ĐẠI HỌC QUỐC GIA HÀ NỘI
TRƯỜNG ĐẠI HỌC KHOA HỌC TỰ NHIÊN

PHAN VĂN TÂN

NGÔN NGỮ LẬP TRÌNH FORTRAN 90

HÀ NỘI – 2004

MỞ ĐẦU

Tập hợp các qui tắc đặc biệt để mã hoá những kiến thức cho máy tính hiểu được gọi là ngôn ngữ lập trình. Có rất nhiều những ngôn ngữ như vậy, ví dụ FORTRAN, BASIC, Pascal, C,... FORTRAN là tên cấu tạo từ FORmula TRANslation (công thức dịch), là ngôn ngữ lập trình bậc cao đầu tiên. Nó có thể sử dụng những tên tượng trưng để biểu diễn định lượng toán học và viết các công thức toán học dưới dạng thức hợp lý có thể hiểu được, như X=B/(2*A). Ý tưởng của FORTRAN được John Backus đề xuất vào khoảng cuối năm 1953, ở New York, và chương trình FORTRAN đầu tiên đã được chạy vào tháng 4 năm 1957.

Việc sử dụng FORTRAN đã nhanh chóng được phổ biến rộng rãi. Điều đó đòi hỏi cần phải sớm tiêu chuẩn hoá nó sao cho chương trình viết ra phải bảo đảm chạy được ở mọi nơi. Vào năm 1966 lần đầu tiên phiên bản chuẩn của ngôn ngữ lập trình này được ấn hành. Phiên bản này, như đã biết, là Fortran 66 (chính xác hơn là FORTRAN 66, nhưng thực tế người ta cho cách viết hoa là không trang trọng). Phiên bản chuẩn mới, Fortran 77, được ấn hành vào năm 1978. Không bằng lòng với sự cạnh tranh của các ngôn ngữ mới như Pascal và C, FORTRAN tiếp tục phát triển một cách mạnh mẽ. Và phiên bản chuẩn gần đây, FORTRAN 90 (hoặc Fortran 90), ra đời vào tháng 8 năm 1991. Cho đến nay, FORTRAN đã phát triển đến những phiên bản mới, như FORTRAN 95, FORTRAN 2000. Trong khuôn khổ quyển sách này chúng tôi chỉ hạn chế trình bày những kiến thức cơ bản của FORTRAN 90. Trong một số tình huống cụ thể, để giúp người đọc đã từng làm quen với FORTRAN 77 hoặc cần có thêm kiến thức để đọc những chương trình của người khác viết bằng FORTRAN 77, chúng tôi sẽ có thêm những ghi chú "mở rộng" thích hợp. Những người thành thạo Fortran muốn quan tâm đến lịch sử của ngôn ngữ đề nghị đọc cuốn *Fortran 90 Explained*, Oxford University Press (Oxford, 1990) của Michael Metcalf và John ReidMetcalf và Reid.

Như đã nói ở trên, chính xác hơn nên viết ngôn ngữ FORTRAN, nhưng do "sở thích tuỳ tiện", ở đây chúng tôi cũng sẽ Fortran thay cho cách viết FORTRAN.

Đây là bản thảo lần đầu của quyển sách, do đó chúng tôi chưa kịp đưa vào hệ thống các bài tập cũng như một số kiến thức nâng cao khác. Tất cả những vấn đề đó sẽ được cập nhật trong các bản thảo tiếp theo.

CHƯƠNG 1. NHỮNG YẾU TỐ CƠ BẢN CỦA NGÔN NGỮ FORTRAN

1.1 Chạy một chương trình FORTRAN

Nếu bạn là người mới làm quen với Fortran, bạn nên chạy các chương trình ví dụ trong phần này càng sớm càng tốt, không cần cố gắng hiểu một cách chi tiết chúng làm việc như thế nào. Việc giải thích sẽ được giới thiệu dần dần ở các phần sau. Để chạy được các chương trình này trước hết bạn cần phải có một bộ phần mềm biên dịch đã được cài đặt trên hệ thống máy của bạn. Ngoài ra bạn cũng cần phải làm quen với bộ phần mềm này, và cần phải biết cách đưa các chương trình Fortran vào và chạy nó như thế nào.

Chương trình sau đây sẽ đưa ra lời chào mừng bạn nếu bạn đưa tên của bạn vào khi được hỏi:

```
! Vi du 1.
! Loi Chao mung!
CHARACTER NAME*20
PRINT*, 'Ten ban la gi?'
READ*, NAME
PRINT*, 'Xin chao ban ', NAME
END
```

Bạn sẽ nhận được kết quả sau đây (câu trả lời của bạn là dòng chữ in nghiêng):

Ten ban la gi?
Nam
Xin chao ban Nam

Tuy nhiên, với chương trình trên, nếu bạn gõ tên bạn đầy đủ cả *Họ* và *tên* và giữa các từ có dấu cách thì kết quả có thể hơi bất ngờ đấy.

Chương trình sau đây tính giá trị của hàm $A(t) = 174.6(t - 1981.2)^3$ khi bạn cho vào giá trị của biến t

```
PROGRAM TinhHam
! Tinh gia tri ham A(t)
INTEGER T         ! gia tri bien t
REAL  A           ! gia tri ham A(t)
PRINT*,'Cho gia tri cua bien t:'
READ*, T
A = 174.6 * (T - 1981.2) ** 3
PRINT*,'Gia tri ham A(t) khi t=', T, ' la :', A
END PROGRAM TinhHam
```

Nếu bạn đưa vào giá trị 2000 cho biến t bạn sẽ nhận được kết quả

```
Gia tri ham A(t) khi t = 2000 la :  1.1601688E+06
```

Kết quả được cho dưới dạng ký hiệu khoa học. E+06 có nghĩa là số trước đó nhân với 10 luỹ thừa 6, như vậy trị số của A(t) vào khoảng 1.16 triệu. Bạn hãy chạy chương trình nhiều lần, mỗi lần thay đổi giá trị của t và thử tìm xem khi nào thì giá trị của hàm A(t) sẽ đạt khoảng 10 triệu.

Thử gõ nhầm giá trị của t (ví dụ 2,000) để xem Fortran phản ứng lại như thế nào.

Một ví dụ khác, giả sử bạn có 1000 đôla gửi tiết kiệm trong ngân hàng với lãi suất 9% mỗi năm. Hỏi sau một năm số tiền của bạn bằng bao nhiêu?

Để lập trình cho máy tính giải bài toán này trước hết bạn phải làm rõ vấn đề về mặt nguyen tắc. Bạn hãy hình dung lôgic của vấn đề như sau:

1) Nhập số liệu vào máy (số tiền ban đầu và lãi suất)

2) Tính tiền lãi (tức 9% của 1000, bằng 90)

3) Cộng tiền lãi vào số tiền ban đầu (90+1000, tức 1090)

4) In (hiển thị) số tiền bạn có sau một năm.

Sau đây là chương trình:

```
PROGRAM TinhTien
! Tinh tien gui tiet kiem
REAL SoTien, TienLai, LaiSuat
SoTien = 1000
LaiSuat = 0.09
TienLai = LaiSuat * SoTien
SoTien = SoTien + TienLai
PRINT*, 'So tien se co sau mot nam:', SoTien
END PROGRAM TinhTien
```

Bạn hãy gõ chương trình vào máy và chạy tính, và chú ý rằng ở đây máy không đòi hỏi bạn phải nhập đầu vào (input) từ bàn phím (Tại sao lại không?). Kết quả bạn nhận được sẽ là 1.0900000E+03 (1090).

Bạn hãy làm lặp lại nhiều lần các ví dụ trên đây, mỗi lần như vậy hãy thử sửa đổi một ít trong chương trình và theo dõi xem kết quả thay đổi như thế nào. Điều đó sẽ rất có ích, vì nó giúp cho bạn tự tin hơn khi tiếp cận với những nội dung sau này của Fortran.

Khi bạn chạy chương trình Fortran sẽ có hai quá trình tách biệt xảy ra. Đầu tiên chương trình được biên dịch (compile), tức là mỗi câu lệnh được dịch (translated) sang mã máy (*machine code)* sao cho máy tính có thể hiểu được. Quá trình này xảy ra như sau. Trước hết các câu lệnh của chương trình sẽ được kiểm tra về cú pháp (Syntax). Nếu không có lỗi, chúng sẽ được dịch sang mã máy và lưu trữ vào một file gọi là *đối tượng* (Object) hay *đích*. Sau đó chúng sẽ được *liên kết* (Link) với hệ thống thư viện chuẩn của Fortran để tạo thành file có thể *thực hiện* (executable) được. Nếu chương trình còn lỗi, các lỗi sẽ được chỉ ra và quá trình biên dịch kết thúc mà không tạo được file *đích*.

Thứ hai, chương trình đã dịch (tức file có thể thực hiện được) sẽ được thực hiện (*executed*). Ở bước này mỗi một chỉ thị đã dịch của chương trình sẽ lần lượt được thực hiện theo qui tắc đã lập.

Bộ chương trình thực hiện cả hai quá trình này thường được gọi là trình biên dịch (*compiler).*

Trong khi biên dịch, không gian bộ nhớ RAM của máy tính định vị cho mọi dữ liệu sẽ được phát sinh bởi chương trình. Phần bộ nhớ này có thể hiểu như là những "vùng" bộ nhớ khu trú mà mỗi một trong chúng, tại một thời điểm chỉ, chỉ có thể xác định một giá trị dữ liệu. Các bộ nhớ khu trú này được tham chiếu đến bởi các tên ký hiệu (định danh) trong chương trình. Bởi vậy câu lệnh:

```
SoTien = 1000
```

là cấp phát số 1000 đến vị trí bộ nhớ có tên `SoTien`. Vì nội dung của `SoTien` có thể thay đổi trong khi chương trình chạy nên nó được gọi là *biến* (*variable)*.

Về hình thức, chương trình tính tiền gửi tiết kiệm trên đây được biên dịch như sau:

1) Đưa số 1000 vào vị trí bộ nhớ `SoTien`
2) Đưa số 0.09 vào vị trí bộ nhớ `LaiSuat`
3) Nhân nội dung của `LaiSuat` với nội dung của `SoTien` và đưa kết quả vào vị trí bộ nhớ `TienLai`
4) Cộng nội dung của `SoTien` với nội dung của `LaiSuat` và đưa kết quả vào `SoTien`
5) In (hiển thị) thông báo với nội dung của `SoTien`
6) Kết thúc.

Khi thực hiện, các câu lệnh dịch này được thực hiện theo thứ tự từ trên xuống dưới. Quá trình thực hiện, các vị trí bộ nhớ được sử dụng sẽ có các giá trị sau:

```
SoTien : 1000
LaiSuat: 0.09
TienLai: 90
SoTien : 1090
```

Chú ý rằng nội dung ban đầu của `SoTien` đã bị thay thế bởi giá trị mới.

Câu lệnh `PROGRAM` ở dòng đầu tiên mở đầu cho chương trình. Nó là câu lệnh tuỳ chọn, và có thể kèm theo tên tuỳ ý. Dòng thứ hai, bắt đầu với dấu chấm than, là lời giải thích, có lợi cho người đọc chương trình, và không ảnh hưởng gì tới chương trình dịch. Các biến trong chương trình có thể khác kiểu (*type)*; câu lệnh `REAL` trong ví dụ này là khai báo kiểu. Các dòng trống (nếu có) trong chương trình là không thực hiện (*non-executable)*, tức là không có tác động nào được thực hiện.

Bây giờ bạn hãy thử làm lại bài tập này như sau.

1) Chạy chương trình và ghi nhớ lại kết quả

2) Thay đổi câu lệnh thực hiện `SoTien = 1000` bởi câu lệnh `SoTien = 2000` và chạy lại chương trình. Chắc chắn rằng bạn đã hiểu điều gì xảy ra khi bạn nhận được kết quả mới.

3) Loại bỏ dòng lệnh

```
SoTien = SoTien + TienLai
```

và chạy lại chương trình. Bạn có thể giải thích điều gì xảy ra không?

1.2 Cấu trúc chung của một chương trình FORTRAN

Cấu trúc chung của một chương trình Fortran đơn giản như sau (những phần đặt trong dấu ngoặc vuông là tuỳ ý, có thể có, cũng có thể không):

```
[PROGRAM TenChuongTrinh]
      [Cac cau lenh khai bao]
      [Cac cau lenh thuc hien]
END [PROGRAM [TenChuongTrinh]]
```

Như đã thấy, chỉ có một câu lệnh bắt buộc trong chương trình Fortran là `END`. Câu lệnh này báo cho chương trình dịch rằng không còn câu lệnh nào hơn nưa để dịch.

Ký hiệu

```
END [PROGRAM [TenChuongTrinh]]
```

có nghĩa rằng *`TenChuongTrinh`* có thể bỏ qua trong câu lệnh END, nhưng nếu có *`TenChuongTrinh`* thì từ khoá PROGRAM là bắt buộc.

`TenChuongTrinh` là tên của chương trình, thường được đặt một cách tùy ý sao cho mang tính gợi nhớ rằng chương trình sẽ giải quyết vấn đề gì. *`Cac cau lenh khai bao`* là những câu lệnh khai báo biến, hằng,... để trình biên dịch cấp phát bộ nhớ, phân luồng xử lý. *`Cac cau lenh thuc hien`* là những câu lệnh xác định qui tắc và trình tự thực hiện tính toán, xử lý để đạt được kết quả.

1.3 Cấu trúc câu lệnh

Dạng câu lệnh cơ bản của mọi chương trình Fortran có thể gồm từ 0 đến 132 ký tự (câu lệnh có thể là trống rỗng; câu lệnh trống rỗng làm cho chương trình dễ đọc hơn bởi sự phân cách lôgic giữa các đoạn). Đối với phiên bản Fortran 77 và các phiên bản trước đó, nội dung các câu lệnh phải bắt đầu từ cột thứ 7 và kéo dài tối đa đến cột thứ 72. Nếu câu lệnh có nội dung dài hơn, nó sẽ được ngắt xuống dòng dưới, và ở dòng nối tiếp này phải có một ký tự bất kỳ (khác dấu cách) xuất hiện ở cột thứ 6. Bạn đọc cần lưu ý đặc điểm này khi sử dụng các chương trình của người khác hoặc lập trình với các phiên bản Fortran 77 và trước đó. Fortran 90 không có sự hạn chế đó.

Tất cả các câu lệnh, trừ câu lệnh gán (ví dụ `Sotien = 1000`), đều bắt đầu bằng các từ khoá (*keyword)*. Trên đây chúng ta đã gặp một số từ khoá như `END`, `PRINT`, `PROGRAM`, và `REAL`.

Nói chung trên mỗi dòng có một câu lệnh. Tuy nhiên, nhiều câu lệnh cũng có thể xuất hiện trên một dòng, nhưng chúng phải được phân cách nhau bởi dấu chấm phẩy (;). Để cho rõ ràng chỉ nên viết những câu lệnh gán rất ngắn, như:

```
A = 1; B = 1; C = 1
```

Những câu lệnh dài có thể viết trên nhiều dòng.

1.3.1 Ý nghĩa của dấu cách (Blank)

Nói chung các dấu cách là không quan trọng, bạn có thể sử dụng chúng để làm cho chương trình dễ đọc hơn bằng cách viết thụt câu lệnh vào (thêm dấu cách vào phía bên trái) hoặc đệm vào giữa các câu lệnh. Tuy nhiên, cũng có những chỗ bạn không được phép chèn dấu cách vào, như các qui ước về cách viết từ khóa, tên biến,... mà ta gọi là các ký hiệu qui ước.

Ký hiệu qui ước trong Fortran 90 là một chuỗi liên tiếp các ký tự có ý nghĩa, chẳng hạn các *nhãn*, các *từ khóa*, *tên*, *hằng*,... Như vậy, các cách viết `INTE GER`, `So Tien` và `< =` là không được phép (`<=` là một phép toán), trong khi `A * B` thì được phép và giống như `A*B`.

Tuy nhiên, tên, hằng hoặc nhãn cần phải được phân cách với các từ khoá, tên, hằng hoặc nhãn khác ít nhất một dấu cách. Như vậy `REALX` và `30CONTINUE` là không được phép (`30` là nhãn).

1.3.2 Lời chú thích

Mọi ký tự theo sau dấu chấm than (!) (ngoại trừ trong chuỗi ký tự) là lời chú thích, và được bỏ qua bởi chương trình dịch. Toàn bộ nội dung trên cùng một dòng có thể là lời chú thích. Dòng trắng cũng được dịch như là dòng chú thích. Lời chú thích có thể được dùng một cách tuỳ ý để làm chương trình dễ đọc.

Đối với Fortran 77, nếu cột đầu tiên có ký tự 'C' hoặc 'c' thì nội dung chứa trên dòng đó sẽ được hiểu là lời chú thích. Qui tắc này không được Fortran 90 chấp nhận. Nhưng thay cho các ký tự 'C' hoặc 'c', nếu sử dụng ký tự dấu chấm than thì chúng lại tương đương nhau.

1.3.3 Dòng nối tiếp

Nếu câu lệnh quá dài nó có thể được chuyển một phần xuống dòng tiếp theo bằng cách thêm ký hiệu (&) vào cuối cùng của dòng trước khi ngắt phần còn lại xuống dòng dưới. Ví dụ:

```
A = 174.6 *    &
    (T - 1981.2) ** 3
```

Như đã nói ở trên, Fortran 77 sử dụng cột thứ 6 làm cột nối dòng, do đó cách chuyển tiếp dòng của Fortran 90 sẽ không tương thích với Fortran 77.

Dấu & tại cuối của dòng chú thích sẽ không được hiểu là sự nối tiếp của dòng chú thích, vì & được xem như là một phần của chú thích.

1.4 Kiểu dữ kiệu

Khái niệm kiểu dữ liệu (*data type)* là khái niệm cơ bản trong Fortran 90. Kiểu dữ liệu bao gồm tập hợp các giá trị dữ liệu (chẳng hạn, toàn bộ các số), cách thức biểu thị chúng (ví dụ –2, 0, 999), và tập hợp các phép toán (ví dụ phép toán số học) cho phép xuất hiện trong chúng.

Fortran 90 định nghĩa 5 kiểu dữ liệu chuẩn, được chia thành hai lớp là lớp các kiểu số (*numeric)* gồm số nguyên (integer), số thực (real) và số phức (complex), và lớp các kiểu không phải số (*non-numeric)* gồm kiểu ký tự (character) và kiểu lôgic (logical).

Liên kết với mỗi kiểu dữ liệu là các loại dữ liệu khác nhau. Về cơ bản điều đó liên quan đến khả năng lưu trữ. Chẳng hạn, có thể có hai loại số nguyên (integer): số nguyên ngắn và số nguyên dài. Chúng ta sẽ đề cập đến vấn đề này sâu hơn ở các phần sau.

Ngoài các kiểu dữ liệu chuẩn trên đây, bạn có thể định nghĩa cho riêng mình *các kiểu dữ liệu khác*, chúng có thể có các tập giá trị và các phép toán riêng.

1.4.1 Lớp các kiểu số (Integer, Real, Complex)

a. Kiểu số nguyên

Dữ liệu có kiểu số nguyên là những dữ liệu nhận các giá trị thuộc tập số nguyên, ví dụ 0, 1, 2, 3,..., –5, –10,... Để khai báo biến hoặc hằng có kiểu số nguyên ta sử dụng câu lệnh:

`INTEGER [([ KIND=]`*`kind-value`*`)][,`*`attrs`*`] ::] `*`vname`*

Trong đó:

kind–value là loại, nhận các giá trị 1, 2, 4

attrs là thuộc tính, nhận một hoặc nhiều hơn trong các giá trị PARAMETER, DIMENSION, ALLOCATABLE, POINTER,…

vname là danh sách biến hoặc hằng.

Tùy theo loại mà một biến/hằng nguyên sẽ chiếm dung lượng bộ nhớ và phạm vi là lớn hay nhỏ. Bảng sau đây chỉ ra miền giá trị hợp lệ đối với các loại số nguyên được khai báo.

Cách khai báo	Số byte chiếm giữ	Phạm vi giá trị
INTEGER	4	–2,147,483,648 đến 2,147,483,647
INTEGER*1 hoặc INTEGER (1) hoặc INTEGER (KIND=1)	1	–128 đến 127
INTEGER*2 hoặc INTEGER (2) hoặc INTEGER (KIND=2)	2	–32,768 đến 32,767
INTEGER*4 hoặc INTEGER (4) hoặc INTEGER (KIND=4)	4	–2,147,483,648 đến 2,147,483,647

Các ví dụ sau đây cho thấy có thể sử dụng các cách khác nhau để khai báo kiểu số nguyên cho các biến, hằng.

```
INTEGER, DIMENSION(:), POINTER :: days, hours
INTEGER(2), POINTER :: k, limit
INTEGER(1), DIMENSION(10) :: min
```

hoặc

```
INTEGER days, hours
INTEGER(2) k, limit
INTEGER(1) min
DIMENSION days(:), hours(:), min (10)
POINTER days, hours, k, limit
```

hoặc

```
INTEGER (2) :: k=4
INTEGER (2), PARAMETER :: limit=12
```

hoặc

```
INTEGER days, hours
INTEGER (2):: k=4, limit
DIMENSION days(:), hours(:)
POINTER days, hours
PARAMETER (limit=12)
```

b. Kiểu số thực

Kiểu số thực nói chung gần giống với tập số thực trong toán học. Để biểu diễn số thực Fortran sử dụng hai phương pháp gần đúng là độ chính xác đơn và độ chính xác kép. Có thể khai báo kiểu số thực bằng các câu lệnh sau.

```
REAL [([KIND=]kind-value)][[,attrs] ::] vname
```

hoặc

```
DOUBLE PRECISION [[,attrs] ::] vname
```

Trong đó:

kind-value là loại, nhận giá trị 4 hoặc 8

attrs là thuộc tính, nhận một hoặc nhiều hơn các giá trị PARAMETER, DIMENSION, ALLOCATABLE, POINTER,…

vname là danh sách biến hoặc hằng.

Cách khai báo, phạm vi giá trị, độ chính xác và dung lượng bộ nhớ bị chiếm ứng với từng loại số thực được cho trong bảng sau.

Cách khai báo	Số byte chiếm giữ	Độ chính xác (số chữ số)	Phạm vi giá trị
REAL REAL*4 REAL (KIND=4)	4	6	–3.4028235E+38 đến –1.1754944E–38; 0; +1.1754944E–38 đến +3.4028235E+38
REAL*8 REAL (KIND=8) DOUBLE PRECISION	8	15	–1.797693134862316D+308 đến –2.225073858507201D–308; 0; +2.225073858507201D–308 đến +1.797693134862316D+308

Sau đây là một số ví dụ khai báo các biến, hằng có kiểu số thực.

```
REAL X, Y(10)
REAL*4 A,B
REAL (KIND=8), DIMENSION (5) :: U,V
DOUBLE PRECISION, DIMENSION (:), ALLOCATABLE :: T
REAL, PARAMETER :: R_TDat = 6370.0
```

c. Kiểu số phức

Số phức được định nghĩa như một cặp có thứ tự của hai số thực được gọi là phần thực và phần ảo. Dữ liệu kiểu số phức được khai báo bằng câu lệnh:

`COMPLEX [([KIND =]`*`kind-value`*`)] [[,`*`attrs`*`] :: ]` *vname*

Trong đó

kind-value nhận giá trị 4 hoặc 8;

attrs là một hoặc nhiều thuộc tính, nhận các giá trị PARAMETER, DIMENSION, ALLOCATABLE, POINTER,…;

vname là danh sách biến hoặc hằng.

Độ chính xác và phạm vị giá trị của kiểu số phức là độ chính xác và phạm vi giá trị của các phần thực và phần ảo. Nhưng dung lượng bộ nhớ chiếm giữ của một số phức là dung lượng của hai số thực. Bảng sau đây liệt kê các cách khai báo và số byte chiếm giữ của các biến, hằng có kiểu số phức.

Cách khai báo	Số byte chiếm giữ
COMPLEX COMPLEX *4 COMPLEX (4) COMPLEX (KIND=4)	8
COMPLEX *8 COMPLEX (8) COMPLEX (KIND=8) DOUBLE CPMPLEX	16

1.4.2 Kiểu ký tự (Character) và kiểu lôgic (Logical)

a. Kiểu ký tự

Kiểu ký tự có tập giá trị là các ký tự lập thành xâu ký tự. Độ dài của xâu là số ký tự trong xâu đã được khai báo. Mỗi ký tự trong xâu ký tự chiếm 1 byte bộ nhớ. Do đó, số byte chiếm giữ bộ nhớ của biến, hằng kiểu ký tự là tùy thuộc độ dài của xâu. Câu lệnh tổng quát khai báo một biến có kiểu ký tự như sau.

```
CHARACTER (length-selector) vname
```

hoặc

```
CHARACTER (type-param [ , type-param . . .] ) [
attrib-spec [ , attrib-spec] ...] :: vname
```

hoặc

```
CHARACTER [*chars] vname [*length][(dim)]
[/values/][, vname [*length][(dim)] ] [/values/]
```

Trong đó:

length-selector: Độ dài cực đại của *vname*, có thể có các dạng: **n*, (*n*), hoặc (*), với *n* là một số nguyên dương chỉ độ dài của xâu.

vname: Danh sách tên biến, hằng có kiểu xâu ký tự, viết cách nhau bởi dấu phẩy (,).

type–param: là tham số độ dài và loại, nhận một trong các dạng:

- (**LEN** = *type-param-value*)
- (**KIND** = *expr*)
- (**KIND** = *expr*, **LEN** = *type-param-value*)
- ([**LEN** =] *type-param-value*, **KIND** = *expr*)

type–param–value: có thể là dấu sao (*), hằng nguyên không dấu, hoặc biểu thức nguyên.

expr: là biểu thức xác định giá trị hằng nguyên tương ứng với phương pháp biểu diễn ký tự (chẳng hạn, chữ cái Latinh, chữ cái Hylạp,…).

attrib–spec: là một hoặc nhiều thuộc tính, cách nhau bởi dấu phẩy (,). Nếu bạn chỉ ra thuộc tính thì sau đó phải sử dụng dấu (::). Các thuộc tính có thể là: ALLOCATABLE, DIMENSION, PARAMETER, POINTER,…

chars: Độ dài (cực đại) của các xâu, có thể là một số nguyên không dấu, biểu thức nguyên nằm trong ngoặc đơn, hoặc dấu sao nằm trong ngoặc đơn (*).

length : Độ dài (cực đại) của xâu, có thể là số nguyên không dấu, biểu thức nguyên nằm trong ngoặc đơn, hoặc dấu sao nằm trong ngoặc đơn (*).

dim: Khai báo mảng, tức *vname* như là mảng.

/values/: Liệt kê các hằng ký tự, là giá trị của các biến hằng *vname*.

