** **

ARTICLES

** **

Jeanne Martin: Fortran 90 Pointers vs. "Cray" Pointers†

Lawrence Livermore National Laboratory
October 17, 1991

The Fortran 77 standard does not contain pointer facilities, but because of heavy user demand, many Fortran 77 compilers have been extended with "Cray" pointers. The demand for pointers in Fortran was heard by the standards committee, X3J3, and a pointer facility was added to the follow-on Fortran standard, Fortran 90. X3J3, for reasons that may soon become apparent, chose not to follow existing practice and specify "Cray" pointers, but to standardize a somewhat different pointer facility. Fortran 90 pointers complement the Fortran 90 language; they fit well with the new Fortran 90 array processing and data facilities. The popularity of "Cray" pointers indicates that they fit well with Fortran 77, and since Fortran 90 contains all of Fortran 77, it would be possible to extend Fortran 90 processors to accept "Cray" pointers as well. This would make it easier for existing codes that use pointers to migrate to new Fortran 90 processors. But is it a good idea for a processor to provide two pointer facilities? How difficult is it to convert from "Cray" pointers to Fortran 90 pointers? This paper provides some information that may be helpful in answering these questions.

Description of the Two Pointer Facilities

The meta language used in this paper to describe pointer-related statements makes use of square brackets [ ] to indicate optional elements and ellipses ... to indicate multiple occurrences of preceding optional elements.

"Cray" Pointers. The *POINTER* statement takes the form:

POINTER (*pointer* , *target* [(*dimensions*)]) [, (*pointer* , *target* [(*dimensions*)])]...

Examples are:

```
POINTER (PA, A), (PB, B)
POINTER (PX, X(1)), (PY, Y(1))
POINTER (PZ, Z(0:IMAX, 0:JMAX))
REAL A, B, X, Y, Z
```

The *POINTER* statement associates two names: the name of a pointer and the name of its target. No space is set aside for the target (such as **A, B, X, Y,** or **Z)**, so initially it can have no value. First an initial address for the target must be assigned to the pointer. Then whenever the target is referenced, its initial address is obtained from the pointer. The pointer (such as **PA, PB, PX, PY,** or **PZ)** is assumed to be of type integer, and it can be assigned an integer value that is taken to be an absolute address:

---

†This work was performed under the auspices of the U.S. Department of Energy by Lawrence Livermore National Laboratory under contract No. W-7405-Eng-48.

NVIDIA Ex1012
NVIDIA Corp. v. Neural AI, LLC
IPR2025-00606

```
PA = 0
PX = 2048
```

Integer arithmetic may be performed on a pointer. A pointer value may be assigned to an integer variable, but not to a real variable. For example,

```
INTEGER I
PB = PA + 64
I = PX
PY = I + 3
```

A pseudo intrinsic function, *LOC,* is provided to return the address of a variable. It may be used to initialize or reset pointers.

```
REAL BUFFER (1000)
PA = LOC(BUFFER) + 10
```

For optimization purposes, the compiler assumes that a pointer target never has storage in common with another variable. It is the responsibility of the user to prevent this kind of storage association or aliasing.

There is no way to indicate that a pointer contains no valid address. If an address of zero is not valid for a particular application, then a pointer value of zero is sometimes used for this purpose.

The essential syntactic parts of this facility are a *POINTER* statement that associates two names and a function *LOC* that returns the address of a variable. For other essential elements, "Cray" pointers rely on existing features of the Fortran language, such as ordinary assignment and integer arithmetic. They may also rely on the assumption that zero is not a valid pointer value and can be used to indicate that a pointer is currently not in use. In some environments, such an assumption may be highly error prone. For a complete pointer facility some kind of memory management library is also required. It might be supplied by a vendor; if not, users may find it advantageous to create such a library.

