operators

+, =, * 등과 같은 고유 연산자들은 추가적인 데이터 타입들에 적용될 수 있도록 그 의미를 확장해 사용할 수 있는데 이를 연산자 오버로드라 한다. 연산자 오버로드는 모듈 안에 캡슐화하여 사용한다.

- INTERFACE OPERATOR문으로 generic 연산자 심볼을 정의한다.
- Generic 인터페이스로 오버로드 셋을 정의한다.
- 어떻게 연산이 수행되는지 정의한 모듈 프로시저를 선언한다.

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

다음은 캐릭터 타입 변수에 대해 '+' 연산을 오버로드하여 사용하는 예이다.

```
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
```

5.5.2 Defining New Operator

Fortran90에서는 이미 존재하는 고유 연산자를 오버로드하여 사용할 수도 있고 연산자를 새롭게 정의해 사용할 수도 있다. Fortran에서 .AND.나 .OR.와 같이 non-symbolic 연산자를 나타내는 도트 표기를 이용해 새로운 연산자를 표현한다.

.<name>.

여기서 연산자의 이름 <name>에는 반드시 문자만 올 수 있다. 모든 연산자의 정의에는 하나 또는 두 개의 INTENT(in) 인수가 있어서 단항 또는 이항 연산을 표현한다.

다음은 두 점 사이의 거리를 구하는 새로운 연산자 .dist.를 정의해서 사용하는 예이다.

```
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
```

+, - 보다 *, /의 우선 순위가 높은 것처럼 모든 연산에는 우선 순위가 있다. 사용자가 정의한 연산 중 단항 연산은 모든 연산에서 가장 우선 순위가 높으며 이항 연산은 우선 순위가 가장 낮다.

5.5.3 Assignment Overloading

같은 고유 타입 또는 같은 유도 타입을 가지는 두 객체 사이의 할당은 고유하게 정의되어 있으나 서로 다른 유도 타입을 가지거나 유도 타입과 고유 타입 사이의 할당은 프로그램상에서 명시적으로 정의 되어야 한다.

할당의 오버로딩은 연산자 오버로딩과 같이 할 수 있으나, 할당을 정의하는 프로시저가 두 개의 인수를 가지는 서브루틴이 된다는 점이 다르다. 이때 서브루틴의 두 인수는 다음과 같은 특성을 가진다.

- 첫 인수는 할당된 값을 받는 변수로 INTENT(out) 속성을 가지며 실제 할당 표현에서 LHS가 된다.

- 두 번째 인수는 값이 전환되어 결과로 할당되는 변수로 실제 할당 표현에서 RHS가 된다. 이에 대응하는 더미 인수는 INTENT(in) 속성을 가진다.

다음은 유도 타입을 가지는 변수 px의 값을 실수 ax에 할당하기 위해 할당 연산 '='를 오버로드하여 사용하는 예이다.

```
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


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
```

5.6 Recursive Procedure

되부름(recursion)은 프로시저가 직간접적으로 자기 자신을 호출할 때 발생한다. 다양한 문제영역에서 되부름이 간결하고 깔끔한 기법이기는 하지만 잘못 사용되면 프로그램의 효율에 과부하를 줄 수 있다.

Fortran77에서도 사용자 정의 스택과 대응하는 조작 함수를 이용해 되부름을 구현해 볼 수 있지만 Fortran90에서는 언어자체가 명시적으로 되부름 기능을 지원하고 있다. 효율을 높이기 위해 되부름 프로시저(SUBROUTINE and FUNCTION)는 RECURSIVE 키워드를 통해 명시적으로 선언되어야 한다.