Ví dụ:

```
CHARACTER (20) St1, St2*30
CHARACTER  wt*10, city*80, ch
CHARACTER (LEN = 10), PRIVATE :: vs
CHARACTER*(*) arg
CHARACTER  name(10)*20  ! Mảng
CHARACTER(len=20), dimension(10):: plume ! Mảng
CHARACTER(2) susan, patty, alice*12, dotty, jane(79)
CHARACTER*5 word /'start'/
```

b. Kiểu lôgic

Dữ liệu kiểu lôgic chỉ nhận các giá trị .TRUE. hoặc .FALSE. Câu lệnh khai báo kiểu dữ liệu lôgic có dạng:

LOGICAL [([KIND =]*kind*)] [, *attrs*] * ::] *vname*

Trong đó:

kind: là độ dài tính bằng byte, nhận các giá trị 1, 2, hoặc 4.

attrs: là các thuộc tính, có thể một hoặc nhiều thuộc tính, phân cách nhau bởi dấu phẩy (,).

vname: Danh sách các biến, hằng, phân cách nhau bởi dấu phẩy.

Số byte chiếm giữ bộ nhớ của kiểu dữ liệu logic phụ thuộc vào loại dữ liệu như mô tả trong bảng dưới đây.

Cách khai báo	Loại (KIND)	Số byte chiếm giữ
LOGICAL	4	4
LOGICAL*1 hoặc LOGICAL (1) hoặc LOGICAL (KIND=1)	1	1
LOGICAL*2 hoặc LOGICAL (2) hoặc LOGICAL (KIND=2)	2	4
LOGICAL*4 hoặc LOGICAL (4) hoặc LOGICAL (KIND=4)	4	4

Ví dụ, các câu lệnh khai báo sau đây

```
LOGICAL, ALLOCATABLE :: flag1, flag2
LOGICAL (2), SAVE :: doit, dont=.FALSE.
LOGICAL  switch
```

tương đương với các câu lệnh

```
LOGICAL flag1, flag2
LOGICAL (2) doit, dont=.FALSE.
ALLOCATABLE  flag1, flag2
SAVE doit, dont
```

1.4.3 Phép toán trên các kiểu dữ liệu

Fortran định nghĩa các lớp phép toán sau:

– Phép toán số học: Sử dụng với các số nguyên, số thực và số phức.

– Phép toán quan hệ hay phép toán so sánh: Sử dụng với số nguyên, số thực, kiểu ký tự; cũng có thể đối với cả số phức trong trường hợp so sánh *bằng* hoặc *không bằng*.

– Phép toán lôgic: Sử dụng đối với kiểu lôgic, và có thể với cả số nguyên.

– Phép toán gộp xâu ký tự: Sử dụng vơi kiểu ký tự.

Bảng sau đây liệt kê ký hiệu các phép toán, thứ tự ưu tiên, thứ tự thực hiện trong biểu thức và ý nghĩa của chúng.

Ký hiệu phép toán	Tên gọi/Ý nghĩa	Thứ tự ưu tiên	Thứ tự thực hiện	Ví dụ
Phép toán số học				
**	Phép lũy thừa	1	Phải sang trái	A ** B
*	Phép nhân	2	Trái sang phải	A * B
/	Phép chia	2	Trái sang phải	A / B
+	Phép cộng	3	Trái sang phải	A + B
–	Phép trừ	3	Trái sang phải	A – B
Phép toán quan hệ				
.EQ. (==)	Bằng	–	Không phân định	A.EQ.B; A == B
.LT. (<)	Nhỏ hơn	–	Không phân định	A.LT.B; A < B
.LE. (<=)	Nhỏ hơn hoặc bằng	–	Không phân định	A.LE.B; A <= B
.GT. (>)	Lớn hơn	–	Không phân định	A.GT.B; A > B
.GE. (>=)	Lớn hơn hoặc bằng	–	Không phân định	A.GE.B; A >= B
.NE. (/=)	Không bằng (Khác)	–	Không phân định	A.NE.B; A /= B
Phép toán lôgic				
.NOT.	Phủ định	1	Không phân định	.NOT. L1
.AND.	Và (Phép hội)	2	Trái sang phải	L1. AND. L2
.OR.	Hoặc (Phép tuyển)	3	Trái sang phải	L1. OR. L2
.XOR.	Hoặc triệt tiêu	4	Trái sang phải	L1. XOR. L2
.EQV.	Tương đương	4	Trái sang phải	L1. EQV. L2
.NEQV.	Không tương đương	4	Trái sang phải	L1. NEQV. L2
Gộp ký tự				
//	Gộp hai xâu ký tự	–	Trái sang phải	ST1 // ST2

Chú ý:

– Trong một biểu thức số học, nếu các toán hạng có cùng kiểu dữ liệu thì kiểu dữ liệu kết quả là kiểu dữ liệu của các toán hạng. Nếu các toán hạng khác kiểu dữ liệu thì kết quả nhận được sẽ có kiểu dữ liệu của toán hạng có kiểu "mạnh nhất". Chẳng hạn, biểu thức gồm hỗn hợp cả số nguyên và số thực thì kết quả sẽ có kiểu số thực, vì kiểu số thực "mạnh hơn" số nguyên. Tuy nhiên, khi kết quả đó gán cho một biến thì kiểu của kết quả sẽ được chuyển thành kiểu dữ liệu của biến. Ví dụ, nếu a, b, x là các biến thực, còn n là biến nguyên thì:

a= –22.9; b=6.1 => x=a+b= –16.8; nhưng n=a+b = –16

a= 2.9; b=6.8 => x=a+b= 9.7; nhng n=a+b = 9

– Kết quả của biểu thức quan hệ và biểu thức lôgic luôn luôn nhận hoặc .TRUE. hoặc .FALSE.

Ví dụ:

2 ** 9 ** 0.5 cho kết quả là 8

10 + 3 * 2 ** 4 – 16 / 2 cho kết quả là 50

3.5 > 7.2 cho kết quả là .FALSE.

Nếu a và b là hai biến lôgic, khi đó các phép toán giữa a và b sẽ cho kết quả:

	a .AND. b	**a .OR. b**	**a .EQV. b**	**a .NEQV. b**	**a.XOR.b**
a=.TRUE., b=.TRUE.	.TRUE.	.TRUE.	.TRUE.	.FALSE.	.FALSE.
a=.TRUE., b=.FALSE.	.FALSE.	.TRUE.	.FALSE.	.TRUE.	.TRUE.
a=.FALSE., b=.TRUE.	.FALSE.	.TRUE.	.FALSE.	.TRUE.	.TRUE.
a=.FALSE., b=.FALSE.	.FALSE.	.FALSE.	.TRUE.	.FALSE.	.FALSE.

Nếu ST1 và ST2 là hai xâu ký tự, với:

```
ST1='Hanoi'
ST2=' - Vietnam'
```

thì

ST1 // ST2 sẽ cho kết quả là `'Hanoi - Vietnam'`

1.5 Hằng

Hằng là những ký hiệu qui ước được sử dụng để biểu thị các giá trị có kiểu riêng, tức là những ký tự thực tế mà chúng có thể được sử dụng.

1.5.1 Hằng nguyên

Hằng nguyên được sử dụng để biểu thị các giá trị kiểu số nguyên thực sự. Biểu diễn đơn giản nhất và rõ ràng nhất là số nguyên không dấu hoặc có dấu, ví dụ:

1000, 0, +753, –999999, 2501

Trong trường hợp hằng nguyên dương, thì dấu là không bắt buộc (tuỳ ý).

Toàn bộ các số dương cũng có thể được biểu diễn dưới dạng binary (nhị phân – cơ số 2), octal (cơ số 8) hoặc hexa (cơ số 16), tức:

```
binary:      B'1011'
octal:       O'0767'
hexadecimal:   Z'12EF'
```

Có thể sử dụng chữ in thường hoặc chữ in hoa. Dấu nháy kép (") có thể được sử dụng thay cho dấu nháy đơn (') như là sự phân ranh giới. Các dạng thức này không được dùng với câu lệnh DATA.

1.5.2 Hằng thực

Hằng thực dùng để biểu thị các giá trị có kiểu số thực và có hai dạng.

– Dạng thứ nhất được viết rất rõ ràng, gọi là *dạng dấu phẩy tĩnh*, bao gồm một dãy số có chứa dấu chấm thập phân. Nó có thể có dấu hoặc không dấu. Ví dụ:

```
0.09   37.   37.0   .0   -.6829135
```

Có thể không có số nào phía bên trái hoặc phía bên phải dấu chấm thập phân, nhưng chỉ có một dấu chấm thập phân không thôi thì không được phép.

– Dạng thứ hai được gọi là dạng *dấu phẩy động*. Về cơ bản nó bao gồm hoặc một số nguyên hoặc một số thực dấu phẩy tĩnh (có thể có dấu hoặc không) và sau đó là chữ cái E (hoặc e), tiếp theo là số nguyên (có dấu hoặc không). Số nguyên đứng đằng sau E là chỉ số mũ của 10, nó chỉ ra rằng đó là 10 luỹ thừa mà số đằng trước E phải nhân với nó. Ví dụ:

2.0E2 ($2. \times 10^2 = 200.0$)

2E2 ($2 \times 10^2 = 200.0$)

4.12E+2 ($4.12 \times 10^{+2} = 412.0$)

-7.321E-4 ($-7.321 \times 10^{-4} = -0.0007321$)

Hằng thực được lưu trữ dưới dạng luỹ thừa trong bộ nhớ, không quan trọng chúng được viết thực sự như thế nào. Bởi vậy, nếu số thực có thể được biểu diễn dưới dạng phân số thì chúng cũng sẽ được biểu diễn gần đúng. Thậm chí giữa số nguyên và số thực có cùng giá trị, chúng cũng được biểu diễn khác nhau. Ví vụ 43 là số nguyên, trong khi 43.0 là số thực. Chúng sẽ được biểu diễn khác nhau trong bộ nhớ.

Phạm vi và độ chính xác của hằng thực không được chỉ ra một cách chuẩn xác, nhưng độ chính xác khoảng 6–7 chữ số thập phân.

1.5.3 Hằng ký tự

Hằng ký tự là một chuỗi các ký tự nằm trong cặp dấu nháy đơn (‘ ’) hoặc nháy kép (“ ”). Các ký tự không phải là ký tự điều khiển (chẳng hạn, #27 là ESC) thuộc bảng mã ký tự ASCII đều có thể được sử dụng để biểu diễn hằng ký tự. Bởi vì mỗi ký tự trong bảng mã ASCII tương ứng với số thứ tự duy nhất của nó, nên các giá trị của hằng ký tự có sự phân biệt giữa chữ thường và chữ hoa. Ví dụ:

“HANOI” khác với “Hanoi”, hoặc “Hai Phòng” khác với “Hai phong”,…

1.6 Tên biến và tên hằng

Chúng ta đã thấy rằng vị trị bộ nhớ có thể được cho bởi tên tương trưng (symbolic *names*), gọi là tên biến hoặc tên hằng, như *SoTien* và *LaiSuat*. Tên biến, tên hằng có thể gồm 1 đến 31 ký tự, và phải bắt đầu bởi một chữ cái tiếng Anh. Các ký tự được sử dụng để cấu tạo tên gồm 26 chữ cái tiếng Anh (A–Z hoặc a–z), 10 chữ số (0–9), và *dấu gạch dưới* (_).

Ngoại trừ xâu ký tự, Fortran không phân biệt tên viết bằng chữ thường hay chữ hoa, ví dụ `MYNAME` và `MyName` là như nhau. Có lẽ những người có truyền thống lập trình Fortran lâu năm thường viết chương trình chỉ bằng chữ cái in hoa. Tuy nhiên ta nên viết lẫn cả chữ thường và chữ hoa cho dễ đọc. Chẳng hạn, nếu ta viết `SoTien` chắc chắn sẽ dễ đọc hơn là viết `SOTIEN`. Mặt khác, vì Fortran 90, và cả các phiên bản mới hơn sau này, không khống chế độ dài tên chỉ 6 ký tự như các phiên bản cũ, nên để rõ ràng, tên viết dài có thể sẽ tốt hơn viết ngắn, vì nó mang tính gợi nhớ hơn. Chẳng hạn, nên viết `SoTien` thay cho cách viết đơn giản `ST`.

Bảng sau đây chỉ ra các tên biến, tên hằng hợp lệ và không hợp lệ:

Tên hợp lệ	Tên không hợp lệ
`X`	`X+Y` (vì chứa dấu + là một phép toán)
`R2D2`	`SHADOW FAX` (vì chứa dấu cách)
`Pay_Day`	`2A` (vì ký tự đầu tiên là chữ số)
`ENDOFTHEMONTH`	`OBI-WAN` (vì chứa dấu – là một phép toán)

Biến là vị trí bộ nhớ mà giá trị của nó có thể bị thay đổi trong quá trình thực hiện chương trình. Tên của biến được cấu tạo theo qui tắc trên đây. Biến có kiểu và loại dữ liệu xác định được cho bởi khai báo kiểu, ví dụ:

```
INTEGER X
REAL LaiSuat
CHARACTER LETTER
REAL :: A = 1
```

Chú ý rằng, biến có thể được khởi tạo khi khai báo nó. Trong trường hợp này phải sử dụng dấu hai chấm kép (::). Giá trị của biến được khởi tạo theo cách này có thể bị thay đổi trong quá trình chương trình thực hiện.

Mặc dù các biến `X`, `LaiSuat` và `LETTER` đã được khai báo trong đoạn chương trình trên, nhưng giá trị của vẫn chưa được xác định. Bạn (đặc biệt là những người mới bắt đầu lập trình) phải chú ý tránh việc tham chiếu đến các biến chưa được xác định này, vì nó có thể sẽ dẫn đến lỗi trong lúc thực hiện chương trình (Run time error), rất khó gỡ rối. Biến có thể được xác định bằng nhiều cách, ví dụ bằng việc khởi tạo nó (như ở trên) hoặc bằng việc gán giá trị cho nó, như trong các ví dụ khác chúng ta đã thấy.

Biến cũng có thể được cho giá trị ban đầu bằng lệnh `DATA` sau khi đã khai báo, ví dụ:

```
REAL A, B
INTEGER I, J
DATA A, B / 1, 2 / I, J / 0, -1/
```

Tên trong chương trình phải là duy nhất. Chẳng hạn, nếu chương trình được đặt tên là `TinhTien`, thì việc khai báo một biến khác cùng tên sẽ dẫn đến lỗi.

Các biến đã mô tả trên đây gọi là những biến vô hướng, hay biến đơn, vì chúng chỉ quản lý một giá trị đơn nhất. Ngoài các biến vô hướng còn có các loại biến khác, chẳng hạn biến mảng.

1.7 Qui tắc kiểu ẩn

Các phiên bản trước của Fortran có một qui tắc đặt tên ngầm định được gọi là qui tắc kiểu ẩn (*implicit type rule*). Theo qui tắc này, các biến bắt đầu bằng các chữ cái I, J, K, L, M, N được tự động hiểu là biến có kiểu số nguyên (Integer), còn các biến bắt đầu bằng những chữ cái khác được hiểu là biến thực (real). Để bảo đảm tính tương thích của các chương trình viết với các phiên bản trước, Qui tắc này vẫn được áp dụng ngầm định trong Fortran 90.

Tuy nhiên, trong một số tình huống, qui tắc kiểu ẩn có thể dẫn đến lỗi chương trình trầm trọng. Giá trị thực có thể được gán một cách không cố ý cho biến nguyên, làm cho phần thập phân sau dấu chấm thập phân sẽ bị chặt cụt. Ví dụ, nếu không khai báo kiểu cho biến `LaiSuat` thì câu lệnh

```
LaiSuat = 0.12
```

trong chương trình, biến `LaiSuat` sẽ được gán giá trị 0.

Để đề phòng những lỗi như vậy, ngay từ đầu chương trình ta **nên đưa vào** câu lệnh sau

```
IMPLICIT NONE
```

Câu lệnh này sẽ xoá bỏ thuộc tính qui tắc kiểu ẩn, do đó tất cả các biến sử dụng trong chương trình bắt buộc phải được khai báo. Đó là cách lập trình tốt, vì có khai báo ý nghĩa của biến bạn buộc phải để tâm đến nó.

Ví dụ:

Nếu một hòn đá được tung lên thẳng đứng với tốc độ ban đầu u, quãng đường dịch chuyển thẳng đứng s của nó sau thời gian t được cho bởi công thức

$s(t) = ut - \frac{gt^2}{2}$, trong đó g là gia tốc trọng trường. Bỏ qua lực cản của không khí. Hãy tính giá trị của s, khi cho các giá trị của u và t.

Để lập chương trình giải bài toán này ta có thể hình dung lôgic chuẩn bị chương trình như sau:

1) Nhập các giá trị g, u và t vào máy tính

2) Tính giá trị của s theo công thức đã cho

3) In giá trị của s

4) Kết thúc

Dàn bài này có vẻ tầm thường đối với bạn, thậm chí bạn có thể cho rằng nó lãng phí thời gian viết ra. Tuy thế, bạn sẽ bị ngạc nhiên tại sao nhiều người mới bắt đầu lập trình lại thích làm trực tiếp trên máy tính, và lập trình bước 2 trước bước 1, để rồi lúng túng trước kết quả nhận được. Thực tế điều này rất quan trọng, vì nó tạo cho bạn một thói quen phân tích bài toán một cách kỹ lưỡng, thiết kế chương trình có tính lôgic, và chọn tên biến, kiểu biến để khai báo một cách phù hợp nhất.

Dựa theo các bước trên đây ta có chương trình như sau:

```
PROGRAM ChuyenDongThangDung
! Chuyen dong thang dung duoi luc trong truong
IMPLICIT NONE
REAL, PARAMETER :: G = 9.8 ! Gia toc trong truong
REAL S             ! Quang duong (m)
REAL T             ! Thoi gian
REAL U             ! Toc do ban dau (m/s)
PRINT*, ' Thoi gian    Quang duong'
PRINT*
U = 60
T = 6
S = U * T - G / 2 * T ** 2
PRINT*, T, S
END PROGRAM ChuyenDongThangDung
```

Trước hết khai báo G là hằng, vì giá trị của nó được xác định không thay đổi trong chương trình. Cách khai báo này có vẻ hơi mới lạ đối với bạn. Nhưng không sao, bạn hãy tạm chấp nhận nó. Trong chương trình có sử dụng câu lệnh `IMPLICIT NONE` do đó bạn phải khai báo tất cả các biến. Bạn hãy thử *bỏ qua một câu lệnh khai báo biến* nào đó bằng cách thêm dấu chấm than vào đầu dòng lệnh và chạy lại chương trình để xem Fortran phản ứng như thế nào.

1.8 Phong cách lập trình

Trên thực tế có thể xảy ra tình huống rằng, bạn cần sử dụng lại hoặc nâng cấp các chương trình mà bạn đã lập từ rất lâu rồi, hoặc các chương trình do một người nào đó viết. Sẽ rất khó khăn cho bạn nếu trong chương trình chẳng có một lời chú thích nào cả. Đối với những chương trình của bạn, có thể bạn đã quên đi những cái

mình đã viết. Việc tìm hiểu lại một chương trình không có những lời chú thích như vậy đôi khi làm cho bạn nản chí, không đủ kiên nhẫn để thực hiện.

Để tránh tình trạng đó, bạn phải có một phong cách lập trình tốt. Nghĩa là trong chương trình bạn phải có những lời chú thích đúng chỗ, đầy đủ, rõ ràng; trong các câu lệnh nên chèn vào những dấu cách hợp lệ, sử dụng hợp lý các ký tự in thường và in hoa; giữa các đoạn chương trình nên có các dòng trắng; nên phân cấp các câu lệnh để bố trí chúng sao cho có sự thụt, thò, dễ theo dõi.

Chẳng hạn, các chương trình được viết trên đây, chúng ta đã đưa vào những lời chú thích mang ý nghĩa mô tả, như dòng mô tả chương trình sẽ làm gì, các biến được khai báo có ý nghĩa gì,…

1.9 Biểu thức số

Chương trình *ChuyenDongThangDung* ở trên đã sử dụng dạng mã nguồn sau:

```
U * T - G / 2 * T ** 2
```

Đây là một ví dụ về biểu thức số biểu diễn bằng ngôn ngữ Fortran, công thức liên kết các hằng, các biến (và các hàm như hàm tính căn bậc hai) bằng các phép toán thích hợp. Nó chỉ ra qui tắc để tính giá trị của một biểu thức đại số thông thường. Trong trường hợp trên đây, biểu thức chỉ tính một giá trị đơn nên nó được gọi là biểu thức vô hướng.

Thứ tự thực hiện các phép toán trong một biểu thức được xác định bởi thứ tự ưu tiên của các phép toán. Tuy nhiên, nếu trong biểu thức có các bộ phận nằm trong ngoặc đơn () thì chúng luôn luôn được thực hiện trước tiên. Chẳng hạn, biểu thức 1+2 * 3 sẽ cho kết quả là 7, trong khi (1 + 2) * 3 sẽ cho kết quả là 9. Chú ý rằng –3** 2 sẽ cho kết quả là –9 chứ không phải 9.

Khi có các phép toán cùng bậc ưu tiên xuất hiện liên tiếp nhau trong biểu thức chúng sẽ được thực hiện theo thứ tự từ trái sang phải, ngoại trừ phép lấy lũy thừa. Do đó, biểu thức 1 / 2 * A được thực hiện như (1 / 2) * A mà không phải như 1/ (2 * A). Đối với các phép toán lũy thừa, thứ tự thực hiện là từ phải sang trái. Ví dụ, biểu thức A ** B ** C được thực hiện theo nguyên tắc A ** (B ** C).

1.9.1 Phép chia với số nguyên

Đối với những người mới lập trình bằng Fortran, đây quả là một vấn đề không đơn giản, bởi vì nhiều khi kết quả nhận được của các biểu thức nằm ngoài dự đoán của họ. Vấn đề là ở chỗ, khi một đại lượng có kiểu số nguyên (hằng, biến hoặc biểu thức nguyên) chia cho một đại lượng có kiểu số nguyên khác, kết quả nhận được cũng sẽ có kiểu số nguyên, do đó phần lẻ thập phân sẽ bị chặt cụt. Ta hãy xét các ví dụ sau đây.

`10 / 3` cho kết quả là `3`
`19 / 4` cho kết quả là `4`
`4 / 5` cho kết quả là `0`
`-8 / 3` cho kết quả là `-2`
`3 * 10 / 3` cho kết quả là `10`
`10 / 3 * 3` cho kết quả là `9`

Như vậy, khi chia hai đại lượng nguyên cho nhau, kết quả nhận được là phần nguyên của thương, còn phần dư bị loại bỏ.

1.9.2 Biểu thức hỗn hợp

Fortran 90 cho phép thực hiện phép tính với biểu thức chứa các toán hạng có kiểu khác nhau. Nguyên tắc chung là các kiểu dữ liệu "yếu hơn" hoặc là "đơn giản hơn" buộc phải chuyển đổi sang kiểu dữ liệu "mạnh hơn". Vì kiểu số nguyên là đơn giản nhất, cho nên trong biểu thức có các toán hạng nguyên và thực thì các toán hạng nguyên phải chuyển thành các toán hạng có kiểu thực. Tuy nhiên, quá trình chuyển đổi này chỉ thực hiện đối với từng phép toán mà không nhất thiết áp dụng cho cả biểu thức. Ví dụ:

`10 / 3.0` cho kết quả là `3.33333`

`4. / 5` cho kết quả là `0.8`

`2 ** (- 2)` cho kết quả là `0 (?)`

Nhưng biểu thức

```
3 / 2 / 3.0
```

sẽ cho kết quả bằng 0.333333 vì 3 / 2 được tính trước, nhận giá trị nguyên bằng 1.

1.10 Lệnh gán. Gán hằng, gán biểu thức

Cú pháp câu lệnh gán có dạng:

```
variable = expr
```

Mục đích của câu lệnh gán là tính giá trị của biểu thức ở vế phải và gán cho biến ở vế trái. Như vậy, dấu bằng (=) trong câu lệnh gán hoàn toàn không có nghĩa như dấu bằng trong toán học, mà nó được hiểu là *dấu gán*, và nên đọc là *variable* được gán bởi giá trị của *expr*. Ví dụ, câu lệnh

```
X = A + B
```

nên đọc là nội dung của biến X được gán bởi giá trị của nội dung của biến A cộng với nội dung của biến B. Khi thực hiện câu lệnh, máy sẽ lấy giá trị của A cộng với giá trị của B, kết quả nhận được sẽ gán cho biến X.

Tương tự, câu lệnh

```
N = N + 1
```

hàm nghĩa là tăng giá trị của biến N lên một đơn vị. Đương nhiên trong toán học biểu thức này không thỏa mãn. Tác động của quá trình thực hiện câu lệnh là lấy nội dung của biến N cộng với 1, được bao nhiêu gán lại cho biến N.

Nếu *expr* không cùng kiểu dữ liệu với *variable*, nó được chuyển đổi sang kiểu dữ liệu của *variable* trước khi gán. Có nghĩa là điều đó có thể dẫn đến sai số tính toán. Ví dụ, giả sử N là biến nguyên, còn X và Y là những biến thực thì:

`N = 10. / 3` (giá trị của N là `3`)

`X = 10 / 3` (giá trị của `X` là `3.0`)

`Y = 10 / 3.` (giá trị của `Y` là `3.33333`)

Sự vô ý trong khi lập trình nhiều lúc cũng gây nên sai số không đáng có. Chẳng hạn, bạn muốn tính trung bình cộng hai số, ví dụ điểm của hai môn học, bạn đặt tên biến các môn đó là M1 và M2 mà không khai báo chúng là biến thực (vì hai biến M1 và M2 được ngầm hiểu là hai biến nguyên). Khi đó điểm trung bình sẽ được xác định bởi câu lệnh

```
TBinh = (M1 + M2) / 2
```

Với cách viết này, phần thập phân của biến TBinh sẽ bị chặt cụt, vì vế phải là kết quả của biểu thức nguyên. Nhưng nếu bạn viết câu lệnh trên dưới dạng:

```
TBinh = (M1 + M2) / 2.0
```

thì kết quả lại hoàn toàn chính xác.

Sau đây là một số ví dụ về câu lệnh gán.

```
C = (A ** 2 + B ** 2) ** 0.5 / (2 * A)
```

```
A = P * (1 + R / 100) ** N
```

Câu lệnh thứ nhất có thể được viết bằng cách khác khi sử dụng hàm thư viện SQRT của Fortran như sau:

```
C = SQRT ( A ** 2 + B ** 2 ) / (2 * A)
```

Tuy nhiên, bạn không được viết dưới dạng:

```
C = (A ** 2 + B ** 2) ** (1/2) / (2 * A)
```

Bởi vì (1/2) trong biểu thức lũy thừa nhận giá trị bằng 0 do phép chia hai số nguyên cho nhau.

1.11 Lệnh vào ra đơn giản

Quá trình nhận thông tin vào và kết xuất thông tin ra của máy tính được gọi là quá trình vào ra dữ liệu. Dạng vào ra dữ liệu đơn giản nhất trong Fortran là sử dụng các lệnh READ* và PRINT*, và được gọi là vào ra trực tiếp. Các dạng vào ra dữ liệu phức tạp hơn sẽ được đề cập đến trong những phần sau.

Trong các mục trước ta đã gặp các câu lệnh với READ* và PRINT*, nhưng chưa giải thích gì về chúng. Ở đây ta sẽ thấy rằng đó là những câu lệnh rất thường dùng mà ta cần phải tìm hiểu ngay.