Fortran 90 Pointers. *POINTER* is an attribute of an object in Fortran 90, and there are two ways that attributes may be declared: in separate statements (attribute-oriented declaration) or in a type statement (entity-oriented declaration). The entity-oriented declaration is generally preferred because it permits all of the attributes of an object to be specified in one statement. Attribute-oriented declaration makes use of a *POINTER* statement of the form:

POINTER [: :] ***pointer*** [(***dimensions***)] [, ***pointer*** [(***dimensions***)]]...

Examples are:

```
POINTER A, B
POINTER X, Y
DIMENSION X(:), Y(:)
POINTER Z(:,:)
REAL A, B, X, Y, Z
```

The form of a type declaration statement (simplified) is:

type [[, ***attribute***]... : :] ***entity*** [, ***entity***]...

Examples are:

```
REAL, POINTER :: A, B
REAL, POINTER, DIMENSION(:) :: X, Y
REAL, POINTER :: Z(:,:)
```

For both forms, the dimension information may appear either in an attribute list or be attached to an entity name. No space is set aside for the entities **A, B, X, Y,** or **Z**. Their pointer association status is undefined until it is set by pointer assignment, allocation of space for the target, or a *NULLIFY* statement. A Fortran 90 pointer can **not** be assigned an absolute address. Fortran 90 pointers are descriptors containing

NVIDIA Ex1012
NVIDIA Corp. v. Neural AI, LLC
IPR2025-00606

type and rank information. No arithmetic can be performed on the address part of the descriptor because there is no way to reference it.

An *ALLOCATE* statement is provided to acquire space for a pointer target. The form of the *ALLOCATE* statement is:

ALLOCATE (*pointer* [(*dimensions*)] [, *pointer* [(*dimensions*)]... [, **STAT** = *status*])

Examples are:

```
ALLOCATE (A, X(1000))
ALLOCATE (Z(0:IMAX, 0:JMAX), STAT = IERR)
```

An optional specifier, **STAT**, may be used to designate a variable, such as **IERR**, that will have a positive integer value if an error occurred during execution of the *ALLOCATE* statement. A somewhat similar *DEALLOCATE* statement is also provided.

A pointer assignment statement is used when the target space already exists. The form of the pointer assignment statement is:

pointer => *target*

For example:

```
REAL, TARGET :: BUFFER (1000)
A => BUFFER(11)
Y => X
```

A is a scalar real variable. In the first pointer assignment statement, it becomes an alias for the eleventh element of **BUFFER**. The second pointer assignment statement causes **Y** to have the same target as **X**. Fortran 90 does not provide for a pointer to a pointer.

The *TARGET* attribute is provided merely to aid optimization. The compiler assumes that pointers and objects with the *TARGET* attribute may have aliases, and thus code that refers to these objects may be limited in the ways in which it can be optimized. There is an attribute-oriented declaration for the *TARGET* attribute as well. It has the form:

TARGET [::] *target* [(*dimensions*)] [, *target* [(*dimensions*)]]...

Fortran 90 provides a *NULLIFY* statement to set a pointer of any type and rank to point to no object. Pointer status (whether associated with a target or disassociated) can be tested using the **ASSOCIATED** intrinsic function when there is one argument. The **ASSOCIATED** intrinsic function with two arguments can be used to test whether a pointer is associated with a particular target or whether two pointers are associated with the same target.

The essential syntactic parts of the Fortran 90 pointer facility are two attributes: *POINTER* and *TARGET;* four statements: *ALLOCATE, DEALLOCATE, NULLIFY,* and the pointer assignment statement that uses the symbol => ; and one intrinsic function: **ASSOCIATED**.

Basic Differences Between Fortran 90 and "Cray" Pointers

There are many differences between the two facilities, and some similarities. One fortunate similarity, the appearance of a name in statements expressing operations on the target object, means that these statements are the same, regardless of the pointer facility used. Thus, in many conversions between the two facilities, it is only the declaration and initialization of pointers that must be changed. The basic differences are:

NVIDIA Ex1012
NVIDIA Corp. v. Neural AI, LLC
IPR2025-00606

- There are two names associated with a "Cray" pointer: the name of the pointer and the name used to refer to the pointer target. There is only one name associated with a Fortran 90 pointer. The interpretation of the name is determined by context; that is, certain statements can refer only to pointers, while others can refer only to target objects.
- "Cray" pointers are memory locations. The pointer is treated as an integer and the type of the target can change during execution. Fortran 90 pointers are descriptors containing both type and rank information. They point to specific kinds of data objects; that is, Fortran 90 pointers are "strongly typed". For example, a given pointer may be declared to point to two-dimensional real arrays. It can never point to any other kind of data object. It may point to several different arrays and the dimensions may vary, but it can never point to a scalar integer object, for instance.
- *POINTER* is an attribute in Fortran 90. This attribute may be given to any data object, including an object of user-defined type. In some implementations of "Cray" pointers, it is not possible to point to objects of type **CHARACTER**. An implementer might find it equally difficult to extend "Cray" pointers to point to some of the new Fortran 90 data objects such as objects of user-defined type or nondefault kind. An implementer who added "Cray" pointers to a standard Fortran 90 compiler would have to decide whether to make the "Cray" pointers pervasive throughout the language or to provide them only to aid code migration (that is, applying only to Fortran 77 features).
- "Cray" pointers can be assigned absolute addresses; Fortran 90 pointers cannot.
- A Fortran 90 pointer can be set to point to no object by a *NULLIFY* statement. There is no language-provided way to "nullify" a "Cray" pointer.
- A compiler may assume that a "Cray" pointer target has no storage in common with another variable. This provides optimization at the expense of possibly unreliable code. A Fortran 90 compiler assumes that pointers and objects with the *TARGET* attribute may overlap. This provides reliable code at the expense of some optimization.

Typical Usage

There are at least two reasons why pointers are needed. The most frequently mentioned reason is so that Fortran programmers can manage the memory required by their applications. Usually the amount of memory needed is not known until various sizes are read in as input or calculated. The Fortran 77 requirement that arrays must be declared with the largest sizes that will ever be required is wasteful of memory and may, in some cases, prevent a needed configuration from being executable. The following example shows how memory can be managed using both "Cray" pointers and Fortran 90 pointers.

"Cray" pointers	**Fortran 90 pointers**
declaration	
`REAL A(1), B(1), C(1)` `POINTER (PA, A), (PB, B), (PC, C)`	`REAL, POINTER :: A(:), B(:), C(:)`
pointer initialization	

`COMMON POOL(100000)`
`REAL POOL`
`READ (*,*) N`
`PA = LOC(POOL)`
`PB = PA + N`
`PC = PB + N`
	`COMMON POOL(100000)`
	`REAL, TARGET :: POOL`
	`READ (*,*) N`
	`A => POOL(1:N)`
	`B => POOL(N + 1:2*N)`
	`C => POOL(2*N+1:3*N)`

NVIDIA Ex1012
NVIDIA Corp. v. Neural AI, LLC
IPR2025-00606

use in calculations

```
      ...
      DO 10 I = 1, N
        A(I) = B(I) + C(I)
10    CONTINUE
```

```
      ...
      DO 10 I = 1, N
        A(I) = B(I) + C(I)
10    CONTINUE
```

or

```
      A = B + C
```

Note that the loops for both pointer facilities in the section *use in calculations* are identical, which shows that in a conversion process from "Cray" pointers to Fortran 90 pointers, the code specifying calculations would not require changes. Fortran 90 processes loops just as Fortran 77 does, but when Fortran 90 whole array facilities are available, it is convenient to take advantage of them and eliminate some program loops to simplify the code.