되부름 함수는 결과를 함수 이름으로 리턴할 수 없고, 그래서 일반적인 함수 선언과 조금 다르게 사용된다. 되부름 함수에서 그 결과는 RESULT 키워드를 이용해 지정된 변수를 통해 리턴된다. 이때 함수 이름과 결과 이름은 암시적으로 같은 속성을 가지게 된다.

```
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 함수를 선언하는 위의 세 가지 방법은 모두 동일하다. 이렇게 되부름 함수임을 RECURSIVE 키워드를 통해 미리 선언해 두어야 하고 INTEGER 속성은 fact와 n_fact에 동일하게 적용된다. 세 번째 방법과 같이 선언하는 경우도 역시 n_fact의 타입은 함수 fact에 그대로 적용되어 정수 타입이 된다. 여기서 만약 함수 타입을 INTEGER로 다시 선언해 버리면 중복 선언이 되어 프로그램 오류가 된다.

다음은 정수 n의 계승(factorial)을 계산하기 위해 되부름 함수를 이용하는 프로그램이다.

```
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
```

서브루틴도 RECURSIVE 키워드로 선언해서 되부름 기능을 할 수 있다.

```
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
```

5.7 Keyword/Optional Arguments

Fortran 고유함수에서는 인수에 키워드를 붙여 사용할 수 있다.

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

사용자가 정의해 사용하는 프로시저에서도 이와 같이 키워드를 이용한 인수 대응이 가능하다.

5.7.1 Keyword Arguments

첫 번째 인수는 첫 번째 더미 인수에 대응되고 두 번째 인수는 두 번째 더미 인수에 대응되는 식으로 일반적인 인수 대응은 위치에 의해 이뤄지게 된다. Fortran90에서는 인수의 대응을 사용자가 지정할 수 있는 기능을 제공하고 있다. 키워드를 이용하여 위치 기준의 인수 대응을 키워드 대응으로 바꿀 수 있으며 이때의 키워드는 더미 인수의 이름과 동일하다.

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

서브루틴 axis의 호출을 위해 다음과 같은 방법이 가능하다.

- 위치 대응 호출:

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

- 키워드 인수를 이용한 호출:

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

인수 하나가 키워드로 사용되면 이어서 오는 모든 인수들도 반드시 키워드에 의해 대응 되어야 한다. 이러한 키워드 인수 대응은 명시적 인터페이스를 가질 때 사용할 수 있다.

5.7.2 Optional Arguments

명시적 인터페이스를 가지는 프로시저의 인수는 OPTIONAL 속성으로 선언해 선택적으로 사용될 수 있다. OPTIONAL 속성을 가지는 더미 인수는 실제 인수 리스트에서 생략될 수 있다. 만약 하나의 인수가 생략되면 그 이후 나오는 모든 인수는 키워드 대응을 해서 컴파일러에게 인수의 올바른 대응을 알려줘야 한다.

Fortran의 많은 고유 프로시저들은 이러한 OPTIONAL 인수를 사용하고 있으며, 사용자가 정의한 프로시저에서도 마찬가지로 OPTIONAL 인수를 사용할 수 있다.

고유함수 PRESENT는 OPTIONAL 인수의 상태를 알아보기 위해 사용하며 OPTIONAL 인수에 값이 할당되었을 때 .TRUE. 할당되지 않았을 때 .FALSE.를 리턴해 준다.

```
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
 ...
```

인수 a, b가 OPTIONAL 속성으로 선언되었기 때문에 서브루틴 SEE는 다음과 같이 다양한 방식으로 호출될 수 있다.

```
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)
```

OPTIONAL 속성을 가지는 더미 인수에 실제 인수를 대응시키지 않을 경우 대응이 없는 더미 인수에는 디폴트 값이 대체되어 들어가며 그 인수에 대한 PRESENT 함수 값은 .FALSE.가 되는 것이다.

제 6 장 참고문헌

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)

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)

6.Fortran90 for the Fortran77 Programmer (http://www.whoi.edu/CIS/ training/ classes/f77to90/f77to90.html)

7.Larry R. Nyhoff, Sanford C. Leestma. Fortran90 for Engineers and Scientists. Prentice Hall. 1997.

찾아보기