1.11.1 Lệnh vào dữ liệu

Trước đây ta đã thấy các biến được gán giá trị bằng việc sử dụng câu lệnh gán, ví dụ trong chương trình *TinhTien*:

```
SoTien = 1000
```

```
LaiSuat = 0.09
```

Cách làm này rõ ràng là không linh hoạt, vì khi muốn chạy chương trình với các giá trị *số tiền ban đầu* hoặc *lãi suất* khác nhau, mỗi lần như vậy bạn phải thay đổi trực tiếp các câu lệnh gán này trong chương trình, sau đó biên dịch lại rồi mới thực hiện chương trình. Thay cho cách này bạn có thể sử dụng câu lệnh READ* như sau:

```
READ*, SoTien, LaiSuat
```

Khi bạn chạy chương trình, máy sẽ chờ bạn gõ giá trị của các biến từ bàn phím. Các giá trị này có thể được gõ trên cùng một dòng, phân cách nhau bởi các dấu cách, dấu phẩy hoặc trên các dòng khác nhau.

Dạng tổng quát của lệnh READ* như sau:

```
READ*, list
```

Trong đó *list* là danh sách các biến, viết cách nhau bởi dấu phẩy.

Khi vào dữ liệu với lệnh READ* cần chú ý một số điểm sau.

– Một dòng dữ liệu được gõ liên tục (không dùng dấu ENTER xuống dòng) được gọi là một bản ghi.

– Mỗi một lệnh READ khi nhận dữ liệu đòi hỏi một bản ghi mới. Ví dụ, câu lệnh:

```
READ*, A, B, C
```

sẽ được thỏa mãn với một bản ghi chứa 3 giá trị:

```
3 4 5
```

trong khi các câu lệnh:

```
READ*, A
READ*, B
READ*, C
```

đòi hỏi đưa vào 3 bản ghi, mỗi bản ghi chứa 1 giá trị:

```
3
4
5
```

– Khi gặp một lệnh READ mới, những dữ liệu chưa được đọc trên bản ghi hiện thời sẽ bị bỏ qua, và một bản ghi mới khác sẽ được tìm đến để nhận dữ liệu

– Nếu lệnh READ đòi hỏi nhiều dữ liệu hơn dữ liệu trên bản ghi hiện thời nó cũng sẽ tìm đến bản ghi mới tiếp theo để nhận tiếp dữ liệu

– Nếu dữ liệu không đủ đáp ứng cho lệnh READ thì chương trình sẽ bị kết thúc với thông báo lỗi.

Ví dụ: Các câu lệnh

```
READ*, A
READ*, B, C
READ*, D
```

với các bản ghi dữ liệu đưa vào là

```
1 2 3
4
7 8
9 10
```

sẽ có hiệu quả giống như các lệnh gán sau:

```
A = 1
B = 4
C = 7
D = 9
```

Tức là các giá trị 2, 3 trên bản ghi thứ nhất, 8 trên bản ghi thứ ba và 10 trên bản ghi thứ tư bị bỏ qua.

1.11.2 Đọc dữ liệu từ file TEXT

Trên thực tế thường xảy ra tình huống là bạn muốn kiểm tra chương trình trong đó mỗi lần chạy chương trình cần phải đọc vào nhiều số liệu. Giả sử bạn viết một chương trình tính trung bình của 10 số. Chắc chắn bạn sẽ rất khó chịu nếu cứ mỗi lần thử lại chương trình bạn phải gõ 10 số để chạy tính. Đó là chưa nói đến chương trình của bạn đòi hỏi nhiều dữ liệu hơn, chẳng hạn tính điểm trung bình chung học tập cho một lớp sinh viên khoảng 50 người. Để tránh phiền phức trong những trường hợp như vậy, Fortran cung cấp cho bạn một phương thức khá đơn giản nhưng rất hữu ích, nhờ đó bạn không cần phải nhập lại số liệu mỗi khi thử chương trình.

Ý tưởng là ở chỗ, trước khi chạy chương trình bạn hãy chuẩn bị số liệu và lưu chúng vào một file riêng biệt trên đĩa. Bạn có thể tạo ra file số liệu này bằng một trình soạn thảo bất kỳ và nhớ ghi lại dưới dạng file TEXT (ASCII file) với một tên file nào đó, chẳng hạn SOLIEU.TXT. Ví dụ, file số liệu này chứa nội dung gồm 3 số trên dòng đầu tiên của file:

```
3 4 5
```

Bây giờ bạn hãy viết chương trình sau đây và chạy nó:

```
PROGRAM ThuFile
OPEN(1, FILE = 'SOLIEU.TXT' )
READ(1, *) A, B, C
PRINT*, A, B, C
END
```

Câu lệnh OPEN kết nối UNIT 1 với file SOLIEU.TXT trên đĩa. Câu lệnh READ ở đây định hướng cho chương trình tìm số liệu trong file được kết nối với UNIT 1. Thông thường UNIT nhận giá trị trong khoảng 1–99. Bạn hãy chú ý phân biệt sự khác nhau giữa lệnh READ này với lệnh READ* đã đề cập ở mục trước.

1.11.3 Lệnh kết xuất dữ liệu

Lệnh PRINT* là câu lệnh rất thuận tiện cho việc kết xuất thông tin khi lượng dữ liệu không lớn, thông thường được sử dụng trong quá trình xây dựng, phát triển chương trình, hoặc đưa ra những kết quả tính toán trung gian để theo dõi tiến trình làm việc của chương trình.

Dạng tổng quát của nó như sau:

```
PRINT*, list
```

Trong đó *list* có thể là danh sách hằng, biến, biểu thức và xâu ký tự, được viết cách nhau bởi dấu phẩy (,). Xâu ký tự phải được đặt trong cặp dấu nháy đơn (‘ ’) hoặc dấu nháy kép (“ ”). Ví dụ:

```
PRINT*, "Can bac hai cua ", 2, ' la', SQRT(2.0)
```

Sau đây là một số qui tắc chung của lệnh PRINT.

– Mỗi lệnh PRINT* tạo ra một bản ghi mới

– Đối với số thực, tùy theo độ lớn giá trị của số được in mà chúng có thể được biểu diễn dưới dạng dấu phẩy tĩnh hoặc dấu phẩy động. Nếu bạn muốn in ở dạng cầu kỳ, bạn có thể sử dụng lệnh định dạng FORMAT. Ví dụ, bạn muốn in số 123.4567 dưới dạng dấu phẩy tĩnh trên 8 cột với 2 chữ số sau dấu chấm thập phân, bạn có thể viết:

```
X = 123.4567

PRINT 10, X

10 FORMAT( F8.2 )
```

Bạn có thể sử dụng lệnh PRINT để in một thông báo dài quá một dòng bằng cách dùng ký tự nối dòng. Ví dụ:

```
PRINT*, 'Day la cau thong bao duoc &

&viet bang lenh PRINT co noi dong'
```

1.11.4 Kết xuất ra máy in

Nếu muốn kết xuất ra máy in, bạn chỉ cần đặt tham số FILE=’PRN’ trong câu lệnh OPEN. Ví dụ

```
OPEN (2, FILE = 'prn' )
WRITE(2, *) 'In ra may in'
PRINT*, 'In ra man hinh'
```

Chú ý rằng lệnh WRITE phải gắn kết với số UNIT trong lệnh OPEN. Lệnh này tổng quát hơn lệnh PRINT. Ta sẽ làm quen với câu lệnh này ở những nội dung sau.

CHƯƠNG 2. CÁC CÂU LỆNH CƠ BẢN CỦA FORTRAN

Trong chương trước chúng ta đã làm quen với một số câu lệnh của Fortran, như lệnh gán, các lệnh vào ra đơn giản với READ* và PRINT*, lệnh mở file OPEN để nhận dữ liệu từ file TEXT hoặc kết xuất thông tin ra máy in, lệnh định dạng FORMAT,... Với những câu lệnh đó bạn đã có thể viết được một số chương trình đơn giản. Chương này sẽ nghiên cứu những câu lệnh phức tạp hơn.

2.1 Lệnh chu trình (DO Loops)

Khi viết chương trình, có thể bạn bắt gặp tình huống một hoặc nhiều câu lệnh nào đó phải thực hiện lặp lại nhiều lần giống nhau, chẳng hạn, bạn muốn in 10 số nguyên liên tiếp, mỗi lần in một số, bạn phải dùng đến 10 câu lệnh in ra. Bạn sẽ cảm thấy khó chịu khi làm điều đó. Tuy nhiên, thay cho cách làm trên đây, Fortran hỗ trợ cho bạn một cấu trúc câu lệnh khá đơn giản nhưng rất hiệu quả. Đó là câu lệnh chu trình. Cú pháp câu lệnh có thể có các dạng sau.

```
DO m bdk = TriDau, TriCuoi [, Buoc]
    Các câu lệnh
m Câu lệnh kết thúc
( hoặc: m CONTINUE )
```

hoặc

```
DO bdk = TriDau, TriCuoi [, Buoc]
    Các câu lệnh
END DO
```

Trong đó **bdk, TriDau, TriCuoi, Buoc** phải có cùng kiểu dữ liệu, **m** là nhãn của câu lệnh kết thúc chu trình; trong trường hợp không thể sử dụng câu lệnh kết thúc như vậy, bạn có thể thay thế nó bằng câu lệnh **m CONTINUE**. Nếu **TriDau** < **TriCuoi** thì **Buoc** phải là một số dương, ngược lại nếu **TriDau** > **TriCuoi** thì **Buoc** phải là một số âm. Nếu **Buoc=1** thì có thể bỏ qua **Buoc**. Các câu lệnh nằm giữa **DO** và **m Câu lệnh kết thúc** (kể cả **Câu lệnh kết thúc**) hoặc **m CONTINUE**, hoặc **END DO** là những câu lệnh được thực hiện lặp lại. Số lần lặp lại được xác định bởi:

MAX { (*TriCuoi* – *TriDau* + *Buoc*) / *Buoc*, 0 }

Tác động của lệnh chu trình được mô tả trên hình 2.1. Có thể tóm tắt tác động này qua các bước sau.

1) Bắt đầu chu trình **bdk** được gán giá trị bằng **TriDau**.

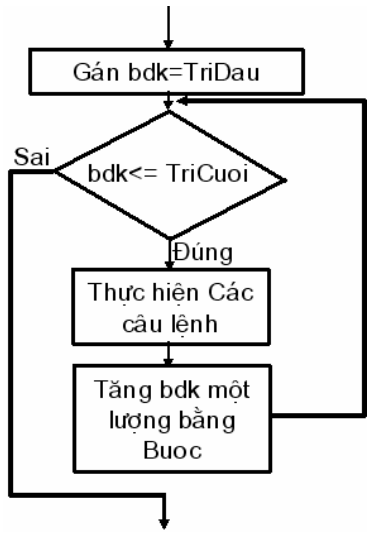
2) Sau đó chương trình sẽ thực hiện biểu thức so sánh **bdk<=TriCuoi** hoặc **bdk>=TriCuoi**:

   a) Nếu biểu thức cho kết quả .TRUE. (đúng):

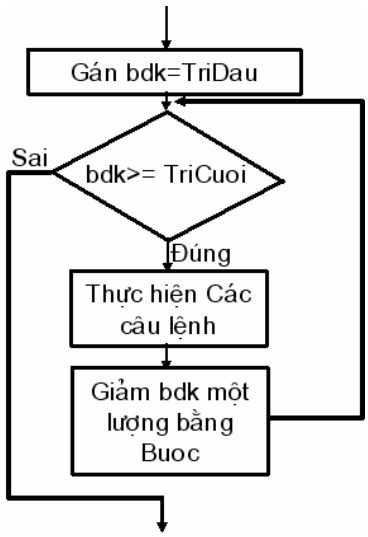
      a.1) Tiếp tục thực hiện **Các câu lệnh**, kể cả **Câu lệnh kết thúc**, nằm trong chu trình rồi tăng hoặc giảm **bdk** một lượng bằng trị tuyệt đối của **Buoc**

a.2) Quay về thực hiện bước 2)

b) Nếu biểu thức cho kết quả .FALSE. (sai) thì kết thúc chu trình



a) Trường hợp **TriDau<=TriCuoi**



b) Trường hợp **TriDau>=TriCuoi**

Hình 2.1 Sơ đồ khối mô tả tác động của lệnh chu trình DO

Như vậy, có thể xảy ra tình huống các câu lệnh trong chu trình lặp sẽ không được thực hiện một lần nào nếu ngay từ đầu bước 2b) được thỏa mãn.

Ví dụ 1. Chương trình sau đây sẽ tính tổng các số nguyên liên tiếp từ N1 đến N2, trong đó N1 và N2 là do bạn nhập vào từ bàn phím.

```
INTEGER N1, N2, TONG, I
PRINT '(A\)', ' CHO GIA TRI N1, N2 (N1<=N2):'
READ*, N1, N2
TONG = 0
DO I = N1,N2,1
   TONG = TONG + I
   PRINT*, I
ENDDO
PRINT '(" TONG=",I5)', TONG
END
```

Khi chạy chương trình bạn sẽ nhận được các số nguyên liên tiếp từ N1 đến N2 hiện lên màn hình và cuối cùng là thông báo kết quả tổng của các số từ N1 đến N2. Trong chương trình này, các câu lệnh

```
PRINT '(A\)', ' CHO GIA TRI N1, N2 (N1<=N2):'
```

và

```
PRINT '(" TONG=",I5)', TONG
```

đã chứa trong đó lệnh định dạng FORMAT. Bạn đừng vội quan tâm đến mà hãy làm theo như vậy. Tuy nhiên, nếu bạn cảm thấy hơi xa lạ, bạn có thể thay thế phần định dạng này bởi dấu sao (*).

Trong câu lệnh

```
DO I = N1,N2,1
```

số 1 cuối cùng là giá trị của **Buoc**, bạn có thể bỏ qua nó mà không ảnh hưởng gì đến kết quả. Nhưng nếu bạn thay nó bằng –1 thì khi nhập N1 và N2 bạn phải lưu ý N1>=N2.

Các câu lệnh

```
DO I = N1,N2,1
   TONG = TONG + I
   PRINT*, I
ENDDO
```

cũng có thể được thay thế bởi các câu lệnh sau đây

```
DO 100 I = N1,N2,1
   TONG = TONG + I
100 PRINT*, I
```

Trong trường hợp này câu lệnh

```
100 PRINT*, I
```

là câu lệnh kết thúc chu trình. Bạn cũng có thể dùng lệnh CONTINUE để kết thúc chu trình:

```
DO 100 I = N1,N2,1
   TONG = TONG + I
   PRINT*, I
100 CONTINUE
```

Lệnh CONTINUE trong trường hợp này có thể xem là "thừa", tuy vậy trong nhiều trường hợp, để an toàn và rõ ràng hơn, bạn có thể sử dụng những câu lệnh "thừa" kiểu này.

Ví dụ 2. Chương trình tính căn bậc hai của số ***a*** theo phong pháp Newton có thể được mô tả như sau:

1) Nhập vào số a

2) Khởi tạo x bằng 1 (gán giá trị cho x bằng 1)

3) Lặp lại **6** lần các bước sau đây:

   a) Thay x bởi (x + a/x)/2

   b) In giá trị của x

4) Kết thúc chương trình

Mã nguồn chương trình như sau:

```
PROGRAM Newton  ! Tinh can bac hai bang pp newton
REAL A        ! Số sẽ lấy căn bậc hai
INTEGER I       ! Biến đếm phép lặp
REAL X  ! Giá trị gần đúng của căn bậc hai của a
WRITE( *,*) ' Cho so se lay can bac hai: '
READ*, A
PRINT*
```

```
X = 1    ! Khởi tạo giá trị ban đầu của x (??)
DO I = 1, 6
   X = (X + A / X) / 2
   PRINT*, X
ENDDO
PRINT*
PRINT*, 'Can bac 2 cua a tinh theo F90 la:', SQRT(A)
END
```

Khi chạy chương trình bạn sẽ thấy trên màn hình xuất hiện **6** lần giá trị của X. Giá trị ở dòng thứ **6** được xem là gần đúng của căn bậc hai của **a** tính bằng phương pháp lặp Newton, còn giá trị in ở dòng cuối cùng là căn bậc hai của **a** tính bằng hàm thư viện của Fortran (SQRT). Giữa chúng có thể có sự khác nhau; khi **a** càng lớn thì sự khác nhau đó càng nhiều. Trong trường hợp này bạn hãy tăng số lần lặp bằng cách thay số **6** ở dòng lệnh `DO I = 1, 6` bằng một số lớn hơn và chạy lại chương trình. Việc so sánh kết quả nhận được sau mỗi lần thay đổi dòng lệnh này có thể sẽ rất có ích cho bạn đấy.

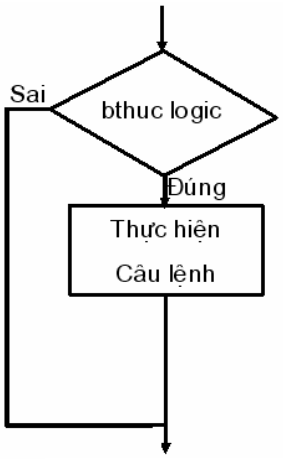
Chú ý: Nói chung Fortran cho phép các biến **bdk, TriDau, TriCuoi, Buoc** có kiểu dữ liệu số nguyên và số thực. Tuy nhiên chúng tôi khuyến cáo bạn không nên dùng kiểu dữ liệu thực do số thực được biểu diễn ở dạng gần đúng, có thể gây nên những sai số không lường trước được.

2.2 Lệnh rẽ nhánh với IF

Cấu trúc rẽ nhánh là kiểu cấu trúc rất phổ biến đối với các ngôn ngữ lập trình. Trong Fortran, cấu trúc rẽ nhánh được cho khá đa dạng. Sau đây ta sẽ lần lượt xét từng trường hợp.

2.2.1 IF (BThuc_Logic) Câu lệnh

Trong đó *Câu* lệnh không được là một trong các câu lệnh có cấu trúc khác, như IF, DO,… Tác động của câu lệnh IF là nếu *BThuc_Logic* nhận giá trị .TRUE. (đúng) thì chương trình sẽ thực hiện *Câu lệnh* ngay sau đó, ngược lại, nếu *BThuc_Logic* nhận giá trị .FALSE. (sai) thì *Câu lệnh* sẽ bị bỏ qua và chương trình tiếp tục với những câu lệnh khác sau IF. Sơ đồ khối mô tả tác động này được cho trên hình 2.2.



Hình 2.2

Ví dụ: Hãy đọc vào một số và cho biết đó là số dương, số âm hay số 0. Chương trình có thể được viết như sau.

```
REAL X
PRINT '(A\)',' CHO MOT SO:'
READ*, X
IF (X.GT.0) PRINT *, ' DAY LA SO DUONG'
IF (X.LT.0) PRINT *, ' DAY LA SO AM'
IF (X.EQ.0) PRINT *, ' DAY LA SO 0'
END
```

Như đã thấy, đối với cấu trúc này, khi *BThuc_Logic* nhận giá trị .TRUE. (đúng) thì chỉ có một câu lệnh sau đó được thực hiện.

2.2.2 IF (BThuc_Logic) THEN

```
        Các câu lệnh
    END IF
```

Về nguyên tắc, tác động của câu lệnh này hoàn toàn giống với cấu trúc IF trên đây. Sự khác nhau giữa chúng chỉ là ở chỗ, trong cấu trúc trước, khi *điều kiện* được thỏa mãn (*BThuc_Logic* nhận giá trị .TRUE.) thì chỉ có một câu lệnh sau IF được thực hiện, còn trong trường hợp này nếu *BThuc_Logic* nhận giá trị .TRUE. thì *có thể có nhiều câu lệnh* nằm giữa IF … THEN và END IF sẽ được thực hiện. (*Các câu lệnh* hàm nghĩa là có thể có nhiều câu lệnh).

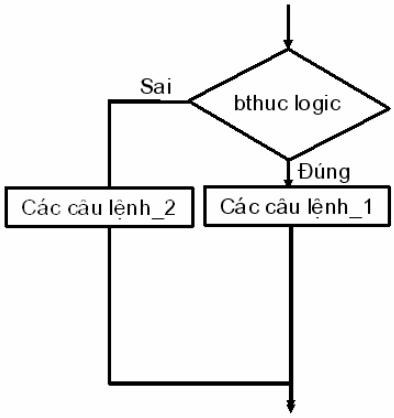
Ví dụ: Viết chương trình nhập vào hai số thực, nếu chúng đồng thời khác 0 thì tính tổng, hiệu, tích, thương của chúng.

```
REAL X, Y, TONG, HIEU, TICH, THUONG
PRINT*, ' CHO 2 SO THUC:'
READ*, X, Y   ! Doc cac so X, Y tu ban phim
IF (X.NE.0.AND.Y.NE.0) THEN  ! X, Y dong thoi khac 0
    TONG = X + Y
    HIEU = X - Y
    TICH = X * Y
    THUONG = X / Y
    PRINT*,' TONG CUA ',X,' VA ',Y,' LA:',TONG
    PRINT*,' HIEU CUA ',X,' VA ',Y,' LA:',HIEU
    PRINT*,' TICH CUA ',X,' VA ',Y,' LA:',TICH
    PRINT*,' THUONG CUA ',X,' VA ',Y,' LA:',THUONG
END IF
IF (X.EQ.0.OR.Y.EQ.0) THEN  ! Mot trong hai so = 0
   PRINT*,' MOT TRONG HAI SO VUA NHAP = 0'
END IF
END
```

2.2.3 IF (BThuc_Logic) THEN

```
        Các câu lệnh_1
    ELSE
        Các câu lệnh_2
    END IF
```

Khác với hai cấu trúc trên, trong cấu trúc này việc thực hiện chương trình có thể rẽ về một trong hai "nhánh": Nếu *BThuc_Logic* nhận giá trị .TRUE. thì chương trình sẽ thực hiện *Các câu lệnh_1*, ngược lại, chương trình sẽ thực hiện *Các câu lệnh_2*. Sơ đồ khối mô tả tác động của cấu trúc này được cho trên hình 2.3



Hình 2.3

Ví dụ: Viết chương trình nhập vào từ bàn phím ba số thực, là kích thước ba cạnh của một tam giác. Nếu ba số đó thỏa mãn điều kiện là ba cạnh của một tam giác thì tính diện tích của tam giác. Ngược lại thì đưa ra thông báo "BA SO NAY KHONG PHAI LA 3 CANH CUA TAM GIAC".

```
REAL A,B,C  ! Ba canh Tam giac
REAL P,S    ! Nua chu vi va Dien tich
LOGICAL L1  !Ba canh Tam giac phai la nhung so duong
LOGICAL L2  ! Ba canh phai thoa man
            ! cac bat dang thuc tam giac
PRINT*, ' CHO 3 CANH CUA TAM GIAC:'
READ*, A,B,C
L1 = A>0.AND.B>0.AND.C>0
L2 = A+B>C.AND.B+C>A.AND.C+A>B
IF (L1.AND.L2) THEN  ! Thoa man dieu kien Tam giac
    P = (A+B+C)/2
    S = SQRT(P*(P-A)*(P-B)*(P-C))
    PRINT*,' DIEN TICH TAM GIAC = ',S
ELSE   ! Khong thoa man dieu kien Tam giac
    PRINT*,"BA SO NAY KHONG PHAI LA 3 CANH &
           &CUA TAM GIAC"
END IF
END
```

Trong chương trình này ta đã sử dụng hai biến lôgic L1, L2 để xác định ba số nhập vào có thỏa mãn điều kiện là ba cạnh của một tam giác hay không. Bạn nên chú ý cách dùng các biến này, vì nó sẽ rất có ích trong những trường hợp phức tạp hơn.

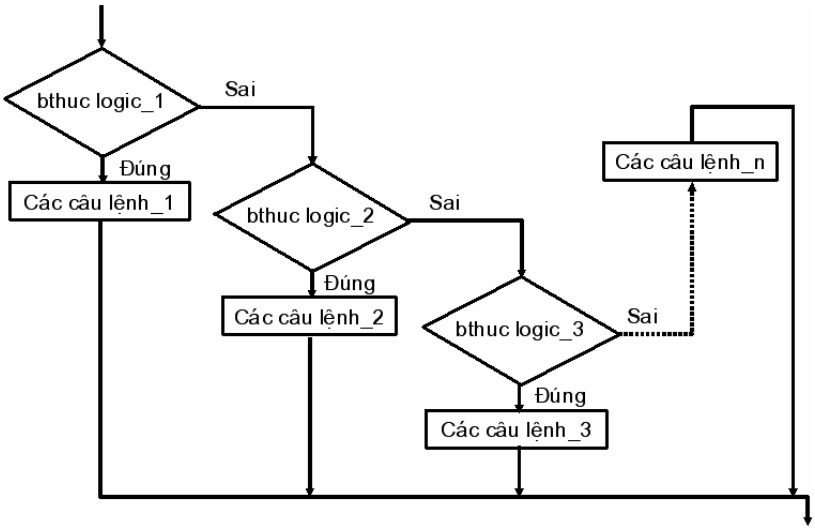
2.2.4 IF (BThuc_Logic_1) THEN

```
      Các câu lệnh_1
   ELSE IF (BThuc_Logic_2) THEN
      Các câu lệnh_2
   ELSE IF (BThuc_Logic_3) THEN
      Các câu lệnh_3
   ...
   ELSE
      Các câu lệnh_n
   END IF
```

Cấu trúc này được gọi là cấu trúc khối IF (Block IF). Tác động của cấu trúc này được mô tả trên hình 2.4.

Trước hết chương trình sẽ kiểm tra *BThuc_Logic_1*. Nếu *BThuc_Logic_1* nhận giá trị .TRUE. thì *Các câu lệnh_1* sẽ được thực hiện; nếu *BThuc_Logic_1* nhận giá trị .FALSE. thì chương trình sẽ kiểm tra đến *BThuc_Logic_2*. Nếu *BThuc_Logic_2* nhận giá trị .TRUE. thì *Các câu lệnh_2* sẽ được thực hiện; nếu *BThuc_Logic_2* nhận giá trị .FALSE. thì chương trình sẽ kiểm tra *BThuc_Logic_3*,… Quá trình cứ tiếp diễn như vậy cho đến khi nếu tất cả các *BThuc_Logic* đều nhận giá trị .FALSE. thì

chương trình sẽ thực hiện *Các câu lệnh_n*. Nếu *Các câu lệnh* ở giai đoạn nào đó của quá trình đã được thực hiện, chương trình sẽ thoát khỏi cấu trúc IF và chuyển điều khiển đến những câu lệnh ngay sau đó, ngoại trừ trường hợp trong *Các câu lệnh* có lệnh chuyển điều khiển GOTO đến một vị trí khác trong chương trình.