Instead of using existing space for pointer targets, frequently it is desirable to "acquire" the space for the period of time in which it is needed. In Fortran 90, this is accomplished through *ALLOCATE* and *DEALLOCATE* statements in the language. With Fortran 77 this is usually accomplished by a vendor-supplied memory management library. If the library has an allocation function named *ALLOC* that takes a pointer and a size as arguments, acquisition of space could be expressed as follows:

"Cray" pointers plus MM Library | **Fortran 90 pointers**

space acquisition

```
      READ (*,*) N
      IERR = ALLOC (PA, N)
      IERR = ALLOC (PB, N)
      IERR = ALLOC (PC, N)
```

```
      READ (*,*) N
      ALLOCATE (A(N), B(N), C(N), STAT = IERR)
```

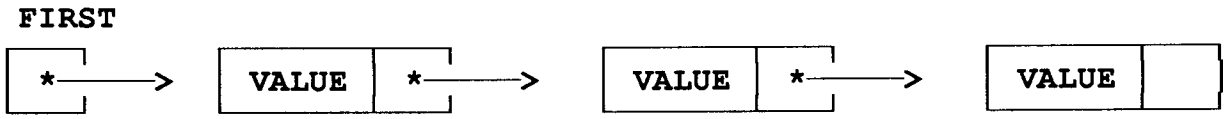
In the "Typical Usage" example, these statements would replace the section *pointer initialization.*

The entire example could be expressed in Fortran 90 as:

```
      REAL, POINTER :: A(:), B(:), C(:)
      READ (*,*) N
      ALLOCATE (A(N), B(N), C(N), STAT = IERR)
      ...
      A = B + C
```

Linked List Example

A second way that pointers are used is in the creation of structures such as linked lists or trees. A linked list can be represented as follows:



where the arrows represent pointers. The following code fragments demonstrate how "Cray" pointers and Fortran 90 pointers can be used to create, and later walk through, such a structure. While not relevant to the pointer comparison, these fragments also demonstrate that Fortran 90, with its user-defined types, has better mechanisms for representing such data structures. Those who have never needed such structures can skip this example and move on to the discussion of the distinct advantages of each of the pointer facilities.

NVIDIA Ex1012
NVIDIA Corp. v. Neural AI, LLC
IPR2025-00606

Fortran 77 plus "Cray" pointers	Fortran 90 pointers

```
C     Define data structures that have two components.  The first is a real value.  The second
C     is a pointer.

      REAL FIRSTV(2), CURRV(2), LASTV(2)                TYPE LINK
      INTEGER FIRSTN(2), CURRN(2), LASTN(2)               REAL VALUE
      EQUIVALENCE (FIRSTV, FIRSTN)                        TYPE (LINK), POINTER :: NEXT
      EQUIVALENCE (CURRV, CURRN), (LASTV, LASTN)          END TYPE LINK

C     Declare pointers

      POINTER (PFIRST, FIRSTV), (PFIRST, FIRSTN)        TYPE(LINK),POINTER :: FIRST, CURRENT, LAST
      POINTER (PCURR, CURRV),   (PCURR, CURRN)
      POINTER (PLAST, LASTV),   (PLAST, LASTN)

C     Initialize pointers

      IERR = ALLOC (PFIRST, 2)                          ALLOCATE (FIRST)
      PLAST = 0                                         NULLIFY (LAST)
      PCURR = PFIRST                                    CURRENT => FIRST

C     Create linked list where FIRST refers to the first element of the list and LAST refers to
C     the last element of the list.  LAST will change as the list grows.  Initially CURRENT is
C     the first element.  It is allocated before it is known whether it will be required and thus
C     when it is known that the list is complete because the last value has been read, the most
C     recent CURRENT can be deallocated.

      DO 10 I = 1, 1000000                              DO
        READ (*, *, IOSTAT = IOEM) CURRV(1)               READ (*, *, IOSTAT = IOEM) CURRENT % VALUE
        IF (IOEM .NE. 0) GO TO 11                         IF (IOEM .NE. 0) EXIT
        PLAST = PCURR                                     LAST => CURRENT
        IERR = ALLOC (PCURR, 2)                           ALLOCATE (CURRENT % NEXT)
        LASTN(2) = PCURR                                  CURRENT => CURRENT % NEXT
10    CONTINUE                                          END DO
11    IF (PLAST .NE. 0) THEN                            IF (ASSOCIATED(LAST)) THEN
        LASTN(2) = 0                                      NULLIFY (LAST % NEXT)
      ELSE                                              ELSE
        PFIRST = 0                                        NULLIFY (FIRST)
      END IF                                            END IF

C     Release unused space (Assume the vendor-supplied memory management library contains a
C     DALLOC function for this purpose.)

      IERR = DALLOC (PCURR, 2)                          DEALLOCATE (CURRENT)

C     A list is now constructed.  To walk through the list:

      PCURR = PFIRST                                    CURRENT => FIRST
      DO 20 I = 1, 1000000                              DO
        IF (PCURR .EQ. 0) GO TO 21                        IF (.NOT.ASSOCIATED(CURRENT)) EXIT
        WRITE (*, *) CURRV(1)                             WRITE (*, *) CURRENT % VALUE
        PCURR = CURRN(2)                                  CURRENT => CURRENT % NEXT
20    CONTINUE                                          END DO
21    CONTINUE
```