Hình 2.4

Ví dụ: Viết chương trình nhập điểm trung bình chung học tập (TBCHT) của một sinh viên và cho biết sinh viên đó được xếp loại học tập như thế nào nếu tiêu chuẩn xếp loại được qui định như sau: Loại xuất sắc nếu TBCHT >=9; Loại giỏi nếu 9<TBCHT<=8; Loại khá nếu 8<TBCHT<=7; Loại trung bình nếu 7<TBCHT<=5 và loại yếu nếu TBCHT<5.

```
PROGRAM XEPLOAI_1
INTEGER DIEM
WRITE (*,'(A\)') ' CHO DIEM TBCHT: '
READ*, DIEM
IF (DIEM.LT.0.OR.DIEM.GT.10) THEN
    PRINT*, ‘ DIEM KHONG HOP LE’
    STOP
END IF
IF (DIEM.GE.9) THEN
    PRINT*,' LOAI XUAT SAC'
ELSE IF (DIEM.GE.8) THEN
    PRINT*,' LOAI GIOI'
ELSE IF (DIEM.GE.7) THEN
    PRINT*,' LOAI KHA'
ELSE IF (DIEM.GE.5) THEN
    PRINT*,' LOAI TRUNG BINH'
ELSE
    PRINT*,' LOAI YEU'
END IF
END
```

Chương trình trên đây có thể viết bằng cách khác như sau.

```
PROGRAM XEPLOAI_2
INTEGER DIEM
WRITE (*,'(A\)') ' CHO DIEM TBCHT: '
```

```
READ*, DIEM
IF (DIEM.LT.0.OR.DIEM.GT.10) THEN
    PRINT*, ‘ DIEM KHONG HOP LE’
    STOP
END IF
IF (DIEM.GE.9) PRINT*,' LOAI XUAT SAC'
IF (DIEM.GE.8.AND.DIEM.LT.9) PRINT*,' LOAI GIOI'
IF (DIEM.GE.7.AND.DIEM.LT.8) PRINT*,' LOAI KHA'
IF(DIEM.GE.5.AND.DIEM.LT.7)PRINT*,' LOAI TRUNG BINH'
IF (DIEM.LT.5) PRINT*,' LOAI YEU'
END
```

Trong hai chương trình trên, lệnh STOP làm kết thúc chương trình khi DIEM nhập vào không hợp lệ (DIEM < 0 hoặc DIEM > 10). Rõ ràng, nếu sử dụng cấu trúc khối IF như ở chương trình XEPLOAI_1, các biểu thức lôgic sẽ gọn gàng hơn, và chương trình trông sáng sủa hơn so với chương trình XEPLOAI_2.

2.2.5 Lệnh nhảy vô điều kiện GOTO

Lệnh này có dạng sau:

```
GOTO m
```

Trong đó **m** là nhãn của một câu lệnh nào đó sẽ được chuyển điều khiển tới trong chương trình. Khi gặp lệnh GOTO, ngay lập tức chương trình sẽ chuyển điều khiển tới câu lệnh có nhãn **m**. Nếu trong chương trình không có câu lệnh nào có nhãn **m** thì lỗi sẽ xuất hiện. Hơn nữa, câu lệnh sẽ được chuyển điều khiển tới không được phép nằm trong vòng kiểm soát của lệnh chu trình DO và cấu trúc rẽ nhánh IF.

Ví dụ, đoạn chương trình sau đây

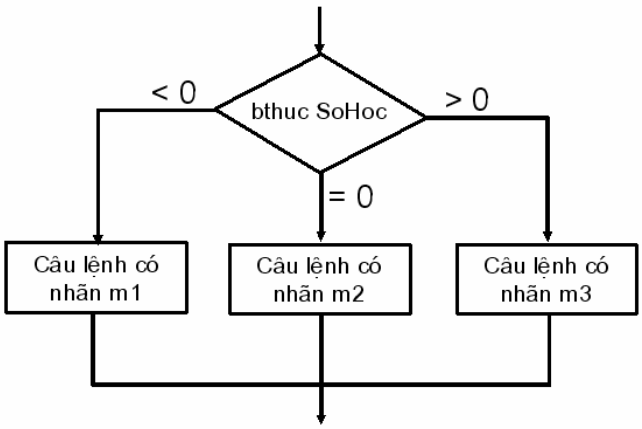
```
IF (L1) THEN
  I = 1
  J = 2
ELSE IF (L2) THEN
  I = 2
  J = 3
ELSE
  I = 3
  J = 4
END IF
```

có thể được thay thế bởi đoạn chương trình sau khi sử dụng lệnh GOTO

```
IF (.NOT.L1) GOTO 10
   I = 1
   J = 2
   GOTO 30
10 IF (.NOT.L2) GOTO 20
   I = 2
   J = 3
   GOTO 30
20 I = 3
   J = 4
30 CONTINUE
```

2.2.6 IF (BThuc SoHoc) m1, m2, m3

Trong đó *BThuc SoHoc* là một biểu thức số học, có thể có kiểu nguyên hoặc thực; *m1, m2, m3* là nhãn của các câu lệnh có trong chương trình. Người ta gọi cấu trúc IF này là IF số học, vì quyết định rẽ nhánh phụ thuộc vào dấu của *BThuc SoHoc*. Tác động của cấu trúc này được mô tả trên hình 2.5.



Hình 2.5

Trước hết chương trình sẽ tính giá trị của *BThuc SoHoc*. Nếu *BThuc SoHoc* nhận giá trị âm, chương trình sẽ chuyển điều khiển tới câu lệnh có nhãn *m1*; nếu *BThuc SoHoc* nhận giá trị bằng 0, chương trình sẽ chuyển điều khiển tới câu lệnh có nhãn *m2*; nếu *BThuc SoHoc* nhận giá trị dương, điều khiển sẽ được chuyển tới câu lệnh có nhãn *m3*. Hai trong ba nhãn *m1, m2, m3* có thể trùng nhau, có nghĩa là hai nhánh của điều khiển có thể chuyển đến cùng một câu lệnh. Tuy nhiên các câu lệnh có nhãn *m1, m2, m3* không được phép nằm trong vòng kiểm soát của lệnh chu trình DO và cấu trúc rẽ nhánh IF.

Ví dụ:

```
INTEGER N
PRINT*, ' CHO MOT SO NGUYEN '
READ*, N
IF (N-50) 10, 20, 30
10 PRINT*,' SO NAY NHO HON 50'
   GOTO 40
20 PRINT*,' SO NAY BANG 50'
   GOTO 40
30 PRINT*,' SO NAY LON HON 50'
40 CONTINUE
END
```

Hoặc

```
INTEGER N
PRINT*, ' CHO MOT SO NGUYEN '
READ*, N
IF (N-50) 10, 10, 30
10 PRINT*,' SO NAY NHO HON HOAC BANG 50'
   GOTO 40
30 PRINT*,' SO NAY LON HON 50'
40 CONTINUE
END
```

2.3 Thao tác với hằng và biến ký tự (CHARACTER)

Ở mục 1.4.2 chúng ta đã xét kiểu dữ liệu ký tự và cách khai báo các biến, hằng có kiểu ký tự. Hằng ký tự là tập hợp các ký tự thuộc bảng mã ASCII, không bao gồm

các ký tự điều khiển, lập thành một dãy đặt trong cặp dấu nháy đơn (‘ ’) hoặc dấu nháy kép (“ ”). Biến ký tự là biến có kiểu ký tự, được khai báo bởi lệnh CHARACTER. Các hằng và biến ký tự có thể được gộp với nhau để tạo thành một xâu ký tự mới. Ví dụ, chương trình sau đây sẽ hỏi tên bạn, khi bạn gõ tên mình vào, máy sẽ đa ra lời chào mừng bạn.

```
Program WelCome
CHARACTER *20 Name
Print *,'Ten ban la gi ?'
Read*, Name
Write(*,*) 'Xin chao ban ', Name
END
```

Trong chương trình trên, lệnh PRINT in ra một hằng ký tự (`Ten ban la gi ?`), lệnh READ* đọc giá trị của biến ký tự *Name* do bạn nhập vào, còn lệnh WRITE in ra một hằng ký tự (`Xin chao ban `) và giá trị của biến ký tự *Name*. Biến *Name* đã được khai báo có kiểu ký tự với độ dài cực đại là 20. Khi chạy chương trình này, nếu bạn nhập một xâu có chứa dấu cách ở giữa (ví dụ *Hoang Nam*) mà không đặt trong cặp dấu nháy, giá trị của biến *Name* có thể sẽ bị cắt bỏ phần bên phải nhất kể từ vị trí dấu cách. Chẳng hạn:

```
Ten ban la gi ?
Hoang Nam
Xin chao ban Hoang
```

(chứ không phải *Hoang Nam* như bạn mong muốn). Nhưng nếu bạn gõ vào “*Hoang Nam*” (đặt trong dấu nháy) thì kết quả nhận được lại hoàn toàn bình thường. Nếu bạn thay câu lệnh

```
Read*, Name
```

bởi câu lệnh

```
Read (*,'(A)') Name
```

thì khi nhập vào giá trị của biến *Name* bạn không được đặt trong dấu nháy, vì trong trường hợp này các dấu nháy sẽ được hiểu là một bộ phận của *Name*. Ví dụ:

```
Ten ban la gi ?
Hoang Nam
Xin chao ban Hoang Nam
```

nhưng

```
Ten ban la gi ?
'Hoang Nam'
Xin chao ban 'Hoang Nam'
```

Nếu độ dài xâu ký tự vượt quá độ dài khai báo cực đại của biến ký tự thì phần bên phải nhất của xâu sẽ bị cắt bỏ. Ví dụ:

```
CHARACTER *7 Name
Name = 'Hoang Nam'
```

Kết quả biến *Name* sẽ có giá trị là `'Hoang N'`.

Chương trình sau đây là một ví dụ về sử dụng phép toán gộp các xâu ký tự.

```
Character Ho*7,Dem*7, Ten*7, HoTen*21
Ho='Nguyen'
Dem='Van'
Ten='Thanh'
HoTen=Ho//Dem//Ten
PRINT*,'12345671234567 1234567'
PRINT*, HoTen
END
```

Khi chạy chương trình này bạn sẽ nhận được:

```
12345671234567 1234567
Nguyen Van    Thanh
```

Biến *HoTen* có giá trị bằng giá trị của ba biến *Ho, Dem* và *Ten*. Vì ba biến này đều có độ dài khai báo là 7 (ký tự), nên giữa *Van* và *Thanh* có 4 khoảng trống (4 dấu cách).

2.4 Kết hợp DO và IF

Chu trình DO có thể bao hàm cả cấu trúc rẽ nhánh IF và ngược lại. Nghĩa là chu trình DO có thể kiểm soát toàn bộ cấu trúc IF, hoặc trong cấu trúc IF có chứa trọn vẹn chu trình DO. Nhất thiết chúng không được phép giao nhau. Cú pháp tổng quát của các cấu trúc này như sau.

– IF nằm trong chu trình DO:

```
DO bdk = TriDau, TriCuoi, Buoc
   ...
   IF (BThuc Logic) THEN
       ...
   END IF
   ...
END DO
```

– Chu trình DO nằm trong cấu trúc IF:

```
IF (BThuc Logic) THEN
   ...
   DO bdk = TriDau, TriCuoi, Buoc
     ...
   END DO
   ...
END IF
```

Sau đây là một số ví dụ.

```
PROGRAM IFinDO  ! Cấu trúc IF nằm trong chu trình DO
Do i=1,100
   write(*,*) i
```

```
   If (i.EQ.20) then
      print*,' Ket thuc sau lan lap thu ',i
      stop
   End If
Enddo
END
```

```
PROGRAM DOinIF  ! Chu trình DO nằm trong cấu trúc IF
Print*,' Cho hai so nguyen: '
Read*, M, N
if (M.LE.N) then
   write(*,*)' Lap tu ',M, ' den ',N
   Do i=M,N
      write(*,*) i
   Enddo
Else
   write(*,*)' M > N'
End If
END
```

2.5 Rẽ nhánh với cấu trúc SELECT CASE

Một trong những phương pháp hữu hiệu để chuyển điều khiển trong chương trình là sử dụng cấu trúc rẽ nhánh SELECT CASE. Dạng tổng quát của cấu trúc này như sau.

```
SELECT CASE (BThuc_Chon)
       CASE (Chon1)
            Các câu lệnh_1
       CASE (Chon2)
            Các câu lệnh_2
       ...
       CASE DEFAULT
            Các câu lệnh_n
END SELECT
```

Trong đó *BThuc_Chon, Chon1, Chon2,…* phải có cùng kiểu dữ liệu số nguyên, lôgic hoặc CHARACTER*1. *BThuc_Chon* là biểu thức được tính toán, nó còn được gọi là *chỉ số chọn*. *Chon1, Chon2,…* là các giá trị hoặc khoảng giá trị có thể có của *BThuc_Chon*. Nếu có nhiều giá trị rời rạc, chúng phải được liệt kê cách nhau bởi dấu phẩy; nếu là khoảng giá trị liên tiếp, chúng phải được biểu diễn bởi hai giá trị đầu và cuối khoảng phân cách nhau bởi dấu hai chấm (:). *Các câu lệnh_1, Các câu lệnh_2,…* là tập các câu lệnh thực hiện. Nếu biểu diễn sơ đồ khối cấu trúc này nó sẽ gần giống với hình 2.4, trong đó các *BThuc Logic_1*,… được thay bởi mệnh đề *nếu BThuc_Chon thuộc tập Chon1,…*

Tác động của cấu trúc này có thể mô tả như sau.

Bắt đầu: Xác định giá trị của (*Bieu_Thuc_Chon*)

Nếu giá trị của *Bieu_Thuc_Chon* thuộc tập (*Chon1*) thì

Thực hiện **Các câu lệnh_1**

Nếu giá trị của *Bieu_Thuc_Chon* thuộc tập (*Chon2*)

Thực hiện **Khối lệnh_2**

...

Nếu giá trị của *Bieu_Thuc_Chon nhận các giá trị khác* thì

Thực hiện **Khối lệnh_n**

Kết thúc

Ví dụ 1: Viết chương trình xem số ngày của một tháng nào đó trong năm.

```
INTEGER Month, Year
Print'(A\)',' Xem so ngay cua thang nao?'
Read*, Month
SELECT CASE (Month)
CASE (1,3,5,7,8,10,12)
       Print*,' Thang ', Month,' co 31 ngay'
CASE (4,6,9,11)
       Print*,' Thang ', Month,' co 30 ngay'
CASE (2)
Print'(A\)',' Nam nao?'
Read*, Year
IF (Year.EQ.2000) then
Print*,' Thang ', Month,' Nam 2000 co &
          &29 ngay'
          Else IF (Mod(Year,4).EQ.0.AND.&
          &Mod(Year,100).NE.0) then
Print*,' Thang ', Month,' Nam ', Year,&
        &' co 29 ngay'
Else
Print*,' Thang ', Month,' Nam ', Year,&
        &' co 28 ngay'
End IF
CASE DEFAULT
Print*,' Khong co thang ', Month
END SELECT
END
```

Ví dụ 2: Gõ một ký tự và cho biết đó là chữ cái hay chữ số.

```
CHARACTER*1  char
Print*,' Hay go mot ky tu:'
Read*, Char
SELECT CASE (char)
      CASE ('0':'9')
         WRITE (*, *) "Day la chu so ", Char
      CASE ('A':'Z','a':'z')
         WRITE (*, *) "Day la chu cai ",Char
      CASE DEFAULT
         WRITE (*, *) "Day khong phai chu so &
                  &hoac chu cai."
```

```
         WRITE (*, *) "Day la ky tu ", Char
      END SELECT
END
```

CHƯƠNG 3. CÁC CẤU TRÚC MỞ RỘNG

3.1 Chu trình DO tổng quát và chu trình DO lồng nhau

Các chu trình DO đã xét ở chương 2 có thể được gán tên và gọi là chu trình DO tổng quát. Cú pháp câu lệnh như sau.

```
[Ten_ChuTrinh:] DO bdk = TriDau, TrCuoi [, Buoc]
    Các câu lệnh
END DO [Ten_ChuTrinh]
```

Về nguyên tắc tác động, chu trình DO này hoàn toàn giống với chu trình DO trước đây.

Các chu trình DO cũng có thể lồng nhau sao cho chu trình ngoài kiểm soát toàn bộ chu trình trong. Có thể có các cấu trúc lồng nhau sau đây.

```
DO m1 bdk1= ...
    ...
    DO m2 bdk2=...
        Các câu lệnh
   m2 Câu lệnh kết thúc
   [ hoặc: m2 CONTINUE ]
       ...
m1 Câu lệnh kết thúc
[ hoặc: m2 CONTINUE ]
```

Hoặc

```
ChuTrinh_1: DO bdk1= ...
      ...
      ChuTrinh_2: DO bdk2=...
         Các câu lệnh
      END DO ChuTrinh_2
          ...
END DO ChuTrinh_1
```

Ví dụ: Lập chương trình tính tổng điểm thi đại học cho các thí sinh.

```
PROGRAM TinhDiem
WRITE(*,'(A\)')" Cho so thi sinh can tinh:"
Read*, N
ThiSinh: DO i=1,N
TongDiem=0.0
MonThi: DO j=1,3
```

```
          Print*, "Cho diem thi mon ", J,&
        &" cua TS thu ", I
Read*,Diem
TongDiem=TongDiem+Diem
     END DO MonThi
     Write(*,'(" Diem TS ",I3,"=",F5.1)')I,TongDiem
END DO ThiSinh
END
```

3.2 Cấu trúc IF tổng quát và cấu trúc IF lồng nhau

Cấu trúc IF cũng có thể được đặt tên và gọi là cấu trúc IF tổng quát. Cú pháp như sau.

```
Ten_Cau_Truc: IF (BThuc Logic) THEN
        ...
END IF Ten_Cau_Truc
```

Hoặc

```
Ten_Cau_Truc: IF (BThuc Logic) THEN
        ...
ELSE Ten_Cau_Truc
        ...
END IF Ten_Cau_Truc
```

Hoặc

```
Ten_Cau_Truc: IF (BThuc Logic_1) THEN
        ...
ELSE IF (BThuc Logic_2)
        ...
ELSE IF (BThuc Logic_3)
        ...
ELSE Ten_Cau_Truc
        ...
END IF Ten_Cau_Truc
```

Nói chung không có gì khác biệt về chức năng giữa cấu trúc IF tổng quát và cấu trúc IF thông thường trước đây đã xét, ngoại trừ thêm *Ten_Cau_Truc* để "đánh dấu" xác định vị trí của khối cấu trúc.

Cấu trúc IF cũng có thể lồng nhau sao cho cấu trúc này nằm trọn vẹn trong cấu trúc kia.

```
IF (BThuc Logic_1) THEN
      ...
      IF (BThuc Logic_2) THEN
         ...
```

```
        END IF
        ...
    END IF
```

Hoặc

```
    Ngoai: IF (BThuc Logic_1) THEN
        ...
        Trong: IF (BThuc Logic_2) THEN
           ...
        END IF Trong
        ...
    END IF Ngoai
```

Ví dụ: Lập chương trình giải phương trình $ax^2 + bx + c = 0$.

Để giải bài toán này trước hết ta lập một dàn bài thực hiện gồm các bước sau.

Bước 1: Nhập các hệ số a, b, c

Bước 2: Nếu a=0: (giải phương trình bậc nhất bx + c = 0)

- Nếu b=0:
  - + Nếu c=0: Trả lời = Vô số nghiệm
  - + Nếu c≠0: Trả lời = Vô nghiệm
- Nếu b≠0: Trả lời = nghiệm x=–c/b

Bước 3: Nếu a≠0: (giải phương trình bậc hai $ax^2 + bx + c = 0$)

- Tính DelTa=b*b–4*a*c
- Nếu DelTa<0: Trả lời=Vô nghiệm (hoặc nghiệm ảo)
- Nếu DelTa≥0:
  - + Tính các nghiệm
  - + Trả lời: Nghiệm x1, x2 =(–b±(DelTa)1/2)/(2a)

Bước 4: Kết thúc

Dựa trên dàn bài này ta có mã chương trình:

```
PROGRAM GiaiPTb2
REAL a, b, c, DelTa, x1, x2
Print*,' Cho cac he so a,b,c:'
Read*, a,b,c
XetA: IF (a==0) THEN
    XetB: IF (b==0) THEN
       XetC: IF (c==0) THEN
          Print*,' Phuong Trinh co VO SO NGHIEM'
       ELSE XetC
          Print*,' Phuong Trinh VO NGHIEM'
       END IF XetC
```

```
    ELSE XetB
       Print*, 'Phuong trinh co 1 nghiem x=', -c/b
    END IF XetB
ELSE XetA
    DelTa=b*b-4*a*c
    XetDelTa: IF (DelTa<0) THEN
       Print*,'Phuong trinh KHONG CO NGHIEM THUC'
    ELSE XetDelTA
       DelTa=SQRT(DelTa)
       X1=( -b - DelTa) / (2*a)
       X2=( -b + DelTa) / (2*a)
       Print*,'PT co cac nghiem X1=',X1,'  X2=', X2
    END IF XetDelTa
END IF XetA
END
```

3.3 Chu trình ngầm

Trước hết hãy xét hai ví dụ sau.

Ví dụ 1:

```
DO I = 1, 5
    PRINT*, I
END DO
END
```

Nếu chạy chương trình này bạn sẽ nhận được kết quả trên màn hình là:

```
1
2
3
4
5
```

Ví dụ 2:

```
PRINT*, (I, I = 1, 5)
END
```

Trong trường hợp này bạn sẽ nhận được kết quả là:

```
    1     2     3     4     5
```

Lệnh PRINT* trong ví dụ 2 cho phép in 5 giá trị của I, với I tăng dần từ 1 đến 5. Khác với ví dụ 1, trong đó lệnh PRINT* được thực hiện 5 lần, mỗi lần in một bản ghi, nên kết quả nhận được là mỗi số in trên một dòng, ở đây, lệnh PRINT* chỉ thực hiện một lần, tức là chỉ in một bản ghi, nên các giá trị đều nằm trên một dòng. Người ta gọi vòng lặp in các giá trị của I trong lệnh PRINT* ở ví dụ 2 là chu trình DO ngầm. Loại chu trình DO này được sử dụng rất nhiều, nhất là trong việc kết xuất dữ liệu. Ví dụ, chương trình sau đây cho phép in ra trên 10 dòng 100 số nguyên dương đầu tiên theo thứ tự tăng dần, mỗi dòng 10 số.

```
DO I=1,91,10
```

```
   PRINT '(10I4)',(j,j=i,i+9)
ENDDO
END
```

3.4 Định dạng dữ liệu bằng lệnh FORMAT

Để đọc vào hoặc kết xuất dữ liệu có qui cách bạn có thể sử dụng lệnh định dạng FORMAT. Cú pháp câu lệnh như sau.

m FORMAT (*Mô tả định dạng*)

Trong đó **m** là nhãn câu lệnh, *Mô tả định dạng* là những qui ước để đọc/ghi dữ liệu theo qui tắc nhất định như đã dẫn ra trong bảng dưới đây.

Mô tả định dạng	ý nghĩa	Ví dụ
Iw[.m]	Đọc/in một số nguyên	I5, I5.5, 4I6
Bw[.m]	Đọc/in một số nhị phân	B4
Ow[.m]	Đọc/in một số cơ số 8	O5
Zw[.m]	Đọc/in một số cơ số 16	Z10.3
Fw.d	Đọc/in một số thực dấu phẩy tĩnh	F10.3, F5.0, 5F7.2
Ew.d	Đọc/in một số thực dấu phẩy động	E14.7, 5E10.6
Dw.d	Đọc/in một số thực độ chính xác gấp đôi	D14.7, 5D10.6
A[w]	Đọc/in một biến ký tự	A, A1, A20, 4A7
Lw	Đọc/in một biến lôgic	L4, 3L5
nX	Bỏ qua n ký tự	1X, 5X
/	Xuống dòng	2x, 5F10.3 / 2x,7F10.3
\ hoặc $	Giữ trên cùng một bản ghi	A\
Xâu KTự	In một xâu ký tự (đặt trong cặp dấu nháy)	‘Dong tren’/’Dong duoi’

Trong đó: **w** là độ rộng trường, **d** là số chữ số sau dấu chấm thập phân, **m** là số ký tự mà một số nguyên chiếm, kể cả chữ số 0 đứng trước, **n** là số ký tự bỏ qua.

Ví dụ:

```
INTEGER N, M
REAL X, Y, Z
PRINT 10, ' Cho hai so nguyen: '
10 FORMAT (A\)  ! Viet xong, giu con tro tren cung dong
READ (*, *) N, M
WRITE (*, '(A\)')' Cho ba so thuc: '
READ (*, *) X, Y, Z
WRITE (*, 20) N, M, X, Y, Z
20 FORMAT (20X, 'Cac so vua nhap la : '/2x,&
           &' Cac so nguyen: ',' N=', I6,&
           &' M=',I6/2x,' Cac so thuc: ',2x,&
```

```
            &' X=',F6.1,' Y=',F6.1,' Z=', F6.1)
END
```

Như đã thấy qua ví dụ trên, định dạng FORMAT có thể được mô tả ngay trong các câu lệnh READ và WRITE hoặc PRINT. Chẳng hạn, các câu lệnh

```
WRITE (*, 30) X, Y, Z
30 FORMAT (3X, 2F10.3, E12.5)
```

tương đương với câu lệnh

```
WRITE (*, '(3X, 2F10.3, E12.5)') X, Y, Z
```

Khi đọc dữ vào, bạn có thể dùng định dạng tự do như trong các ví dụ trên đây. Tuy nhiên bạn cũng có thể dùng định dạng có qui cách. Ví dụ, các trường hợp sau đây sẽ cho kết quả như nhau.

Giả sử bạn muốn nhập vào ba số x=12.3, y=23.45, z=123.4. Với câu lệnh

```
READ (*, *) X, Y, Z
```

Bạn chỉ cần gõ vào:

```
12.3  23.45  123.4
```
(Các số cách nhau bởi các dấu cách)

Nhưng nếu bạn viết:

```
READ (*,'(3F5.2)') X, Y, Z
```

bạn có thể gõ vào:

```
012300234512340
```

(Các số là các nhóm gồm 5 chữ số với 2 chữ số sau dấu chấm thập phân)

3.5 Chu trình lặp không xác định

Chu trình DO đã xét trước đây chính là chu trình lặp với số bước lặp được xác định bởi các tham số *TriDau, TriCuoi* và *Buoc*. Trong nhiều bài toán, số bước tính toán cần lặp đi lặp lại không thể xác định được một cách cụ thể, mà là được xác định thông qua một điều kiện cho trước nào đó. Cấu trúc lặp này được gọi là lặp không xác định.

3.5.1 Lặp không xác định bằng việc kết hợp IF và GOTO

Có thể tạo ra chu trình lặp không xác định bằng việc kết hợp IF và GOTO như sau.

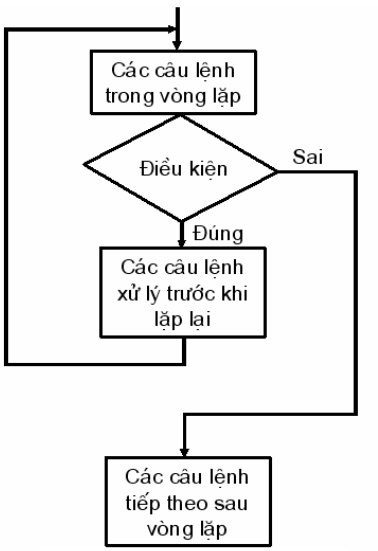
```
m Câu lệnh đầu vòng lặp
  Các câu lệnh tiếp theo trong vòng lặp
  IF (BThuc Logic) GOTO m
```

hoặc:

```
m Câu lệnh đầu vòng lặp
  Các câu lệnh tiếp theo trong vòng lặp
  IF (BThuc Logic) THEN
     Các câu lệnh xử lý trớc khi lặp lại
```

```
      GOTO m
   END IF
```

Trong đó **m** là nhãn câu lệnh đầu tiên của quá trình cần lặp; *BThuc Logic* là điều kiện để lặp lại quá trình. Nếu *BThuc Logic* nhận giá trị .TRUE. thì chương trình sẽ chuyển điều khiển đến câu lệnh đầu vòng lặp, ngược lại, nếu *BThuc Logic* nhận giá trị .FALSE. thì quá trình lặp sẽ kết thúc. Sơ đồ khối mô tả tác động của chu trình này được cho trên hình 3.1.



Hình 3.1

Ví dụ: Viết chương trình nhập vào một số dương không vượt quá 10. Với yêu cầu này, chương trình phải bảo đảm điều kiện, chừng nào số nhập vào không phải là một *số dương không vượt quá 10* thì phải nhập lại. Như vậy, điều kiện quay lại vòng lặp ở đây là số nhập vào hoặc là số âm hoặc là số lớn hơn 10. Ta có qui trình sau.

1) Nhập vào một số X

2) Kiểm tra số X:

– Nếu X<0 hoặc X>10 thì

+ Thông báo lỗi

+ Quay lại bước 1)

– Nếu 0<X<=10: Kết thúc

Mã nguồn chương trình như sau.

```
REAL X
10 PRINT '(A\)',' CHO MOT SO: '
   READ*, X
   IF (X.LT.0.OR.X.GT.10) THEN
      PRINT*,' SO VUA NHAP=',X
      PRINT*,' SAI ! NHAP LAI !'
      PRINT*
      GOTO 10
   END IF
PRINT*
PRINT*,' DUNG ROI! SO VUA NHAP=',X
END
```

3.5.2 Cấu trúc DO và EXIT

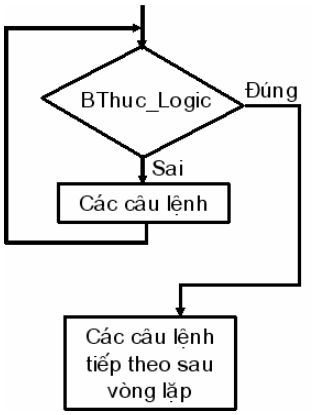
Đây cũng là dạng chu trình lặp không xác định. Cú pháp cấu trúc như sau.

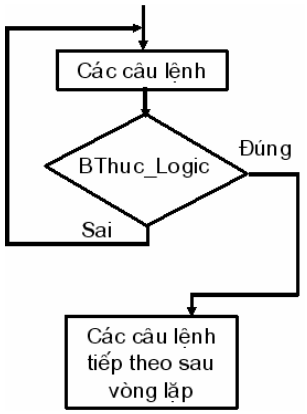
```
[TenChuTrinh:] DO
    IF (BThuc Logic) EXIT
    Các câu lệnh
END DO [TenChuTrinh]
```

Hoặc

```
[TenChuTrinh:] DO
    Các câu lệnh
    IF (BThuc Logic) EXIT
END DO [TenChuTrinh]
```

Sơ đồ khối mô tả tác động của cấu trúc được cho trên hình 3.2. Bạn cần hết sức chú ý sự khác nhau giữa hai cấu trúc này. Đối với cấu trúc thứ nhất, vì *BThuc Logic* được xác định trước khi *Các câu lệnh* được thực hiện, nên có thể *Các câu lệnh* sẽ không được thực hiện một lần nào nếu ngay từ đầu *BThuc Logic* nhận giá trị .TRUE. Trong khi đó, ở cấu trúc thứ hai, *Các câu lệnh* được thực hiện ít nhất một lần.





Ví dụ, hãy làm lại ví dụ ở mục 3.5.2 khi sử dụng hai cấu trúc DO và EXIT.

Với cấu trúc thứ nhất:

```
REAL X
X = -999.  ! Khoi tao X
CauTruc1: DO
   IF (X.GT.0.AND.X.LE.10) EXIT
   PRINT '(A\)','CHO MOT SO: '
   READ*, X
   IF (X.LT.0.OR.X.GT.10) THEN
      PRINT*,' SO VUA NHAP=',X
      PRINT*,' SAI ! NHAP LAI !'
      PRINT*
   END IF
END DO CauTruc1
PRINT*
PRINT*,' DUNG ROI! SO VUA NHAP=',X
END
```

Bây giờ bạn hãy thay câu lệnh

```
X = -999.
```

bởi câu lệnh trong đó 0 < X <= 10, chẳng hạn

```
X = 5.
```

và chạy lại chương trình, bạn sẽ nhận được kết quả bất ngờ đấy. Bạn hãy giải thích tại sao.

Với cấu trúc thứ hai:

```
REAL X
CauTruc2: DO
   PRINT '(A\)','CHO MOT SO: '
   READ*, X
   IF (X.GT.0.AND.X.LE.10) EXIT
   PRINT*,' SO VUA NHAP=',X
   PRINT*,' SAI ! NHAP LAI !'
   PRINT*
END DO CauTruc2
PRINT*
PRINT*,' DUNG ROI! SO VUA NHAP=',X
END
```

Trong cấu trúc này bạn không cần khởi tạo giá trị của X. Tại sao?

3.5.3 Cấu trúc DO WHILE…END DO

```
DO WHILE (BThuc Logic)
    Các câu lệnh
END DO
```

Cấu trúc này hoàn toàn tương đương với cấu trúc

```
[TenChuTrinh:] DO
    IF (BThuc Logic) EXIT
    Các câu lệnh
END DO [TenChuTrinh]
```

Để minh họa cho cách sử dụng cấu trúc này ta hãy làm lại ví dụ ở mục trước. Mã nguồn chương trình như sau.

```
REAL X
X = -999.  ! Khoi tao X
DO WHILE (X.LT.0.OR.X.GT.10)  ! Dieu kien lap lai
   PRINT*,' SAI ! '
   PRINT*
   PRINT '(A\)',' CHO MOT SO: '
   READ*, X
END DO
PRINT*
PRINT*,' DUNG ROI! SO VUA NHAP=',X
END
```

Nếu giá trị khởi tạo của X thỏa mãn điều kiện 0<X<=10 thì *Các câu lệnh* nằm giữa DO WHILE và END DO sẽ không bao giờ được thực hiện. Đó là điều bạn phải luôn luôn ghi nhớ.

3.5.4 Lệnh CYCLE

Cú pháp:

```
CYCLE [Tên Chu Trình]
```

Lệnh CYCLE nằm trong các chu trình lặp DO hoặc DO WHILE, có tác dụng bỏ qua các câu lệnh trong vòng lặp nằm sau CYCLE và chuyển điều khiển về khối kiểm tra điều kiện lặp lại của chu trình có tên là *Tên Chu Trình*.

Lệnh CYCLE có thể nằm trong các chu trình lồng nhau. Nếu không chỉ ra *Tên Chu Trình* thì CYCLE chỉ có tác động đối với chu trình lặp trong nhất chứa nó.

Ví dụ:

```
INTEGER I, N
PARAMETER (N = 10)
LapDO: DO I=1,N
          print*,'Chi so vong lap DO: ',i
       IF (i>3) CYCLE LapDO
          print*,'Lan duoc LAP boi DO:',i
END DO LapDO
END
```

Trong ví dụ trên, những câu lệnh nằm giữa hai dòng lệnh DO và IF sẽ được thực hiện N lần (I=1,N), nhưng các câu lệnh nằm sau câu lệnh IF cho đến hết chu trình DO chỉ được thực hiện chừng nào biểu thức `(i>3)` còn nhận giá trị .FALSE. Câu lệnh IF trong trường hợp này qui định khi nào thì lệnh CYCLE được gọi tới, và khi nó được gọi, quá trình lặp sẽ bỏ qua những câu lệnh sau đó, chỉ số lặp được tăng lên, điều khiển được chuyển về đầu vòng lặp.

Trên thực tế có thể kết hợp giữa CYCLE và EXIT trong các chu trình lặp phức tạp hơn. Bạn hãy khảo sát kỹ ví dụ sau đây để hiểu rõ hơn.

```
INTEGER i,j,k,n
PARAMETER (N = 10)
write (*,'(/A, I2)')' Dieu khien lap khi su dung &
                 & CYCLE va EXIT, N = ', N
write (*,900)
Vong1: DO i = 1, n
    if (i.gt.3) EXIT Vong1
    write (*,910) i
    Vong2: DO j = 1, n
        if (j.gt.2) CYCLE Vong2
        if (i.eq.2.and.j.gt.1) EXIT Vong2
        write (*,920) j
        Vong3: DO k = 1, n
            if (k.gt.2) CYCLE Vong3
            if (i.eq.1.and.j.gt.1) EXIT Vong2
            write (*,930) k
        END DO Vong3
    END DO Vong2
END DO Vong1
WRITE (*,'(/A)')  '  Hoan tat cac chu trinh.'
900   FORMAT(/'  Vong:     1         2         3 ')
910   FORMAT(11x, i2)
920   FORMAT(21x, i2)
930   FORMAT(31x, i2)
END
```

3.5.5 Một số ví dụ về chu trình lặp không xác định

a. Tính số PI theo công thức $\frac{\pi}{4} = 1 - \frac{1}{3} + \frac{1}{5} - \frac{1}{7} + ...$

Số hạng tổng quát của chuỗi là $\frac{(-1)^{n-1}}{2n-1}$, n=1,2,… Vậy ta có các bước tính sau.

1) Khởi tạo N=1, Tmp=0, EP=0.0001 [,SS=1.0]
2) Tính Sn = Tmp + (–1)**(N – 1) / (2*N – 1) (Số hạng thứ n)
   (Tổng tích lũy đến số hạng thứ n)
3) Tính Sai số tương đối SS = ABS((Sn-Tmp)/Sn)
4) So sánh SS với EP:
   - Nếu SS >= EP:
     - Lưu giá trị của Sn vào Tmp
     - Quay lại bước 2)
   - Nếu SS < EP:
     - Tính PI = Sn * 4
     - In kết quả và Kết thúc chương trình

Sau đây là lời chương trình viết bằng các cách khác nhau khi sử dụng các cấu trúc lặp đã trình bày trong các mục trước. Cũng cần lưu ý trước rằng, các chương trình ở đây chỉ nhằm mục đích giải thích về khía cạnh lập trình mà chưa chú ý đến tính tối ưu của chương trình. Sau khi bạn đã làm chủ được ngôn ngữ lập trình, bạn có thể thay đổi hoặc viết lại cho tốt hơn.

```
PROGRAM TINHPI1 ! Cach 1: IF & GOTO
REAL EPS, SS, PI, TMP
INTEGER N
EPS=0.0001
TMP=0.0
N=1
100 PI=TMP+(-1.0)**(N-1)/FLOAT(2*N-1)
       PRINT*,'Vong lap thu ',N,' Sai so=',SS
   SS=ABS((PI-TMP)/PI)
   IF (SS.GE.EPS) THEN
      TMP = PI
      N=N+1
      GOTO 100
   ELSE
      PI=PI*4.0
      WRITE(*,300)PI
 300 FORMAT(4X,' PI = ',F10.4)
   END IF
END
```

Hoặc

```
PROGRAM TINHPI2 ! CACH 2: DO & EXIT
REAL EPS, SS, PI, TMP
```

```
INTEGER N
EPS=0.0001
TMP=0.0
N=1
DO
   PI=TMP+(-1.0)**(N-1)/FLOAT(2*N-1)
   SS=ABS((PI-TMP)/PI)
       PRINT*,'Vong lap thu ',N,' Sai so=',SS
   IF (SS.LT.EPS) EXIT
      TMP = PI
      N=N+1
END DO
PI=PI*4.0
WRITE(*,300)PI
300  FORMAT(4X,' PI = ',F10.4)
END
```

Hoặc

```
PROGRAM TINHPI3 ! CACH 3: DO WHILE
REAL EPS, SS, PI, TMP
INTEGER N
EPS=0.0001
TMP=0.0
N=1
SS=1.0
DO WHILE (SS.GE.EPS)
   PI=TMP+(-1.0)**(N-1)/FLOAT(2*N-1)
   SS=ABS((PI-TMP)/PI)
       PRINT*,'Vong lap thu ',N,' Sai so=',SS
   TMP = PI
   N=N+1
END DO
PI=PI*4.0
WRITE(*,300)PI
300  FORMAT(4X,' PI = ',F10.4)
END
```

b. Tìm số nguyên dương lớn nhất **n** thỏa mãn điều kiện $3n^3 - 212n<10$

```
INTEGER N
N=0
DO WHILE (3*N**3 - 212*N < 10)
       PRINT*,'N = ',N,'  A=',3*N**3 - 212*N
   N = N + 1
END DO
END
```

CHƯƠNG 4. CHƯƠNG TRÌNH CON (SUBROUTINE VÀ FUNCTION) VÀ MODUL

4.1 Khái niệm

Trong lập trình, nhất là đối với những bài toán lớn, các chương trình thường bao gồm nhiều bộ phận khác nhau, trong đó có những bộ phận thường được sử dụng lặp đi lặp lại nhiều lần. Ngoài ra, những đoạn chương trình này có thể được sử dụng cho các chương trình khác. Việc viết một chương trình như vậy sẽ làm cho bạn cảm thấy nhàm chán và không hiệu quả, thậm chí làm cho chương trình trở nên rối rắm hơn. Để tổ chức một chương trình gọn gàng, dễ khai thác, Fortran cho phép phân mảnh chương trình và tạo thành các chương trình con.

Có hai khái niệm chương trình con là thủ tục (SUBROUTINE) và hàm (FUNCTION). Các chương trình con cũng có thể chia thành hai loại là chương trình con trong và chương trình con ngoài. Bạn cũng có thể chọn ra những chương trình con trong số các chương trình con để tạo ra một thư viện riêng cho mình. Tập hợp các chương trình con này có thể được gọi là modul. Các chương trình chính, chương trình con ngoài và các modul được gọi là các đơn vị chương trình (program unit). Về nguyên tắc, các chương trình con trong sẽ nằm trong các đơn vị chương trình khác và được biên dịch cùng với đơn vị chương trình mà nó phụ thuộc, trong khi các chương ngoài có thể được biên dịch một cách độc lập. Cái khác nhau cơ bản giữa chương trình con trong và chương trình con ngoài là ở chỗ, trong khi các chương trình con trong có thể sử dụng tên và những khai báo của đơn vị chương trình quản lý nó, thì các chương trình con ngoài do không được phép nằm trong đơn vị chương trình khác nên không có tính chất đó.

4.2 Thư viện các hàm trong

Trước đây, trong một số ví dụ, chúng ta đã sử dụng các hàm thư viện của Fortran, như hàm tính căn bậc hai, hàm tính cosin của một góc,… Để thuận tiện trong cho bạn sử dụng sau này, ở đây sẽ dẫn ra những hàm thông dụng nhất trong thư viện các chương trình con của Fortran.

ABS(A)**:** Tính trị tuyệt đối của số A (nguyên, thực hoặc phức).

ACOS(X)**:** Tính Arccos (arc cosine) của X.

AIMAG(Z)**:** Tính phần ảo của số phức Z

AINT(A [,KIND])**:** Lấy phần nguyên của số thực A, ví dụ AINT(3.9) là 3.0.

ANINT(A [,KIND])**:** Số nguyên gần nhất với số A, ví dụ ANINT(3.0) là 4.0.

ASIN(X)**:** Tính Arcsin (arc cosine) của X

ATAN(X)**:** Tính Arctang (arc tangent) của X, phạm vi từ –pi/2 đến pi/2.

ATAN2(Y, X)**:** Tính Atctang của tỷ số Y/X, phạm vi từ –pi đến pi.

CEILING(A)**:** Số nguyên nhỏ nhất không nhỏ hơn A.

CMPLX(X [,Y] [,KIND])**:** Chuyển X hoặc (X, Y) sang kiểu số phức.

CONJG(Z)**:** Liên hợp phức của số phức Z.

COS(X)**:** cosine của X

COSH(X)**:** cosine hyperbol của X

DIM(X, Y)**:** max(X-Y, 0).

EXP(X)**:** Hàm e^x

FLOOR(A)**:** Số nguyên lớn nhất không vượt quá A, ví dụ FLOOR(–3.9) là –4.

INT(A [,KIND])**:** Chặt cụt phần thập phân của A và đổi sang kiểu số nguyên

LOG(X)**:** Logarit cơ số tự nhiên

LOG10(X)**:** Logarit cơ số 10

MAX(A1, A2 [,A3,...])**:** Lớn nhất của các số

MIN(A1, A2 [,A3,...])**:** Nhỏ nhất của các số

MOD(A, P)**:** Phần dư của A modulo P, tức A-INT(A/P)*P. Ví dụ MOD(2.2, 2.0) là 0.2.

NINT(A [,KIND])**:** Số nguyên gần nhất với A.

REAL(A [,KIND])**:** Đổi A sang kiểu số thực

SIGN(A, B)**:** Trị tuyệt đối của A nhân với dấu của B.

SIN(A)**:** sine của A.

SINH(A)**:** sine hyperbol của A

SQRT(A)**:** Căn bậc hai của A

TAN(A)**:** tang của A

TANH(A)**:** tang hyperbol của A

ACHAR(I)**:** với I trong khoảng 0–127

CHAR(I [,KIND])**:** Ký tự ASCII có mã I với KIND cho trước

IACHAR(C)**:** Mã ASCII của ký tự C

ICHAR(C)**:** Mã ASCII của ký tự C

INDEX(STRING, SUBSTRING [BACK])**:** Vị trí bắt đầu của xâu con SUBSTRING trong xâu STRING, hoặc =0 nếu không xuất hiện. Ví trí này có thể là ký tự đầu tiên hoặc cuối cùng của xâu con tùy thuộc tham số BACK có (TRUE) hay không (FALSE)

LEN_TRIM(STRING): Độ dài của xâu STRING không tính các dấu cách bên phải

LEN(STRING)**:** Số ký tự trong xâu STRING nếu vô hướng, hoặc số phần tử trong mảng STRING nếu là mảng

TRIM(STRING)**:** Trả về xâu STRING (vô hướng) đã loại bỏ các ký tự trống bên phải.

4.3 Các chương trình con trong

4.3.1 Hàm trong (FUNCTION)

Hàm có thể được khai báo như sau.

```
[KiểuDL][RECURSIVE] FUNCTION funcname ([Các đối số])
[RESULT (result-name) ]
      [Các dòng lệnh khai báo]
      [Các dòng lệnh thực hiện]
      [funcname = ...]
    END FUNCTION [funcname]
```

Trong đó:

KiểuDL là kiểu dữ liệu mà hàm sẽ trả về. Bạn có thể bỏ qua tùy chọn này nếu bạn sử dụng tùy chọn *RESULT*.

Funcname là tên hàm

Các đối số là danh sách các đối số hình thức, liệt kê cách nhau bởi dấu phẩy.

result–name là kết quả trả về của hàm. Nếu bạn sử dụng tùy chọn này thì câu lệnh `funcname = ...` không được phép xuất hiện. Ngược lại, nếu bạn không sử dụng tùy chọn RESULT thì phải có dòng lệnh `funcname = ...`

Hàm có thể được gọi tới bằng cách hoặc gán giá trị hàm cho biến, hoặc hàm tham gia vào biểu thức tính:

TenBien = funcname ([*Các đối số*])

Ví dụ:

```
Cx = COS (x)
Pi = 4.0 * ATAN (1.0)
```

Khi xây dựng hàm, *Các đối số* là những đối số hình thức, nhưng khi gọi hàm thì *Các đối số* phải được thay vào đó là danh sách đối số thực. Ví dụ: Hàm `YNew` được xây dựng với ba đối số hình thức X, Y, A:

```
FUNCTION YNew ( X, Y, A )
   ...
   YNew = ...
END FUNCTION YNew
```

Khi hàm này được gọi tới, các ***đối số hình thức*** được thay bởi những ***đối số thực***:

```
YNew( U, V, Pi/2 )
```

Các đối số hình thức và đối số thực phải tương ứng **1-1.**

4.3.2 Thủ tục trong (SUBROUTINE)

Về cơ bản cú pháp khai báo thủ tục giống với khai báo hàm. Chỉ có một số khác biệt sau:

– Không có giá trị nào được liên kết với tên thủ tục

– Để gọi tới thủ tục phải dùng từ khóa CALL

– Từ khóa SUBROUTINE được sử dụng để định nghĩa thủ tục thay cho từ khóa FUNCTION

– Nếu hàm không có đối số sẽ được gọi tới bằng cách thêm vào sau tên hàm cặp dấu ngoặc đơn rỗng () (Ví dụ: `MyFunction()`), thì đối với thủ tục không có đối số, khi gọi tới sẽ không cần cặp dấu ngoặc đơn này (Ví dụ: `CALL MySubroutine`).

Cú pháp khai báo thủ tục như sau.

```
SUBROUTINE Subname [( Các đối số )]
      [Các câu lệnh khai báo]
      [Các câu lệnh thực hiện]
END SUBROUTINE [Subname]
```

Trong đó:

Các đối số là danh sách đối số hình thức, được liệt kê cách nhau bởi dấu phẩy.

Lời gọi thủ tục:

```
CALL Subname [( Các đối số thực )]
```

Trong đó danh sách các đối số hình thức và danh sách các đối số thực phải tương ứng 1–1 với nhau.

Chú ý:

- Hàm là một chương trình con chỉ trả về duy nhất một giá trị: Giá trị của hàm ứng với các đối số
- Trong định nghĩa màm (FUNCTION), trước khi trả về chương trình gọi, giá trị của hàm luôn được xác định bởi một câu lệnh gán hoặc cho *Tên hàm* hoặc cho biến trong tùy chọn *RESULT*.
- Nói chung hàm (và cả thủ tục) kết thúc ở câu lệnh END cuối cùng. Tuy nhiên cũng có thể sử dụng câu lệnh RETURN để trả về chương trình gọi. Khi gặp câu lệnh RETURN chương trình con sẽ được giải phóng và quay về chương trình gọi.

4.4 Câu lệnh CONTAINS

Câu lệnh CONTAINS là câu lệnh không thực hiện, dùng để phân cách thân chương trình chính với các chương trình con trong thuộc nó. Các chương trình con trong được sắp xếp ngay sau câu lệnh CONTAINS và trước từ khóa END của chương trình chính. Bố cục tổng quát của chương trình có dạng như sau.

```
PROGRAM Progname
  [Các dòng lệnh khai báo]
  [Các dòng lệnh thực hiện]
[CONTAINS
  Các chương trình con trong ]
```

```
END [PROGRAM [Progname]]
```

Ví dụ:

```
PROGRAM OUTER
REAL A(10)
. . .
CALL INNER (A)
CONTAINS
   SUBROUTINE INNER (B)
   REAL B(10)
   . . .
   END SUBROUTINE INNER
END PROGRAM OUTER
```

4.5 Một số ví dụ về chương trình con trong

4.5.1 Tính tích phân xác định $I = \int_a^b f(x)dx$ *bằng phương pháp hình thang.*

Giả sử $f(x) = \frac{1}{\sqrt{2\pi}} e^{-\frac{1}{2}x^2}$. Ta có sơ đồ tính và lời chương trình như sau.

B1) Cho giá trị của a, b (a<b), Epsilon

B2) Khởi tạo N=0, S1=0

B3) Tăng số khoảng chia lên 1: N = N+1

B4) Chia (a,b) làm N khoảng, cự ly mỗi khoảng DelX = (b-a)/N

B5) Tính tổng diện tích N hình thang và gán cho S2:

1) Gán S2=0
2) Lặp lại N lần, mỗi lần ứng với một hình thang: j=1,N
   a) Xác định x1, x2, f(x1), f(x2): x1=a+(j-1)*DelX; x2=x1+DelX
   b) Tính diện tích hình thang thứ j: Tmp=(f(x1)+f(x2)) * DelX / 2
   c) Cộng dồn diện tích hình thang vừa tính vào S2: S2=S2+Tmp

B6) Tính sai số: SS=ABS((S2-S1)/S2)

B7) Kiểm tra điều kiện kết thúc:

1) Nếu SS < Epsilon: In kết quả và kết thúc chương trình
2) Nếu SS >= Epsilon:
   a) Lưu giá trị S2 vào S1
   b) Lặp lại từ bước B3)

```
PROGRAM TICHPHAN
INTEGER N, J
REAL A,B,S1,S2,DELX
REAL X, F1,F2, SS,EP, HSO
PARAMETER (EP=1.E-4, A=0., B=3.)
```

```
N=0
S1=0
DO
                          N=N+1
                  DELX = (B-A)/REAL(N)
                           S2=0
                         DO J=1,N
                                          X = A + (J-1)*DELX
                                            F1= F(X)
                                        X = X + DELX
                                            F2= F(X)
                                    S2= S2 + (F1+F2)*DELX/2.0
                          END DO
                  SS = ABS((S2-S1)/S2)
                   IF (SS < EP ) EXIT
                         S1 = S2
              PRINT*,'SO HINH THANG =',N
END DO
PRINT '('' GIA TRI TP = '',F10.4)', S2
CONTAINS
   FUNCTION F(X) RESULT (F1)
      F1=1.0/SQRT(2.0*(4.0*ATAN(1.)))*EXP(-0.5*X*X)
   END FUNCTION F
END
```

Trong chương trình trên F(X) là hàm trong với đối số hình thức là X.

4.5.2 Giải phương trình f(x) = 0 bằng phương pháp lặp Newton.

Nội dung phương pháp này như sau.

1) Khởi tạo x bằng một giá trị ban đầu nào đó

2) Gán x bởi x–f(x)/f'(x), trong đó f'(x) là đạo hàm bậc một của f(x)

3) Tính và kiểm tra điều kiện f(x) ~ 0

– Nếu chưa thỏa mãn thì quay lại bước 2)

– Nếu thỏa mãn thì in kết quả và kết thúc chương trình.