Advantages of Fortran 90 Pointers

Because Fortran 90 pointers are descriptors containing bounds and stride information, it is possible to point to part of a variable; for example, a row, the interior elements, or the corners of a matrix:

```
REAL, TARGET :: MATRIX(M, N)
REAL, POINTER :: ITH_ROW(:), INNER(:,:), CORNERS(:,:)
        ...
ITH_ROW => MATRIX(:, I)
INNER => MATRIX (2:M-1, 2:N-1)
CORNERS => MATRIX (1:M:M-1, 1:N:N-1)
```


NVIDIA Ex1012
NVIDIA Corp. v. Neural AI, LLC
IPR2025-00606

The array variables **ITH_ROW**, **INNER**, and **CORNERS** can be used in calculations and thus contribute to more meaningful Fortran statements. Since "Cray" pointers are initial addresses only, they cannot provide this functionality. In general, Fortran 90 code using pointers is easier to read, and consequently easier to maintain, than the corresponding Fortran 77 code with the addition of "Cray" pointers and a vendor-supplied memory management library.

With Fortran 90, it is possible to create dangling pointers (pointers whose targets no longer exist) and it is possible to allocate space that subsequently becomes inaccessible (the target exists, but there is no longer a pointer to it). But, on the whole, Fortran 90 pointers are much safer and more reliable than "Cray" pointers.

Advantages of "Cray" Pointers

There are two usages that are available with "Cray" pointers but not with Fortran 90 pointers. First, "Cray" pointers can be set to point to absolute addresses, which is very useful in certain system programming applications. Second, there are situations in which a multi-valued object must be accessed before it is known what kind of object it is. Information about the type of the object may be found at a known location relative to the initial address of the object. This situation can be handled with "Cray" pointers, but cannot be handled with the strongly typed Fortran 90 pointers.

Both of these usages are encountered most frequently in systems programming and seldom encountered in scientific programming. Both usages are seldom portable; that is, when they are used, the code must be changed if it is moved to a different platform. Since one of the primary aims of standardization is to enhance portability, these usages were not considered to impose high priority requirements for a scientific language such as Fortran and are thus not available with Fortran 90 pointers.

Coexistence

Could the two pointer facilities coexist in a single compiler? Certainly, a vendor can implement anything if the motivation is there. A standard conforming Fortran 90 compiler must provide Fortran 90 pointers. They are well suited to the language and fit with the other features in the language. Should a vendor provide "Cray" pointers as well? There is strong motivation to make it easy for users to migrate to a new compiler. "Cray" pointers are heavily used in certain applications; however they are used only with existing Fortran 77 features. If a vendor does choose to provide both kinds of pointers, then the "Cray" pointers should probably be restricted to the existing Fortran 77 functionality and not extended to work with the new features of Fortran 90.

References:

ISO/IEC 1539:1991 International Standard Programming Language Fortran, ISO Publications Dept., Geneva, Switzerland, August 1991

CF77™ Compiling System, Volume 1: Fortran Reference Manual SR-3071 4.0, Cray Research, Inc., Eagan, MN, June 1990

NVIDIA Ex1012
NVIDIA Corp. v. Neural AI, LLC
IPR2025-00606