Giả sử cho $f(x) = x^3 + x - 3$. Khi đó $f'(x) = 3x^2 + 1$. Ta chọn giá trị khởi tạo của x là 2. Khi $f(x) < 10^{-6}$ hoặc sau 20 lần lặp thì x được xem là nghiệm của phương trình. Lời chương trình như sau.

```
PROGRAM Newton
! Giai PT f(x) = 0 bang PP Newton
IMPLICIT NONE
INTEGER :: Its    = 0   ! Dem lan lap
INTEGER :: MaxIts = 20  ! So lan lap cuc dai
LOGICAL :: Converged = .false. ! Dieu kien hoi tu
REAL :: Eps = 1e-6   ! Sai so cho phep
REAL :: X = 2        ! Gia tri khoi tao
DO WHILE (.NOT. Converged .AND. Its < MaxIts)
  X = X - F(X) / DF(X)
  PRINT*, X, F(X)
  Its = Its + 1
```

```
  Converged = ABS( F(X) ) <= Eps
END DO

IF (Converged) THEN
  PRINT*, 'Hoi tu'
ELSE
  PRINT*, 'Phan ky'
END IF
CONTAINS
  FUNCTION F(X)
    REAL F, X
    F = X ** 3 + X - 3
  END FUNCTION F
  FUNCTION DF(X)
    REAL DF, X
    DF = 3 * X ** 2 + 1
  END FUNCTION DF
END PROGRAM Newton
```

4.5.3 In một dãy các ký tự giống nhau

```
IMPLICIT NONE
CALL DayKyTu( 5, 65 )  ! 5 chu A lien tuc
CONTAINS
  SUBROUTINE DayKyTu ( Num, Symbol )
    INTEGER I, Num, Symbol
    CHARACTER*80 Line
    DO I = 1, Num
       Line(I:I) = ACHAR( Symbol )
    END DO
    PRINT*, Line
  END SUBROUTINE
END
```

4.5.4 Tính tổ hợp chập $C_n^k = \frac{n!}{k!(n-k)!}$.

Để tính tổ hợp chập cần phải xây dựng hàm tính giai thừa. Chương trình sau tính và in tổ hợp chập từ 0 đến 10 của 10.

```
PROGRAM TOHOPCHAP
INTEGER I
DO I = 0, 10
  PRINT*, I, Fact(10)/(Fact(I)*Fact(10-I))
END DO

CONTAINS
  FUNCTION Fact ( N )
    INTEGER Fact, N, Temp , I
    Temp = 1
    DO I = 2, N
      Temp = I * Temp
    END DO
    Fact = Temp
```

```
  END FUNCTION
END
```

4.6 Biến toàn cục và biến địa phương

Hãy xét hai chương trình dưới đây, trong đó mục đích của các chương trình này là tính và in lần lượt giai thừa của các số từ 1 đến 10. Bạn hãy để ý đến sự khác nhau giữa chúng.

```
PROGRAM VER1
INTEGER I
DO I = 1, 10
  PRINT*, I, Fact(I)
END DO
CONTAINS
  FUNCTION Fact ( N )
    INTEGER Fact, N, Temp
    Temp = 1
    DO I = 2, N
      Temp = I * Temp
    END DO
    Fact = Temp
  END FUNCTION
END
```

và

```
PROGRAM VER2
INTEGER I
DO I = 1, 10
  PRINT*, I, Fact(I)
END DO
CONTAINS
  FUNCTION Fact ( N )
    INTEGER Fact, N, Temp, I
    Temp = 1
    DO I = 2, N
      Temp = I * Temp
    END DO
    Fact = Temp
  END FUNCTION
END
```

Khi lần lượt chạy các chương trình này bạn sẽ thấy bất ngờ, vì cả hai chương trình này viết gần như hoàn toàn giống nhau, nhưng kết quả lại rất khác nhau. Vì sao vậy? Vấn đề ở chỗ là sự có mặt của biến I trong câu lệnh khai báo của chương trình con *Fact*:

```
    INTEGER Fact, N, Temp, I
```

Vì chương trình con *Fact* là chương trình con trong, nên khi biến I không được khai báo, nó sẽ sử dụng biến đã được khai báo bởi chương trình chính điều khiển nó. Như vậy, ở chương trình VER1, biến I đồng thời bị điều khiển bởi cả chương trình chính lẫn chương trình con. Tác động của quá trình dùng chung biến I có thể được mô tả như sau.

– Khi trong chương trình chính (CTC) biến I=1, nó sẽ truyền tham số N=I=1 cho chương trình con (FACT), đồng thời trong FACT, biến I=2, 1 nên Fact=Tmp=1 (Vòng DO không thực hiện). Khi FACT trả về CTC thì I=2.

– Sau khi FACT trả về CTC thì biến được tăng lên I=I+1=2+1=3, và giá trị này lại truyền cho FACT, nên N=I=3, đồng thời trong FACT, I=2, 3 do đó Fact=1.2.3=6 (Ra khỏi vòng DO của FACT: I=N+1=3+1=4). Khi FACT trả về CTC thì I=4.

– Tiếp tục, trong CTC: I=I+1=4+1=5; khi truyền tham số cho FACT thì N=I=5, và trong FACT: I=2, 5 nên Fact=1.2.3.4.5=120 (Ra khỏi vòng DO của FACT: I=N+1=5+1=6). FACT lại trả về CTC giá trị của I=6.

Quá trình cứ tiếp diễn như vậy và giai thừa của các số chẵn không được tính cho đến khi xuất hiện lỗi do biến I bị loạn.

Nhưng nếu trong chương trình con ta khai báo thêm biến I thì quá trình tính toán diễn ra chính xác và chương trình kết thúc bình thường.

Biến I khai báo trong chương trình chính được gọi là biến toàn cục, còn biến I khai báo trong chương trình con là biến địa phương. Các chương trình con trong được phép tham chiếu đến các biến toàn cục khi các biến địa phương không được khai báo. Tuy nhiên, chương trình chính sẽ không tham chiếu được đến các biến địa phương khai báo ở các chương trình con trong.

4.7 Định nghĩa hàm bằng câu lệnh đơn

Hàm có thể được định nghĩa bằng cấu trúc hàm như đã thấy trong mục 4.3.1, nhưng hàm cũng có thể được định nghĩa bằng câu lệnh khai báo hàm. Ta hãy xét ví dụ sau đây.

```
PROGRAM BT_HAM1
REAL X
F(x) = 3*x**2 - 5*x + 2
Print*,' Cho gia tri cua X: '
Read*,x
Print '('' Gia tri ham F(x)='',F10.3)', F(x)
END
```

Trong chương trình trên, câu lệnh

```
F(x) = 3*x**2 - 5*x + 2
```

là một cách định nghĩa hàm F(x) và gọi là biểu thức hàm. Chương trình này tương đương với chương trình sau.

```
PROGRAM BT_HAM2
REAL X
Print*,' Cho gia tri cua X: '
Read*,x
Print '('' Gia tri ham F(x)='',F10.3)', F(x)
CONTAINS
   FUNCTION F(X)
     F = 3*x**2 - 5*x + 2
   END FUNCTION
END
```

4.8 Chương trình con ngoài

Các chương trình con trong là những chương trình con chỉ do một đơn vị chương trình (chẳng hạn, chương trình chính) kiểm soát, chúng khu trú giữa hai câu lệnh CONTAINS và END của đơn vị chương trình.

Các chương trình con tồn tại ở ngoài dưới dạng các file độc lập được gọi là các chương trình con ngoài. Chúng có thể được tham chiếu bởi nhiều chương trình khác nhau.

Các chương trình con ngoài cũng có thể có các chương trình con trong riêng của chúng. Nhưng các chương trình con trong lại ***không được phép*** chứa các chương trình con trong khác.

Cú pháp khai báo các chương trình con ngoài có thể có dạng.

Khai báo hàm:

```
FUNCTION funcname ([ Các đối số ])
  [Các câu lệnh khai báo]
  [Các câu lệnh thực hiện]
[CONTAINS
   Các chương trình con trong ]
END [FUNCTION [funcname] ]
```

Khai báo thủ tục:

```
SUBROUTINE Subname [( Các đối số )]
  [Các câu lệnh khai báo]
  [Các câu lệnh thực hiện]
[CONTAINS
  Các chương trình con trong]
END [SUBROUTINE [Subname] ]
```

4.8.1 Câu lệnh EXTERNAL

Để tránh nhầm lẫn trong việc sử dụng các chương trình thư viện của Fortran và chương trình con ngoài có tên trùng nhau, **nên** khai báo **tên** các chương trình con ngoài bằng câu lệnh EXTERNAL. Để minh họa ta hãy xét hai ví dụ sau đây.

Ví dụ 1: Chương trình này định nghĩa chương trình con ngoài COS(X) có tên trùng với hàm COS(X) của thư viện Fortran. Khi chạy chương trình bạn sẽ thấy chương trình con này không được gọi tới mà thay cho nó, hàm COS(X) của Fortran sẽ được gọi.

```
PROGRAM EXT1
REAL A
PRINT*,'Cho so A:'
READ*, A
A = COS( A )
PRINT*, A
END
```

```
FUNCTION COS( X )
COS = X + 5.
END FUNCTION
```

Ví dụ 2: Cũng chương trình trên đây, nhưng nếu bạn thêm vào câu lệnh

```
EXTERNAL COS
```

vào ngay phần khai báo của chương trình, kết quả là chương trình con ngoài sẽ được gọi tới.

```
PROGRAM EXT2
REAL A
EXTERNAL COS
PRINT*,'Cho so A:'
READ*, A
A = COS( A )
PRINT*, A
END
FUNCTION COS( X )
COS = X + 5.
END FUNCTION
```

4.8.2 Khai khối giao diện (INTERFACE BLOCK)

Ta đã thấy ở mục trên, trong nhiều trường hợp trình biên dịch sẽ không hiểu ý đồ của bạn khi bạn muốn sử dụng những chương trình con ngoài có cùng tên (có thể vô tình) với các chương trình con thư viện của Fortran. Để khắc phục tình trạng đó có thể sử dụng câu lệnh EXTERNAL. Câu lệnh này cung cấp cho trình biên dịch ***tên*** của chương trình con ngoài, cho phép nó tìm được và liên kết (LINK). Tuy nhiên, để trình biên dịch tạo ra lời gọi các chương trình con ngoài một cách chính xác, ngoài tên ra, nó cần phải biết chắc chắn những thông tin về chương trình con, như số biến và kiểu của biến,... Tập hợp những thông tin đó gọi là phần giao diện của chương trình con.

Đối với các chương trình con thư viện của Fortran, chương trình con trong và chương trình con modul, phần giao diện luôn được trình biên dịch hiểu. Nhưng khi trình biên dịch phát sinh lời gọi đến một chương trình con ngoài, những thông tin thuộc phần giao diện hoàn toàn chưa sẵn có, tức nó ở trạng thái ẩn (implicit), và trong nhiều trường hợp phức tạp (như các đối số tùy chọn hoặc các đối số từ khóa) đòi hỏi phải cung cấp những thông tin giao diện đầy đủ hơn. Do đó cần có khối giao diện.

Khai báo khối giao diện như sau.

```
INTERFACE
   Thân của khối giao diện
END INTERFACE
```

Trong đó *Thân của khối giao diện* nên được sao chép một cách chính xác phần đầu (header) của các chương trình con, những khai báo đối số và kết quả của chúng, và cả câu lệnh END của chúng.

Ví dụ:

```
IMPLICIT NONE
```

```
INTERFACE
  SUBROUTINE SWOP( X, Y )
    REAL X, Y
  END SUBROUTINE
END INTERFACE

REAL A, B
PRINT*,' CHO 2 SO: '
READ*, A, B
print*,A, B
CALL SWOP( A, B )
print*,A, B
END
SUBROUTINE SWOP( X, Y )
REAL X, Y
REAL TMP
   TMP = X
   X = Y
   Y = TMP
END SUBROUTINE
```

4.9 Modul

FORTRAN định nghĩa 3 khái niệm đơn vị chương trình (Program Unit) là: Chương trình chính, Chương trình con ngoài, và *modul*. Modul khác với các chương trình con ở 2 điểm quan trọng:

– Modul có thể chứa trong đó nhiều hơn một chương trình con (được gọi là các chương trình con modul);

– Modul có thể chứa những câu lệnh khai báo và đặc tả mà chúng có thể tham chiếu được đối với tất cả các đơn vị chương trình có sử dụng modul.

Các modul cũng có thể được biên dịch một cách độc lập.

Cấu trúc chung của modul có dạng như sau:

```
MODULE name
  [Các câu lệnh khai báo]
[CONTAINS
  Các chương trình con modul]
END [MODULE [name]]
```

Để sử dụng modul hãy dùng câu lệnh khai báo USE ngay đầu chương trình:

```
USE Tên_Các_Modul_được_sử_dụng
```

Như vậy, về cơ bản cấu trúc của modul giống như cấu trúc của chương trình con ngoài, ngoại trừ các từ khóa SUBROUTINE và FUNCTION. Modul cũng có thể có các chương trình con trong của chính nó. Modul cũng có thể sử dụng các modul khác. Vì modul có thể chứa các câu lệnh khai báo có thể truy cập đối với tất cả các đơn vị chương trình, nên các biến toàn cục có thể được khai báo theo cách này cho tất cả các chương trình sử dụng modul. Tính chất này rất hữu ích để tạo ra những

khai báo phức tạp, như các kiểu dữ liệu do người dùng định nghĩa,… Các khối giao diện cũng có thể được gộp vào trong các modul.

Ví dụ:

```
PROGRAM EXAMP
USE MyModul
IMPLICIT NONE
REAL A, B
PRINT*, ' Cho mot so: '
READ*, A
B = Pi             ! Khai bao tu Modul
CALL SWOP( A, B ) ! Khai bao tu Modul
PRINT*, A, B
END


MODULE MyModul
REAL Pi
PARAMETER (Pi = 3.1415927)
CONTAINS
  SUBROUTINE SWOP( X, Y )
    REAL Tmp, X, Y
    Tmp = X
    X = Y
    Y = Tmp
  END SUBROUTINE SWOP
END MODULE MyModul
```

4.10 Phép đệ qui

Trong nhiều trường phép đệ qui cho phép làm đơn giản hoá chương trình một cách đáng kể. Có thể xem phép đệ qui là một hàm hay một thủ tục có thể tham chiếu đến chính nó. Phép đệp qui bao gồm hai thành phần: Phần neo, trong đó tác động của hàm hay thủ tục được đặc tả cho một hay nhiều tham số, và phần đệ qui trong đó tác động cần được thực hiện cho giá trị hiện thời của tham số được xác định bằng các tác động hay giá trị được đinh nghĩa trước đó.

Ví dụ điển hình cho phép đệ qui là hàm hoặc thủ tục tính giai thừa. Xuất phát từ định nghĩa n! ta có: 0! = 1! = 1, với mọi n>0 thì n! = n.(n–1)!. Đây là một công thức truy hồi, tức là khi đã biết (n–1)! thì có thể tính được n!. Ta có hàm và thủ tục tính n! như sau:

```
RECURSIVE FUNCTION GIAITHUA1 ( N ) RESULT (Fact)
INTEGER Fact, N
IF( N == 0 .OR. N == 1 ) THEN
    Fact = 1
ELSE
    Fact = N * GIAITHUA1( N-1 )
END IF
END FUNCTION
```

Hoặc

```
RECURSIVE SUBROUTINE GIAITHUA2( F, N )
INTEGER F, N
```

```
IF (N == 0 .OR. N == 1) THEN
    F = 1
ELSE
    CALL GIAITHUA2( F, N-1 )
    F = N * F
END IF
END SUBROUTINE
```

Giả sử để tính 3!, ta gọi GIAITHUA1(3), hoặc GIAITHUA2(F,3). Lời gọi này sẽ tham chiếu đến GIAITHUA1(2) hoặc GIAITHUA2(F,2), rồi GIAITHUA1(2) hoặc GIAITHUA2(F,2) lại tham chiếu đến GIAITHUA1(1) hoặc GIAITHUA2(F,1) là phần neo (`IF (N == 0 .OR. N==1) THEN...`).

Một ví dụ khác, trung bình số học của một dãy số $x_1, x_2,..., x_n$ có thể được tính theo công thức: $\overline{x_n} = \frac{1}{n}\sum_{i=1}^{n} x_i$. Ta có thể biểu diễn công thức này dưới dạng khác: $\overline{x_n}$ = $((n-1).\overline{x_{n-1}} + x_n)/n$, trong đó $\overline{x_{n-1}}$ là trung bình của n–1 thành phần đầu của dãy. Phần neo có thể xác định bởi định nghĩa sau:

– Số thành phần n>0

– Nếu n=1 thì $\overline{x_n} = x_1$,

– Nếu n=2 thì $\overline{x_n} = (x_1 + x_2)/2$

Bạn hãy viết chương trình con đệ qui cho bài toán này như là một bài tập.

CHƯƠNG 5. MẢNG

5.1 Khái niệm về mảng trong FORTRAN

Có thể định nghĩa mảng là một tập hợp các phần tử có cùng kiểu dữ liệu, được sắp xếp theo một trật tự nhất định, trong đó mỗi phần tử được xác định bởi chỉ số và giá trị của chúng. Chỉ số của mỗi phần tử mảng được xem là "địa chỉ" của từng phần tử trong mảng mà nó được dùng để truy cập/tham chiếu đến phần tử của mảng. Mỗi phần tử của mảng được xác định bởi duy nhất một "địa chỉ" trong mảng. Kiểu dữ liệu của các phần tử mảng có thể là kiểu số hoặc không phải số. Mỗi mảng được xác định bởi tên mảng, số chiều, kích thước cực đại và cách sắp xếp các phần tử của mảng. Tên mảng còn gọi là tên biến mảng, hay ngắn gọn hơn là biến mảng.

Biến mảng là biến ít nhất có một chiều. Mảng có thể là mảng tĩnh hoặc mảng động. Nếu là mảng tĩnh thì vùng bộ nhớ dành lưu trữ mảng là cố định và nó không bị giải phóng chừng nào chương trình còn hiệu lực. Kích thước của mảng tĩnh không thể bị thay đổi trong quá trình chạy chương trình. Nếu mảng là mảng động, vùng bộ nhớ lưu trữ nó có thể được gán, thay đổi và giải phóng khi chương trình đang thực hiện.

Các con trỏ (POINTER) cũng là những biến động. Nếu con trỏ cũng là mảng thì kích thước của mỗi chiều cũng có thể bị thay đổi trong lúc chương trình chạy, giống như các mảng động. Các con trỏ có thể trỏ đến các biến mảng hoặc biến vô hướng.

5.2 Khai báo mảng

Để sử dụng mảng nhất thiết cần phải khai báo nó. Khi khai báo mảng cần phải chỉ ra tên và số chiều của nó, nhưng có thể chưa cần chỉ ra kích thước và cách sắp xếp các phần tử mảng. Có rất nhiều cách khai báo biến mảng. Sau đây sẽ liệt kê một số trường hợp ví dụ.

```
REAL A(10, 2, 3)        ! Mảng thực 3 chiều, sử dụng khai báo kiểu
DIMENSION A(10, 2, 3)   ! Mảng thực 3 chiều, sử dụng DIMENSION
ALLOCATABLE B(:, :)     ! Mảng thực 2 chiều với ALLOCATABLE
POINTER C(:, :, :)      ! Mảng thực 3 chiều với POINTER
REAL, DIMENSION (2, 5) :: D ! Kết hợp kiểu và DIMENSION
REAL,ALLOCATABLE :: E(:,:,:)! Kết hợp kiểu và  ALLOCATABLE
REAL, POINTER :: F(:,:)   ! Kết hợp kiểu và POINTER
```

Trong các ví dụ trên, mảng A(10, 2, 3) là mảng ba chiều gồm các phần tử có kiểu số thực loại 4 byte, kích thước cực đại của mảng là 10 x 2 x 3 = 60 phần tử, dung lượng bộ nhớ cấp phát cho mảng là 60 x 4 (byte) = 240 byte, cách sắp xếp các phần tử là 10 hàng, 2 cột và 3 lớp, địa chỉ các hàng, cột và lớp được đánh số từ 1 (hàng 1 đến hàng 10, cột 1 đến cột 2, lớp 1 đến lớp 3). Mảng B là mảng động 2 chiều, trong đó kích thước và cách sắp xếp các phần tử chưa được xác định. Mảng C là mảng thực ba chiều có kiểu con trỏ.

Cách đánh số địa chỉ các phần tử mảng cũng là một trong những đặc điểm hết sức quan trọng, vì nó quyết định cách truy cập đến các phần tử mảng. Chỉ số xác

định địa chỉ các phần tử mảng phụ thuộc vào giới hạn dưới và giới hạn trên dùng để mô tả cách sắp xếp các phần tử theo các chiều của mảng. Ví dụ, hai mảng

```
INTEGER M(10, 10, 10)
INTEGER K(-3:6, 4:13, 0:9)
```

đều có cùng kích thước (10 x 10 x 10), nhưng mảng M có chỉ số các phần tử mảng theo cả ba chiều biến thiên từ 1 đến 10 (giới hạn dưới bằng 1, giới hạn trên bằng 10), còn mảng K có chỉ số các phần tử mảng biến thiên theo chiều thứ nhất (hàng) là –3 đến 6, theo chiều thứ hai (cột) là 4 đến 13 và theo chiều thứ ba (lớp) là 0 đến 9. Như vậy, giới hạn dưới của chỉ số các phần tử của mảng K tương ứng là –3, 4 và 0, còn giới hạn trên là 6, 13 và 9. Các mảng được mô tả rõ ràng như vậy được gọi là các mảng có mô tả tường minh.

Đối với các mảng mô tả không tường minh, cách sắp xếp và đánh số địa chỉ các phần tử mảng thường được xác định trong lúc chương trình chạy hoặc sẽ được truyền qua tham số của các chương trình con. Ví dụ:

```
REAL X (4, 7, 9)
...
CALL SUB1(X)
CALL SUB2(X)
...
END
SUBROUTINE SUB1(A)
REAL A(:, :, :)
...
END SUBROUTINE SUB1
SUBROUTINE SUB2(B)
REAL B(3:, 0:, -2:)
...
END SUBROUTINE SUB2
```

Trong trường hợp này mảng A trong chương trình con SUB1 sẽ là:

```
A (4, 7, 9)
```

còn mảng B trong chương trình con SUB2 sẽ là:

```
B (3:6, 0:6, -2:6)
```

5.3 Lưu trữ mảng trong bộ nhớ và truy cập đến các phần tử mảng

Nguyên tắc lưu trữ mảng trong bộ nhớ của Fortran là lưu trữ dưới dạng vectơ, cho dù đó là mảng một chiều hay nhiều chiều. Đối với mảng một chiều, các phần tử mảng được sắp xếp theo thứ tự từ phần tử có địa chỉ mảng (chỉ số) nhỏ nhất đến phần tử có địa chỉ lớn nhất. Các phần tử của mảng hai chiều cũng được xếp thành một vectơ, trong đó các "đoạn" liên tiếp của vectơ này là các cột với chỉ số cột tăng dần. Các mảng ba chiều được xem là tập hợp các mảng hai chiều với số thứ tự của các mảng hai chiều này (số thứ tự lớp) chính là chỉ số thứ ba của mảng. Các mảng nhiều chiều hơn cũng được lưu trữ theo nguyên tắc này.

Các phần tử mảng được truy cập đến qua địa chỉ của chúng trong mảng. Để rõ hơn ta xét một số ví dụ sau.

Ví dụ 1: Mảng một chiều.

Giả sử bạn khai báo

```
REAL X(5), Y(0:5)
```

Khi đó các mảng X và Y được sắp xếp trong bộ nhớ như sau:

X(1)	X(2)	X(3)	X(4)	X(5)

Y(0)	Y(1)	Y(2)	Y(3)	Y(4)	Y(5)

Chương trình sau đây minh họa cách truy cập đến các phần tử của các mảng này.

```
REAL X(5), Y(0:5)
DO I=1,5
   X(I) = I*I          ! Gán giá trị cho các phần tử của X
   Y(I) = X(I) + I     ! Nhận giá trị các phần tử của X, tính toán và gán
                       ! cho các phần tử của Y
END DO
PRINT '(6F7.1))', (X(I), I=1,5)  ! In các phần tử của X
PRINT '(6F7.1))', (Y(I), I=0,5)  ! In các phần tử của Y
END
```

Ví dụ 2: Mảng hai chiều.

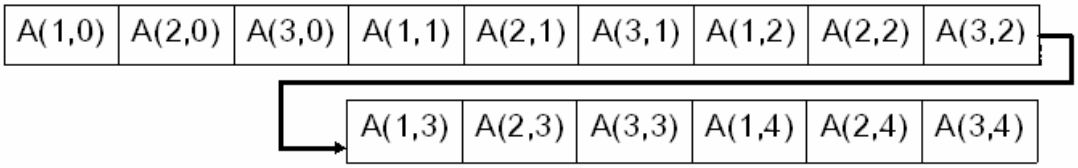
Giả sử bạn khai báo

```
PARAMETER (N=3, M=4)
INTEGER  A(N, 0:M)
```

Khi đó có thể hiểu mảng A như là một ma trận gồm 3 hàng, 5 cột:

$$A=\begin{pmatrix} A(1,0) & A(1,1) & A(1,2) & A(1,3) & A(1,4) \\ A(2,0) & A(2,1) & A(2,2) & A(2,3) & A(2,4) \\ A(3,0) & A(3,1) & A(3,2) & A(3,3) & A(3,4) \end{pmatrix}$$

A được lưu trữ trong bộ nhớ dưới dạng:

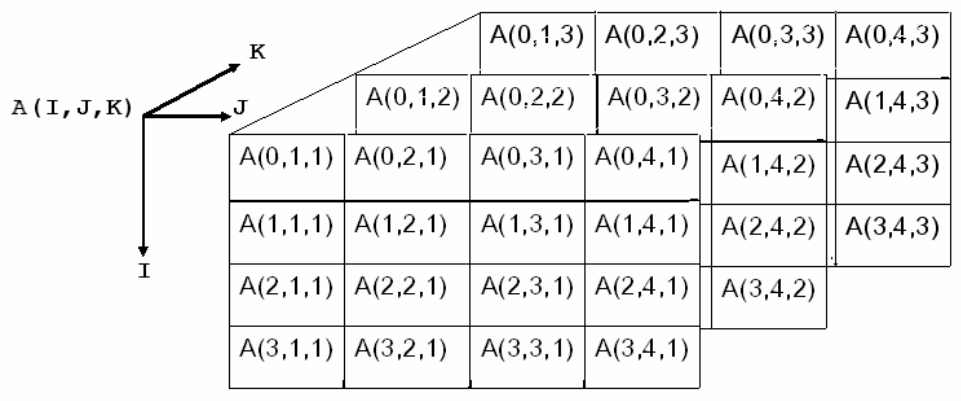


Ví dụ 3: Mảng ba chiều.

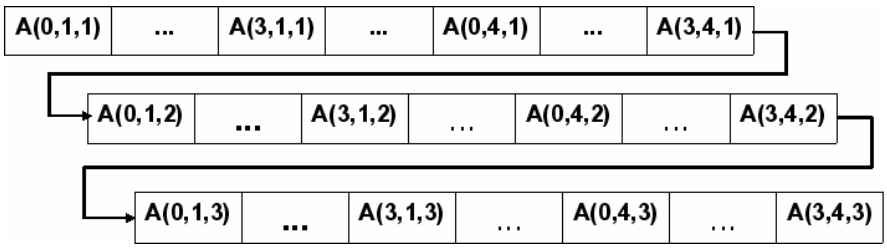
Giả sử mảng A được khai báo bởi

```
PARAMETER (NH=3, MC=4, LLayer=3)
INTEGER  A(0:NH, MC,LLayer)
```

Khi đó A là mảng ba chiều gồm 4 hàng, 4 cột và 3 lớp, có cấu trúc như sau



Và được lưu trữ trong bộ nhớ dưới dạng:



Sau đây là một số ví dụ truy cập mảng.

Nếu mảng A được khai báo bởi

```
REAL A(5,10), B(5,10)
```

Khi đó:

```
A = 3.0    ! Gán tất cả các phần tử của A bằng 3
A(1,1) = 4.  ! Gán phần tử hàng 1, cột 1 bằng 4.,
A(1,2) = 7.  ! Gán phần tử  hàng 1, cột 2 bằng 7.
A(2,1:8:3) = 2.5 ! Gán các phần tử cột 1, 4, 7 hàng 2 bằng 2.5
                 ! [ A(2,1) = A(2,4) = A(2,7) = 2.5 ]
B = SQRT(A) ! Gán tất cả các phần tử của B bằng căn bậc
            ! hai các phần tử tương ứng của A
```

Nếu khai báo

```
REAL A(10)
```

Khi đó:

```
A(1:5:2) = 3.0 ! Gán các phần tử A(1), A(3), A(5) bằng 3.0.
A(:5:2) = 3.0  ! Tương tự câu lệnh trên
A(2::3) = 3.0  ! Gán các phần tử A(2), A(5), A(8) bằng 3.0.
               ! (Chỉ số cao nhất ngầm định bằng 10)
A(7:9) = 3.0   ! Gán các phần tử A(7), A(8), A(9) to 3.0.
               ! (Bước ngầm định bằng 1)
A(:) = 3.0   ! Tương tự như A = 3.0;
```

Một ví dụ khác, nếu có khai báo

```
REAL A(10), B(5, 5)
INTEGER I(4), J(3)
```

Có thể gán giá trị cho I và J bằng cách:

```
I = (/ 5, 3, 8, 2 /)
J = (/ 3, 1, 5 /)
```

Còn câu lệnh

```
A(I) = 3.0
```

Có nghĩa là gán các phần tử A(5), A(3), A(8) và A(2) bằng 3.0:

và câu lệnh

```
B(2,J) = 3.0
```

là gán các phần tử B(2,3), B(2,1) và B(2,5) bằng 3.0:

5.3.1 Sử dụng lệnh DATA để khởi tạo mảng

Bạn cũng có thể sử dụng câu lệnh DATA để gán giá trị cho các phần tử mảng. Ví dụ:

```
REAL, DIMENSION(10) :: A, B, C(3,3)
DATA A / 5*0, 5*1 /     ! Gán 5 phần tử đầu bằng 0,
DATA B(1:5) / 4, 0, 5, 2, -1 /
! Chỉ gán giá trị cho các phần tử từ 1..5 (B(1)..B(5))
DATA ((C(I,J), J= 1,3), I=1,3) / 3*0, 3*1, 3*2 /
! Gán giá trị cho các phần tử của C lần lượt theo hàng
```

5.3.2 Biểu thức mảng

Có thể thực hiện các phép toán trên các mảng. Trong trường hợp này các mảng phải có cùng cấu trúc. Ví dụ:

```
REAL, DIMENSION(10) :: X, Y
X + Y     ! Cộng tương ứng các phần tử của X và Y: X(I) + Y(I)
X * Y     ! Nhân tương ứng các phần tử của X và Y: X(I) * Y(I)
X * 3     ! Nhân tương ứng các phần tử của X với 3: X(I) * 3
X * SQRT( Y )
! Nhân các phần tử của X với căn bậc 2 tương ứng của các phần tử của Y:
!           X(I) * SQRT( Y(I) )
X == Y ! Cho kết quả .TRUE. nếu X(I) == Y(I), và .FALSE. nếu ngược lại
```

5.3.3 Cấu trúc WHERE ... ELSEWHERE ... END WHERE

Đây là cấu trúc dùng trong thao tác với các mảng. Cú pháp cấu trúc như sau.

```
WHERE (Điều kiện)
   Các câu lệnh 1
ELSEWHERE
   Các câu lệnh 2
END WHERE
```

Tác động của câu lệnh là tìm các phần tử trong mảng thỏa mãn *Điều kiện*, nếu *Điều kiện* được thỏa mãn thì thực hiện *Các câu lệnh 1*, ngược lại thì thực hiện *Các câu lệnh 2*. Ví dụ:

```
REAL A (5)
A = (/ 89.5, 43.7, 126.4, 68.3, 137.7 /)
WHERE (A > 100.0) A = 100.0
```

Trong đoạn chương trình trên, tất cả các phần tử của mảng A có giá trị > 100 sẽ được thay bằng 100. Kết quả sẽ nhận được:

```
A = (89.5, 43.7, 100.0, 68.3, 100.0)
```

Hoặc

```
REAL A (5), B(5), C(5)
A = (/ 89.5, 43.7, 126.4, 68.3, 137.7 /)
B = 0.0
C = 0.0
WHERE (A > 100.0)
    A = 100.0
    B = 2.3
ELSEWHERE
    A = 50.0
    C = -4.6
END WHERE
```

Kết quả nhận được

Mảng	PT thứ 1	PT thứ 2	PT thứ 3	PT thứ 4	PT thứ 5
A	50.0	50.0	100.0	50.0	100.0
B	0.0	0.0	2.3	0.0	2.3
C	–4.6	–4.6	0.0	–4.6	0.0

5.4 Mảng động (Dynamical Array)

Mảng có kích thước và cách sắp xếp các phần tử không được xác định ngay từ lúc khai báo gọi là mảng động. Trên đây ta đã gặp một số ví dụ về khai báo và sử dụng mảng động. Có thể khái quát hóa cách khai báo mảng động lại như sau.

```
Kiểu DL, DIMENSION(MôTả), ALLOCATABLE :: DSbiến
```

hoặc

Kiểu DL, ALLOCATABLE [::] *DSbiến* [(*MôTả*)] [,...]

hoặc

ALLOCATABLE [::] *DSbiến* [(*MôTả*)] [,...]

Trong đó: *MôTả* là mô tả số chiều của mảng, được xác định bởi các dấu hai chấm (:). Ví dụ:

```
REAL, DIMENSION(:), ALLOCATABLE :: X  ! Mảng 1 chiều
REAL, ALLOCATABLE ::  vector(:)     ! Mảng 1 chiều
INTEGER, ALLOCATABLE :: matrix(:,:)  ! Mảng 2 chiều
```

Vì chưa được xác định kích thước ngay từ đầu nên để sử dụng mảng động cần phải mô tả rõ kích thước và cách sắp xếp các phần tử của chúng trước khi truy cập.

Câu lệnh **ALLOCATE** dùng để định vị kích thước và cách sắp xếp các phần tử mảng trong bộ nhớ (tức cấp phát bộ nhớ cho biến).

Câu lệnh **DEALLOCATE** dùng để giải phóng vùng bộ nhớ mà biến mảng động đã được cấp phát.

Ví dụ 1:

```
INTEGER, ALLOCATABLE :: matrix(:,:)
REAL, ALLOCATABLE :: vector(:)
...
N = 123
ALLOCATE (matrix(3,5),vector(-2:N+2))
...
DEALLOCATE matrix, vector
```

Ví dụ 2:

```
REAL A, B(:,:), C(:), D(:, :, :)
ALLOCATABLE C, D
....
READ (*, *) N, M
ALLOCATE (C(N), D(M, N, M))
```

Trong ví dụ này, kích thước các mảng C và D sẽ được xác định chỉ sau khi các giá trị của N và M đã xác định.

Ví dụ 3:

```
REAL, ALLOCATABLE :: A(:)
      ...
IF (.NOT. ALLOCATED(A)) ALLOCATE (A (5))
```

Trong ví dụ này, mảng A sẽ được cấp phát bộ nhớ nếu nó chưa đợc cấp phát.

Ví dụ 4:

```
REAL, ALLOCATABLE :: A(:)
```

```
INTEGER ERR
ALLOCATE (A (5), STAT = ERR)
IF (ERR .NE. 0) PRINT *, "Khong cap phat duoc"
```

Tham số STAT trong câu lệnh ALLOCATE sẽ trả về giá trị ERR (số nguyên). Nếu ERR=0 thì việc cấp phát bộ nhớ thực hiện thành công, ngược lại nếu không cấp phát được thì giá trị của ERR chính là mã lỗi lúc chạy chương trình.

Ví dụ 5: Chương trình sau đây nhập một mảng một chiều X gồm các số thực dương nhưng không biết trước số phần tử của mảng tối đa là bao nhiêu. Do đó mảng X sẽ được cấp phát bộ nhớ tăng dần trong khi nhập dữ liệu. Quá trình nhập dữ liệu chỉ kết thúc khi số nhập vào là một số âm. Thủ thuật thực hiện ở đây là sử dụng 2 mảng động, trong đó một mảng để lưu số liệu trung gian.

```
REAL, DIMENSION(:), ALLOCATABLE :: X, OldX
REAL      A
INTEGER   N
ALLOCATE (X(0)) ! Kích thước của X (lúc đầu bằng 0)
N = 0
DO
  Print*, 'Cho mot so: '
  READ(*,*) A
  IF (A < 0 ) EXIT    ! Nếu A<0 thì thoát
    N = N + 1         ! Tăng N lên 1 đơn vị
    ALLOCATE(OldX(SIZE(X))) ! Cấp phát kích thước
                            ! của X bằng OldX
    OldX = X          ! Lưu X vào OldX
    DEALLOCATE( X ) ! Giải phóng X
    ALLOCATE(X(N)) ! Cấp phát X có kích thước bằng N
    X = OldX       ! Gán toàn bộ OldX cho X
    X(N) = A      ! Gán giá trị mới cho phần tử thứ N của X
    DEALLOCATE( OldX ) ! Giải phóng OldX
END DO
PRINT*,N, ( X(I), I = 1, N )
END
```

Hàm SIZE(X) trong chương trình là để xác định kích thước hiện tại của mảng X.

5.5 Kiểu con trỏ

Kiểu con trỏ là một khái niệm để xác định biến có thuộc tính con trỏ. Biến con trỏ có thể là biến vô hướng hoặc biến mảng. Khai báo kiểu con trỏ như sau:

```
POINTER [::] Tên_con_trỏ [(MôTảMảng)] [, ...]
```

hoặc

```
Kiểu DL, POINTER :: Tên_con_trỏ [(MôTảMảng)][,... ]
```

Ví dụ, có thể khai báo biến con trỏ như sau.

```
REAL A, X(:,:), B, Y(5, 5)
POINTER A, X  ! A là con trỏ vô hướng, X là con trỏ mảng
```

hoặc

```
REAL, POINTER :: A (:,:)
REAL B, X(:,:)
POINTER B, X
```

Biến con trỏ có thể được cấp phát bộ nhớ bằng lệnh ALLOCATE hoặc trỏ đến một biến khác. Biến được con trỏ trỏ đến hoặc là một biến có thuộc tính đích (TARGET) hoặc đã được xác định. Trong trường hợp biến con trỏ trỏ đến một biến khác, nó được xem như là "bí danh" của biến nó trỏ đến. Để minh họa ta hãy xét ví dụ sau.

```
INTEGER, POINTER :: P1 (:)
INTEGER, POINTER :: P2 (:)
INTEGER, ALLOCATABLE, TARGET :: D (:)
ALLOCATE (D (7)) ! Cap phat bo nho cho bien DICH
D = 1
D (1:7:2) = 10.
PRINT*, 'DICH=',D

P1 => D  ! Con tro tro vao bien DICH
PRINT*,'CON TRO P1=',P1
ALLOCATE (P1(10)) ! Cap phat bo nho cho bien con tro
P1 = 5
P2 => P1  ! Con tro tro vao bien da xac dinh
PRINT*,'CON TRO P1=',P1
print*
print*,'CON TRO P2=',P2
P2 =8
PRINT*,'CON TRO P1=',P1
print*
print*,'CON TRO P2=',P2
END
```

Khi P1 trỏ vào biến D nó se nhận nội dung của biến D. Nhưng khi P1 được cấp phát bộ nhớ và khởi tạo giá trị mới (P1=5), sau đó P2 trỏ vào nó thì P2 và P1 đều có cùng nội dung của P1 đã thay đổi (tức bằng 5). Bây giờ gán P2 bằng 8 thì cả P2 và P1 đều nhận giá trị bằng 8.

5.5.1 Trạng thái con trỏ

Tất cả các biến con trỏ trong chương trình luôn tồn tại ở một trong 3 trạng thái sau:

– *Trạng thái không xác định (undefined).* Bắt đầu chương trình mọi con trỏ đều ở trạng thái này.

– *Trạng thái không trỏ vào đâu cả* (*null),* tức con trỏ chưa phải là "bí danh" của biến nào cả.

– *Trạng thái đã liên kết* (*associated),* tức con trỏ đã trỏ vào một biến nào đó (đã là "bí danh" của một biến "đích")

Để đưa con trỏ về trạng thái không trỏ vào đâu cả hãy dùng câu lệnh:

```
NULLIFY (P) ! P là biến con trỏ
```

Để xác định trạng thái hiện tại của con trỏ có thể sử dụng hàm

```
ASSOCIATED (P) ! P là biến con trỏ
```

Hàm này trả về giá trị .TRUE. nếu con trỏ đã liên kết với một biến, và trả về giá trị .FALSE. nếu con trỏ ở trạng thái không trỏ vào đâu cả.

5.5.2 Cấp phát và giải phóng biến con trỏ

Biến con trỏ có thể được cấp phát bộ nhớ bằng câu lệnh ALLOCATE và được giải phóng bởi câu lệnh DEALLOCATE tương tự như mảng động. Ví dụ:

```
REAL, POINTER :: P1
ALLOCATE( P1 ) ! Cấp phát bộ nhớ cho P1
P1 = 17 ! Gán giá trị cho P1 như đối với biến bất kỳ
PRINT*, P1
DEALLOCATE( P1 ) ! Giải phóng biến P1
```

Cả ALLOCATE và DEALLOCATE đều có tham số STAT, chẳng hạn:

```
ALLOCATE( P1, STAT = ERR )
DEALLOCATE( P1, STAT = ERR )
```

Số nguyên ERR bằng 0 nếu bộ nhớ đã được cấp phát hoặc giải phóng xong.

Có thể sử dụng cả ALLOCATABLE hoặc POINTER để khai báo mảng động, chẳng hạn:

```
REAL, DIMENSION(:), POINTER :: X
INTEGER, DIMENSION(:,:), ALLOCATABLE :: A
```

Ví dụ:

```
REAL, POINTER :: A(:), B, C
```

```
REAL, ALLOCATABLE, TARGET :: D(:)
REAL, TARGET :: E
REAL, ALLOCATABLE :: F(:, :)
...
ALLOCATE (B, D(5), F(4, 2))
A => D
C => E
...
DEALLOCATE (B, D, F)
```

CHƯƠNG 6. BIẾN KÝ TỰ

6.1 Khai báo biến ký tự

Hiểu một cách đơn giản, hằng ký tự là một xâu (dãy) các ký tự nằm giữa các cặp dấu nháy đơn (‘ ’) hoặc nháy kép (“ ”). Biến ký tự là biến có thể nhận giá trị là các hằng ký tự. Bởi vì mỗi ký tự chiếm 1 byte bộ nhớ, nên dung lượng bộ nhớ mà xâu ký tự chiếm phụ thuộc độ dài của xâu ký tự.

Nói chung có thể có nhiều cách khai báo biến ký tự như đã được đề cập đến trong mục 1.4.2. Sau đây dẫn ra một số ví dụ về cách khai báo biến ký tự thường dùng.

```
CHARACTER StrName [,...]        ! độ dài 1 ký tự
CHARACTER (n) StrName [,...]    ! độ dài n ký tự
CHARACTER *n StrName [,...]     ! độ dài n ký tự
CHARACTER StrName*n [,...]      ! độ dài n ký tự
```

Trong đó *StrName* là tên biến ký tự, **n** là một số nguyên dương chỉ độ dài cực đài của biến *StrName*. Ví dụ:

```
CHARACTER ALPHA   ! ALPHA nhận các giá trị ‘A’,...
CHARACTER (25) Name
! Name là một xâu dài tối đa 25 ký tự
CHARACTER Word*5
! Word là một xâu dài tối đa 5 ký tự
...
Name = “Hanoi, Ngay...“
Word = ‘Hanoi’
...
```

Khi hằng ký tự được xác định bởi khai báo PARAMETER ta còn có thể khai báo xâu có độ dài chưa xác định:

```
CHARACTER *(*) StrName
PARAMETER (StrName= “XauDai12KyTu”)
```

Trong trường hợp này độ dài của xâu sẽ là độ dài thực của xâu được gán.

6.2 Các xâu con (substring)

Xâu con là một bộ phận của xâu ký tự. Xâu con có thể chỉ có 1 ký tự cũng có thể là toàn bộ xâu. Giả sử TEXT là một xâu có độ dài cực đại 80 ký tự:

```
CHARACTER (80) TEXT
```

Khi đó TEXT(I:J) là xâu con gồm các ký tự từ ký tự thứ I đến ký tự thứ J của xâu TEXT. Từng ký tự riêng biệt trong xâu TEXT có thể được tham chiếu (truy cập) đến bằng xâu con TEXT(I:J). Ví dụ:

```
TEXT(:J)  ! từ ký tự thứ 1 đến ký tự thứ J
TEXT(1:J) ! từ ký tự thứ 1 đến ký tự thứ J
TEXT(J:)  ! từ ký tự thứ J đến ký tự thứ 80
TEXT(J:80) ! từ ký tự thứ J đến ký tự thứ 80
TEXT(:)   ! từ ký tự thứ 1 đến ký tự thứ 80 (cả xâu)
TEXT(1:80)  ! (hoặc TEXT) tương tự, cả xâu
```

Nếu

```
TEXT = "Hanoi—Vietnam"
```

thì

```
TEXT(3:5) có giá trị là "noi"
TEXT(:5) có giá trị là "Hanoi"
TEXT(7:) có giá trị là "Vietnam     "
TEXT(6:6) = " * " sẽ cho "Hanoi * Vietnam"
TEXT(2:5) = "ANOI" sẽ cho "HANOI—Vietnam"
```

6.3 Xử lý biến ký tự

Để chèn một xâu *SubStr* vào một vị trí nào đó của xâu *Str* cho trước cần phải dịch chuyển các ký tự phía sau vị trí cần chèn của xâu *Str* sang phải một số vị trí bằng độ dài xâu *SubStr*. Ví dụ, có xâu TEXT = "`Hanoi - Saigon`", nếu muốn chèn xâu con " - Hue" vào xâu này để nhận được xâu "`Hanoi — Hue — Saigon`" ta lập trình như sau:

```
CHARACTER (50) TEXT
TEXT = 'Hanoi - Saigon'
print*,TEXT
I = 6      ! Chèn vào vị trí thứ 6
LENSub = 6 ! Độ dài xâu cần chèn " — Hue" là 6
DO J = LEN_TRIM( TEXT ), I, -1
   TEXT( J+6:J+6 ) = TEXT(J:J)
END DO
TEXT(6:12) = ' - Hue'
print*,TEXT
END
```

Hàm `LEN_TRIM (TEXT)` trong chương trình trả về độ dài xâu TEXT sau khi đã loại bỏ các dấu cách ở bên phải nhất của xâu.

Sau đây là một số hàm xử lý xâu ký tự trong thư viện Fortran.

`LEN (Str)`: trả về độ dài cực đại (khai báo) của xây *Str*

`LEN_TRIM (Str)`: trả về độ dài xâu *Str* sau khi đã loại bỏ các ký tự trống (dấu cách) ở bên phải nhất

`ACHAR (I)`: trả về ký tự thứ I trong bảng mã ASCII

`IACHAR (c)`: trả về số thứ tự trong bảng mã ASCII của ký tự *c*

`INDEX (Str, SubStr [, back] )`: trả về vị trí đầu tiên của xâu con *SubStr* trong xâu *Str*. Tham số tùy chọn *back* có ý nghĩa như sau:

Nếu *back* = .TRUE.: tìm *SubStr* từ cuối xâu *Str*

Nếu *back* = .FALSE.: tìm *SubStr* từ đầu xâu *Str*

Giá trị ngầm định là *back* = .FALSE.

`REPEAT (Str, ncopies)`: trả về một xâu gồm *ncopies* lần copy *Str*

`TRIM (Str)`: trả về xâu *Str* sau khi đã cắt bỏ các ký tự trống ở bên phải nhất

6.4 Phép toán gộp xâu ký tự

Phép toán gộp hai xâu ký tự được ký hiệu là //. Ví dụ, bạn muốn tạo một tên file từ hai xâu là xâu *Name* chứa tên và xâu *Ext* chứa phần mở rộng. Có thể cả hai xâu này còn chứa các dấu cách ở đầu và cuối xâu. Trước khi gộp hai xâu này thành một xâu có ý nghĩa *tên của một file* bạn cần phải cắt bỏ các dấu cách đó. Ta có chương trình sau.

```
CHARACTER (80) FName, Name, Ext
Name = '   gl04012200   ' ! Có chứa dấu cách ở đầu và cuối
Ext='  .dat  '     ! Có chứa dấu cách ở đầu và cuối
! Xác định vị trí dấu cách cuối cùng bên trái của hai xâu:
Len1 = INDEX(TRIM(Name),' ',.true.)+1
Len2 = INDEX(TRIM(Ext),' ',.true.) + 1
! Xác định độ dài xâu sau khi đã cắt bỏ dấu cách bên phải của hai xâu:
Len3 = LEN(TRIM(Name))
Len4 = LEN(TRIM(Ext))
! Gộp tên và phần mở rộng để tạo thành tên file :
FName = Name(Len1:Len3) // Ext(Len2:Len4)
PRINT*, FName
END
```

Khi bạn chạy chương trình này, kết quả nhận được sẽ là "**gl04012200.dat**"

6.5 Tạo định dạng FORMAT bằng xâu ký tự

Biểu thức xâu ký tự có thể được sử dụng để tạo định dạng FORMAT tự động trong chương trình. Ví dụ sau đây cho phép in một số thực trong đó số chữ số thập phân cần in ra được lựa chọn tùy ý:

```
CHARACTER (1), DIMENSION(0:4) :: TP = &
             & (/'0','1','2','3','4'/)
CHARACTER (8) :: FMT = "(F9.?)"
```

```
PRINT*,'Cho so X:'
READ*,X
PRINT*,'Cho so chu so thap phan can in:'
READ*, N
FMT(5:5) = TP( N ) ! Thay dấu (?) bởi số chữ số thập phân
PRINT FMT, X
END
```

Chương trình sau sẽ in N số nguyên dương đầu tiên trên cùng 1 dòng:

```
CHARACTER *11 :: FMT = '(2X, ???I4)'
CHARACTER *3 SubSt
PRINT*,'CHO SO N:'
READ*,N
WRITE(SubSt,'(I3.3)') N
FMT(6:8)=SubSt
WRITE(*, FMT) (I,I=1,N)
END
```

6.6 Mảng xâu ký tự

Xâu ký tự có thể khai báo ở dạng biến đơn cũng có thể khai báo ở dạng biến mảng. Mảng xâu ký tự là mảng trong đó mỗi phần tử là một xâu ký tự. Các phần tử trong mảng xâu ký tự phải có độ dài giống nhau. Như vậy, nếu mỗi phần tử trong mảng có độ dài là **n** ký tự, thì mảng một chiều gồm **m** phần tử sẽ có kích thước **n** x **m** ký tự. Ví dụ:

```
CHARACTER (8), DIMENSION(7) :: DayOfWeek = &
       &(/ 'Thu 2', 'Thu 3', 'Thu 4', 'Thu 5',&
       &   'Thu 6','Thu 7','Chu nhat' /)
PRINT*,'Cac ngay trong tuan la:'
DO I = 1,7
   PRINT*, DayOfWeek (I)
END DO
END
```

Trong ví dụ này, mảng *DayOfWeek* là mảng một chiều gồm 7 phần tử, mỗi phần tử là một xâu ký tự có độ dài cực đại bằng 8 ký tự.

Bằng cách tương tự, ta có thể định nghĩa mảng ký tự hai chiều, ba chiều,…

CHƯƠNG 7. KIỂU FILE

7.1 Khái niệm

Trong hệ thống vào/ra của Fortran, dữ liệu được lưu trữ và chuyển đổi chủ yếu thông qua các **file**. Tất cả các nguồn vào/ra cung cấp và kết xuất dữ liệu được xem là **các file**. Các thiết bị như màn hình, bàn phím, máy in được xem là những **file ngoài** (external files), kể cả các file số liệu lưu trữ trên đĩa. Các biến trong bộ nhớ cũng có thể đóng vai trò như các file, đặc biệt chúng được sử dụng để chuyển đổi từ dạng biểu diễn mã ASCII sang số nhị phân (binary). Khi các biến được sử dụng theo cách này, chúng được gọi là các **file trong**.

Các **file trong** hoặc **file ngoài** đều được liên kết với cái gọi là thiết bị lôgic. Thiết bị lôgic là một khái niệm được sử dụng để tham chiếu đến các file. Ta có thể nhận biết một thiết bị lôgic liên kết với một file bằng định danh (**UNIT=**).

Định danh UNIT đối với một file trong là tên của một biến ký tự liên kết với nó. Định danh UNIT đối với một file ngoài hoặc là một số được gán trong lệnh **OPEN,** hoặc là một số kết nối trước như là định danh UNIT đối với thiết bị, hoặc dấu sao (*). Các định danh UNIT ngoài được kết nối với các thiết bị nhất định không được mở (OPEN). Các UNIT ngoài đã kết nối sẽ bị ngắt kết nối khi kết thúc thực hiện chương trình hoặc khi UNIT bị đóng bởi lệnh **CLOSE**.

Tại một thời điểm UNIT không thể kết nối với nhiều hơn một file, và file cũng không kết nối với nhiều hơn một thiết bị.

Định danh UNIT liên kết với một file ngoài phải là một biểu thức nguyên hoặc dấu sao (*). Nếu là biểu thức nguyên, giá trị của nó sẽ liê kết với một file trên đĩa; nếu là dấu sao (*) thì khi đọc vào nó được hiểu là bàn phím, còn khi in ra nó sẽ là màn hình. Ví dụ:

```
OPEN (UNIT = 10, FILE = 'TEST.dat')
WRITE(10,'(A18,\)')' Ghi vao File TEST.dat&
                  & da lien ket voi UNIT 10'
WRITE (*, '(1X, A30,\)') ' In ra man hinh.'
```

Fortran ngầm định một số thiết bị chuẩn liên kết với định danh UNIT như sau:

– Dấu sao (*): Màn hình hoặc bàn phím

– UNIT=0: Màn hình hoặc bàn phím

– UNIT=5: Bàn phím

– UNIT=6: Màn hình

Bạn hãy chạy chương trình ví dụ sau và thử phân tích xem chương trình viết vì mục đích gì.

```
REAL a, b
! In ra man hinh (UNIT 6 da ket noi truoc).
WRITE(6, '('' Day la UNIT 6'')')
! Su dung lenh OPEN de ket noi UNIT 6
! voi file ngoai co ten 'COSINES'.
OPEN (UNIT = 6, FILE = 'COSINES', STATUS = 'NEW')
```

```
DO a = 0.1, 6.3, 0.1
   b = COS (a)
! Ghi vao file 'COSINES'.
   WRITE  (6, 100) a, b
 100 FORMAT (F3.1, F5.2)
END DO
! Dong file.
CLOSE (6)
! Ket noi lai UNIT 6 voi man hinh bang cach
!                        ghi ra man hinh
WRITE(6,' ('' Cosines da xong'')')
END
```

Định danh UNIT liên kết với file trong là xâu ký tự hoặc mảng ký tự. Đối với các file trong bạn chỉ có thể sử dụng các câu lệnh với READ hoặc WRITE. Bạn không thể mở hoặc đóng file trong như đối với các file ngoài.

Bạn có thể đọc và ghi các file trong bằng lệnh định dạng FORMAT như đối với các file ngoài. Trước khi câu lệnh vào ra được thực hiện các file trong được định vị tại vị trí đầu của bản ghi đầu tiên.

Bằng khái niệm file trong Fortran cho phép bạn chuyển đổi giữa các dạng dữ liệu, chẳng hạn đổi ký tự sang số hoặc đổi số sang dạng ký tự. Để minh họa, ta hãy xét ví dụ sau.

```
CHARACTER(10) str
INTEGER n1, n2, n3
CHARACTER(14) fname
INTEGER  I
str = " 1  2  3"
READ(str, *) n1, n2, n3 ! Đọc n1, n2, n3 từ xâu str
I = 4
! Ghi giá trị của I vào biến fname
WRITE (fname, 200) I
200 FORMAT ('FM', I3.3, '.DAT')
END
```

Trong chương trình trên, ***Str và Fname*** là các file trong. Kết quả chạy chương trình bạn sẽ nhận được *fname* = "FM004.DAT".

7.2 Phân loại file

Fortran hỗ trợ hai phương pháp truy cập file là truy cập tuần tự và truy cập trực tiếp, và ba dạng cấu trúc file là có định dạng – Formatted, không định dạng – Unformatted, và dạng nhị phân – Binary).

7.2.1 File có định dạng (Formatted Files)

Có thể tạo file có định dạng bằng lệnh OPEN với tùy chọn FORM = "FORMATTED", hoặc bỏ qua tham số FORM khi muốn tạo file truy cập tuần tự. Các bản ghi của file có định dạng được lưu trữ như các ký tự ASCII. Mỗi bản ghi kết thúc bằng các ký tự (ASCII) điều khiển RETURN (CR) và xuống dòng (LF – line

feed). Để xem nội dung file có thể sử dụng các trình soạn thảo văn bản thông thường. Ví dụ:

```
OPEN (UNIT=3, FILE= “TEST.TXT”, FORM= “FORMATTED”)
WRITE (3, *) “Day la file co dinh dang”
CLOSE (3)
END
```

7.2.2 File không định dạng (Unformatted Files)

Để tạo một file không định dạng có thể sử dụng lệnh OPEN với tùy chọn FORM=“UNFORMATTED”, hoặc bỏ qua tham số FORM khi muốn tạo file truy cập trực tiếp. File không định dạng là một chuỗi bản ghi các khối vật lý. Mỗi bản ghi chứa tuần tự các giá trị lưu trữ gần giống với những gì sử dụng trong bộ nhớ chương trình. Tốc độ truy cập dữ liệu trong các file này nhanh hơn và chúng được tổ chức chặt chẽ hơn. Nếu các file không định dạng lưu trữ các số, chúng sẽ không thể đọc được bằng các trình soạn thảo văn bản thông thường. Nói chính xác hơn, khi sử dụng các trình soạn thảo để đọc các file không định dạng, những thông tin bằng số sẽ không xem được. Ví dụ:

```
CHARACTER *50 St
INTEGER A,B,C,D
OPEN (UNIT=3, FILE= "TEST.TXT", FORM= "UNFORMATTED")
St="Day la file khong dinh dang truy cap tuan tu"
WRITE (3) St
WRITE (3) 1,2,3,4
WRITE (3) 5,6,7,8
REWIND (3)
READ (3) St
PRINT*,ST
READ (3) A, B, C, D
PRINT*, A, B, C, D
READ (3) A, B, C, D
PRINT*, A, B, C, D
CLOSE (3)
END
```

7.2.3 File dạng nhị phân (Binary Files)

Có thể tạo một file nhị phân bằng lệnh OPEN với tùy chọn FORM= 'BINARY'. File nhị phân là dạng file chặt chẽ nhất, rất tốt cho việc lưu trữ số liệu có dung lượng lớn. Ví dụ:

```
CHARACTER *50 St
```

```
INTEGER A,B,C,D
OPEN (UNIT=3, FILE= "TEST.TXT", FORM= "BINARY")
St= "Day la file dang nhi phan truy cap tuan tu"
WRITE (3) St
WRITE (3) 1,2,3,4
REWIND (3)
READ (3) St
PRINT*,ST
READ (3) A, B, C, D
PRINT*, A, B, C, D
CLOSE (3)
END
```

7.2.4 File truy cập tuần tự (Sequential-Access Files)

Dữ liệu trong file tuần tự cần phải được truy cập hợp lệ, bản ghi này tiếp nối sau bản ghi khác, trừ khi ta thay đổi vị trí con trỏ file bằng các câu lệnh REWIND hoặc BACKSPACE. Các phương pháp vào/ra có thể sử dụng đối với file truy cập tuần tự là NONADVANCING, LIST–DIRECTED, và NAMELIST–DIRECTED. Các file trong cần phải là file tuần tự. Đối với các file liên kết với các thiết bị tuần tự cần phải sử dụng cách truy cập tuần tự. Các thiết bị tuần tự là thiết bị lưu trữ vật lý. Bàn phím, màn hình và máy in là những thiết bị tuần tự.

7.2.5 File truy cập trực tiếp (Direct-Access Files)

Dữ liệu trong file truy cập trực tiếp có thể được đọc và ghi theo một trình tự bất kỳ. Các bản ghi được đánh số một cách tuần tự, bắt đầu từ 1. Tất cả các bản ghi có độ dài được chỉ ra bởi tham số tùy chọn RECL trong câu lệnh OPEN. Số liệu trong file truy cập trực tiếp được truy cập đến bằng việc chỉ ra số thứ tự bản ghi trong file.

7.3 Tổ chức dữ liệu trong file

Như đã thấy ở trên, với hai phương pháp truy cập file và ba dạng cấu trúc file, một cách tương đối ta có thể phân chia thành 6 dạng tổ chức dữ liệu trong file:

1) File truy cập tuần tự có định dạng (Formatted Sequential)

2) File truy cập trực tiếp có định dạng (Formatted Direct)

3) File truy cập tuần tự không định dạng (Unformatted Sequential)

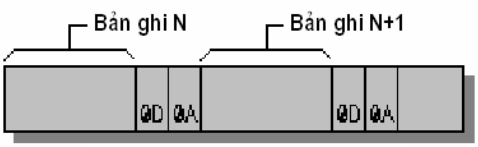
4) File truy cập trực tiếp không định dạng (Unformatted Direct)

5) File truy cập tuần tự dạng nhị phân (Binary Sequential)

6) File truy cập trực tiếp dạng nhị phân (Binary Direct)

7.3.1 File truy cập tuần tự có định dạng

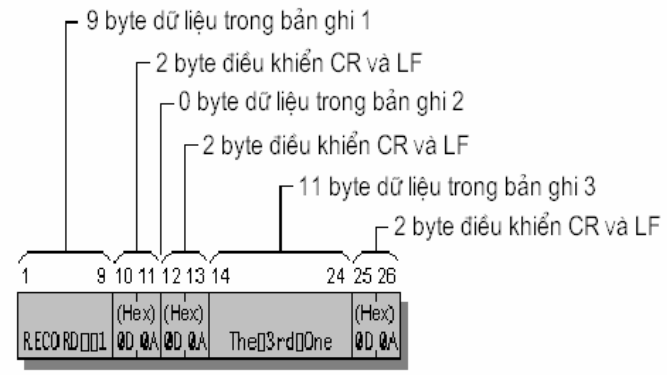
File tuần tự có định dạng là một chuỗi các bản ghi có định dạng được ghi một cách tuần tự (hình 7.1) và được đọc theo thứ tự xuất hiện trong file. Các bản ghi có thể có độ dài biến đổi và có thể rỗng. Chúng được phân cách nhau bởi ký tự điều khiển RETURN ($0D hay #13) và ký tự xuống dòng ($0A hay #10). Ví dụ:



Hình 7.1

```
OPEN (3, FILE='TEST1.TXT')
! TEST1.TXT ngầm định là file tuần tự có định dạng
WRITE (3, '(A, I3)') 'RECORD', 1
WRITE (3, '()')
WRITE (3, '(A11)') 'The 3rd One'
CLOSE (3)
END
```

Mô tả cấu trúc dữ liệu trong file TEST1.TXT được cho trên hình 7.2. Vì giữa các bản ghi được phân cách nhau bởi các ký tự điều khiển nên các bản ghi có thể có độ dài khác nhau tùy ý.



Hình 7.2

7.3.2 File truy cập trực tiếp có định dạng

Trong file truy cập trực tiếp có định dạng, tất cả các bản ghi có cùng độ dài và có thể được ghi hoặc đọc theo thứ tự bất kỳ. Kích thước bản ghi được chỉ ra bởi tùy chọn RECL= trong câu lệnh OPEN và nên bằng hoặc lớn hơn số byte của bản ghi dài nhất. Các ký tự RETURN (CR) và xuống dòng (LF) là những ký tự phân cách giữa các bản ghi và không tính vào giá trị của RECL. Một khi bản ghi truy cập trực tiếp đã được ghi, nó không thể bị xóa nhưng vẫn có thể bị ghi đè.

Trong khi kết xuất (output) ra file truy cập trực tiếp có định dạng, nếu số liệu không lấp đầy hoàn toàn bản ghi, trình biên dịch sẽ đệm vào phần còn lại của bản ghi các dấu cách (BLANK SPACES). Các dấu cách bảo đảm rằng file chỉ chứa những bản ghi đã lấp đầy hoàn toàn và tất cả các bản ghi đều có cùng độ dài.

Khi đọc vào (input), trình biên dịch cũng ngầm định là có đệm các dấu cách vào nếu danh sách đọc vào và định dạng đòi hỏi nhiều dữ liệu hơn bản ghi đã chứa.

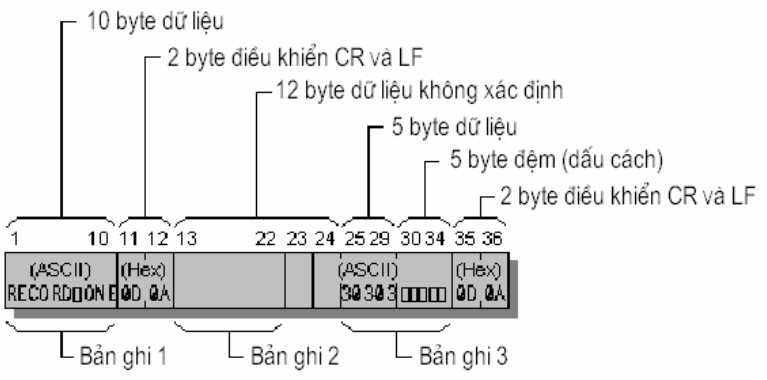
Có thể bỏ qua ngầm định việc đệm vào các dấu cách ở dữ liệu vào bằng cách đặt tùy chọn PAD= "NO" trong câu lệnh OPEN. Nếu PAD= "NO", bản ghi đọc vào cần phải chứa lượng dữ liệu được chỉ ra bởi danh sách đầu vào và định dạng FORMAT. Nếu không sẽ xuất hiện lỗi.

PAD= "NO" không có ảnh hưởng đối với kết xuất.

Ví dụ:

```
character st*10
integer n
OPEN (3,FILE='TEST2.TXT',FORM='FORMATTED',&
          ACCESS='DIRECT',RECL=10)
st = 'RECORD ONE'
WRITE (3, '(A10)', REC=1) st
n= 30303
WRITE (3, '(I5)', REC=3) n
CLOSE (3)
OPEN (3,FILE='TEST2.TXT',       FORM='FORMATTED',    &
          ACCESS='DIRECT',RECL=10)
st = ' '
n=0
read(3,'(A10)',rec=1) st
read(3,'(I5)',rec=3) n
print*,st
print*,n
END
```

Mô tả tổ chức dữ liệu theo chương trình này được cho trên hình 7.3. Độ dài của mỗi bản ghi là 10 byte, cộng với với 2 byte chứa ký tự điều khiển, nên những vị trí bản ghi chưa có dữ liệu sẽ có "khoảng trống" 12 byte không xác định. Đối với những bản ghi có dữ liệu chiếm ít hơn 10 byte, số byte còn lại sẽ được lấp đầy (đệm) bằng các dấu cách.



Hình 7.3

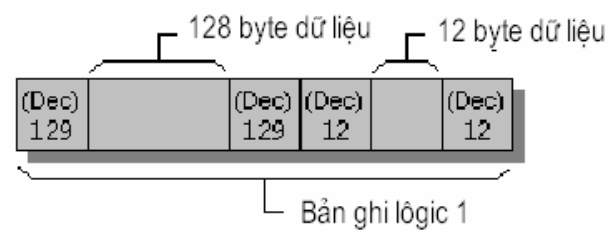
7.3.3 File truy cập tuần tự không định dạng

File tuần tự không định dạng được tổ chức hơi khác nhau chút ít giữa các dòng máy khác nhau. Sau đây sẽ nói đến loại file này đối với Fortran PowerStation.

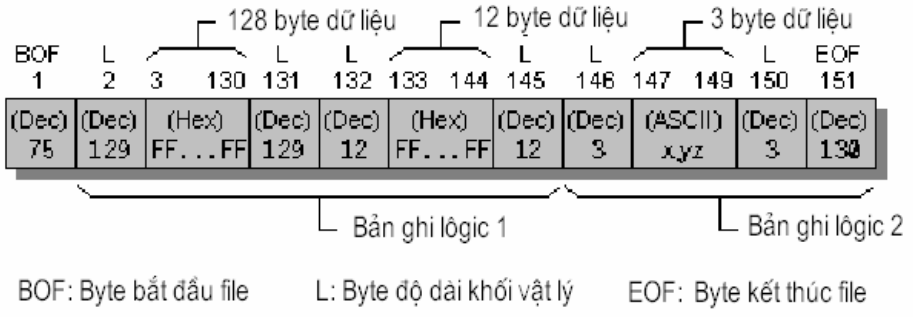
Các bản ghi trong file tuần tự không định dạng có thể có độ dài biến đổi. File tuần tự không định dạng được tổ chức thành từng khúc 130 byte hoặc nhỏ hơn được gọi là các khối vật lý. Mỗi khối vật lý bao gồm dữ liệu gửi vào file (cho đến 128 byte) và 2 byte chỉ độ dài do trình biên dịch chèn vào. Các byte độ dài cho biết mỗi bản ghi bắt đầu và kết thúc ở đâu. Một bản ghi lôgic tham chiếu đến một bản ghi không định dạng chứa một hoặc nhiều hơn các khối vật lý. Các bản ghi lôgic có thể lớn tùy ý; trình biên dịch sẽ biết cung cấp số khối vật lý cần thiết để chứa.

Khi tạo một bản ghi lôgic gồm nhiều hơn một khối vật lý, trình biên dịch đặt byte độ dài bằng 129 để chỉ rằng số liệu trong khối vật lý hiện tại sẽ nối tiếp vào khối vật lý tiếp theo. Ví dụ, một bản ghi lôgic có độ dài 140 byte sẽ được tổ chức như trên hình 7.4. Chương trình sau đây là một ví dụ về tạo file tuần tự không định dạng. Cấu trúc dữ liệu trong file được mô tả trên hình 7.5.

```
CHARACTER xyz(3)
INTEGER(4) idata(35)
DATA  idata /35 * -1/, xyz /'x', 'y', 'z'/
OPEN (3, FILE='TEST3.TXT',FORM='UNFORMATTED')
WRITE (3) idata
WRITE (3) xyz
CLOSE (3)
END
```



Hình 7.4



Hình 7.5

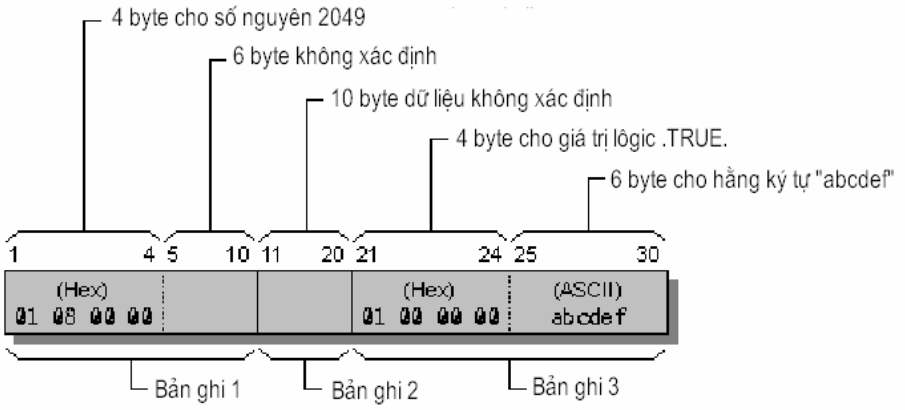
7.3.4 File truy cập trực tiếp không định dạng

File trực tiếp không định dạng là một chuỗi các bản ghi không định dạng. Có thể ghi hoặc đọc các bản ghi theo thứ tự tùy ý. Tất cả các bản ghi có cùng độ dài được cho bởi tham số RECL= trong câu lệnh OPEN. Không có byte phân định ranh giới các bản ghi hay nói cách khác không chỉ ra cấu trúc bản ghi. Có thể ghi một phần bản ghi vào file trực tiếp không định dạng. Fortran PowerStation sẽ đệm vào các bản ghi này bằng các ký tự rỗng (NULL) ASCII để cho độ dài bản ghi là cố định. Những bản ghi không ghi gì cả trong file sẽ chứa các số liệu không xác định.

Ví dụ:

```
OPEN (3, FILE='TEST4.TXT', RECL=10,     &
  FORM = 'UNFORMATTED', ACCESS = 'DIRECT')
WRITE (3, REC=3) .TRUE., 'abcdef'
WRITE (3, REC=1) 2049
CLOSE (3)
END
```

Mô tả cấu trúc dữ liệu trong file được cho trên hình 7.6.



Hình 7.6

7.3.5 File truy cập tuần tự dạng nhị phân

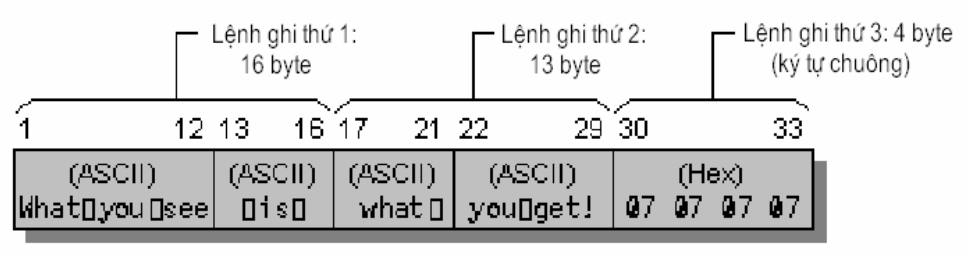
File truy cập tuần tự dạng nhị phân là một chuỗi các giá trị được ghi và đọc theo cùng trình tự và được lưu trữ như những số nhị phân. Trong file tuần tự dạng nhị phân không tồn tại ranh giới bản ghi, và không có byte đặc biệt để chỉ ra cấu trúc file. Số liệu được đọc và ghi không bị thay đổi dạng hoặc độ dài. Đối với mọi hạng mục vào/ra, tuần tự các byte trong bộ nhớ cũng chính là tuần tự các byte trong file.

Ví dụ:

```
INTEGER(1) Chuong(4)
CHARACTER(4) V1(3)
CHARACTER(4) V2
```

```
DATA   Chuong /4*7/
DATA  V2 /' is '/, V1 /'What',' you',' see'/
OPEN (3, FILE='TEST5.TXT',FORM='BINARY')
WRITE (3) V1, V2
WRITE (3) 'what ', 'you get!'
WRITE (3) Chuong
CLOSE (3)
END
```

Cấu trúc dữ liệu trong file TEST5.TXT được mô tả trên hình 7.7.



Hình 7.7

7.3.6 File truy cập trực tiếp dạng nhị phân

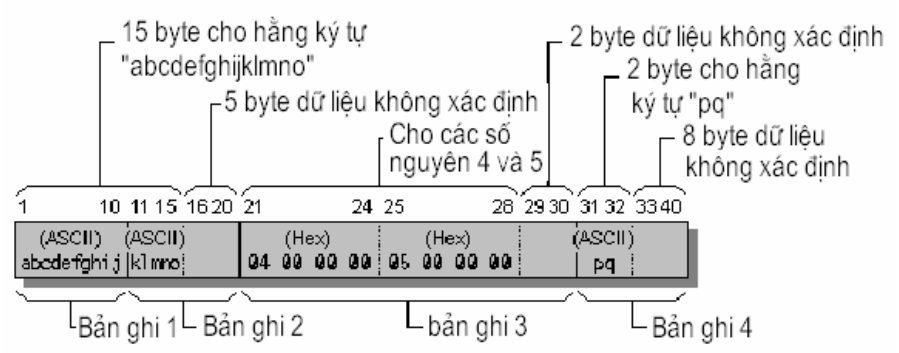
File truy cập trực tiếp dạng nhị phân lưu trữ các bản ghi như là một chuỗi các số nhị phân, có thể truy cập được theo trình tự bất kỳ. Các bản ghi trong file có cùng độ dài được chỉ ra bởi tham số RECL= của câu lệnh OPEN. Có thể ghi một phần bản ghi vào file trực tiếp dạng nhị phân; phần chưa sử dụng của bản ghi sẽ chứa dữ liệu không xác định.

Một câu lệnh đơn READ hoặc WRITE có thể truyền tải dữ liệu nhiều hơn một bản ghi liên tiếp trong file. Tuy nhiên điều đó có thể gây nên lỗi. Ví dụ, bạn hãy chạy chương trình sau đây và cố gắng khám phá xem tác động đến file của mỗi câu lệnh đọc, ghi như thế nào.

```
character*20 st
OPEN(3, FILE='TEST6.TXT',RECL=10,FORM='BINARY', &
             ACCESS='DIRECT')
WRITE (3, REC=1) 'abcdefghijklmno'
WRITE (3) 4,5
WRITE (3, REC=4) 'pq'
CLOSE (3)
OPEN  (3, FILE='TEST6.TXT',RECL=10,FORM='BINARY',&
               ACCESS='DIRECT')
read(3,rec=3) i,j
write(*,*) i,j
```

```
read(3,rec=1)  st
print*,st
read(3,rec=2)st
write(*,*) st
END
```

Mô tả cấu trúc file TEST6.TXT được cho trên hình 7.8. Trong trường hợp này độ dài biến *ST* bằng 20 byte, gấp hai lần độ dài của một bản ghi. Do đó nếu bạn đọc nội dung của bản ghi thứ hai (*rec=2*) và gán cho biến ký tự bạn sẽ nhận được "*klmno*". Bản ghi thứ ba lưu hai số nguyên 4 byte, nên còn dư 2 byte không xác định. Tương tự như vậy đối với bản ghi thứ 4.



Hình 7.8

7.4 Lệnh mở (OPEN) và đóng (CLOSE) file

Ta đã làm quen với lệnh mở file (OPEN) và lệnh đóng file (CLOSE) qua các ví dụ mà chưa giải thích gì thêm. Trong mục này ta sẽ khảo sát kỹ hơn các câu lệnh này.

7.4.1 Lệnh mở file

Một cách tổng quát, cú pháp câu lệnh mở file có dạng:

```
OPEN ([UNIT=] unit [, ACCESS=access] [,ACTION=action]
[, BLANK=blanks] [, BLOCKSIZE=blocksize]
[, CARRIAGECONTROL=carriagecontrol] [ , DELIM=delim]
[, ERR=err] [ , FILE=file] [ , FORM=form] [, IOFOCUS =
iofocus] [ , IOSTAT=iostat] [ , PAD=pad] [, POSITION =
position] [,RECL=recl][,SHARE=share] [,STATUS=status])
```

Như bạn đã thấy, nó khá phức tạp bởi có rất nhiều tham số. Tuy vậy, bạn đừng vội chán nản, vì hầu như các tham số này đều là những tùy chọn, bạn có thể bỏ qua những tùy chọn mà bạn chưa hiểu. Sau đây là ý nghĩa và cách sử dụng các tham số.

UNIT là tham số dùng để khai báo thiết bị lôgic sẽ được liên kết với file. Nếu bỏ qua **UNIT=** thì *unit* cần phải là tham số đầu tiên. Ngược lại, các tham số có thể xuất hiện theo thứ tự bất kỳ.

unit: Là một số nguyên (INTEGER(4)) lớn hơn hoặc bằng 0, dùng như một thiết bị lôgic để liên kết với file ngoài hoặc thiết bị ngoài.

access: Chỉ ra cách truy cập vào file, có thể nhận một trong các giá trị "APPEND", "DIRECT", hoặc "SEQUENTIAL" (ngầm định).

action: Là tham số mô tả tác động dự định đối với file. Có thể nhận giá trị 'READ' (file chỉ đọc), 'WRITE' (chỉ để ghi vào file), hoặc 'READWRITE' (cả đọc từ file và ghi vào file). Nếu bỏ qua *action*, chương trình sẽ cố gắng mở file với 'READWRITE'. Nếu không được, trước hết chương trình sẽ mở file với 'READ', sau đó với 'WRITE'. Việc bỏ qua *action* không giống với **ACTION**='READWRITE'. Nếu **ACTION**='READWRITE' trong khi file không thể truy cập bằng cả đọc và ghi thì việc mở file sẽ không thành công. Do đó việc bỏ qua *action* sẽ mềm dẻo và linh động hơn.

blanks: Có dạng ký tự (Character*(*)), dùng để điều khiển biểu diễn các ký tự trống (dấu cách) đối với vào/ra có định dạng, nhận một trong các giá trị 'NULL' hoặc 'ZERO'. Giá trị 'NULL' (ngầm định) để bỏ qua ký tự trống, giá trị 'ZERO' để xử lý các ký tự trống (các dấu cách ở đầu) như là những số 0.

blocksize: Ngầm định là một số nguyên ((INTEGER(4)) dùng để biểu diễn kích thước vùng đệm (tính bằng byte)

carriagecontrol: Có dạng ký tự (Character*(*)), dùng để chỉ việc hiểu ký tự đầu tiên của bản ghi trong file có định dạng như thế nào; nhận một trong các giá trị 'FORTRAN' hoặc 'LIST'. Ngầm định đối với các **UNIT** kết nối với các thiết bị ngoài như máy in hoặc màn hình là 'FORTRAN'; ngầm định đối với các **UNIT** kết nối với các file là 'LIST'. *carriagecontrol* bị bỏ qua khi sử dụng tùy chọn **FORM**= 'UNFORMATTED' hoặc **FORM**='BINARY'.

delim: Có dạng ký tự (Character*(*)), dùng để chỉ cách định ranh giới các bản ghi trong list-directed hoặc namelist-formatted; nhận một trong các giá trị 'APOSTROPHE', 'QUOTE', hoặc 'NONE' (ngầm định).

err: Là nhãn của câu lệnh thực hiện trong chương trình. Khi gặp lỗi chương trình sẽ chuyển điều khiển đến câu lệnh có nhãn *err*. Nếu bỏ qua, hiệu ứng lỗi vào/ra sẽ được xác định bởi *iostat*.

file: Có dạng ký tự (Character*(*)), dùng để chỉ ra tên file cần mở; có thể là dấu cách, tên file hợp lệ, tên thiết bị hoặc tên biến xác định file trong. Đối với Windows NT và Windows 9x trở lên, tên file cho phép dài hơn 8 ký tự, phần mở rộng dài hơn 3 ký tự. Nếu *file* bị bỏ qua, trình biên dịch sẽ tạo một file tạp nham (file nháp) nào đó chỉ có tên (không có phần mở rộng) và file này sẽ bị xóa khi gặp lệnh đóng file hoặc khi chương trình kết thúc.

form: Xác định kiểu file sẽ được mở, nhận một trong các giá trị 'FORMATTED', 'UNFORMATTED', hoặc 'BINARY'. Đối với file truy cập tuần tự giá trị ngầm định là 'FORMATTED'; đối với file truy cập trực tiếp giá trị ngầm định là 'UNFORMATTED'.

iofocus: Tham số lôgic, dùng để chỉ việc có đặt cửa sổ con (child window) là cửa sổ hoạt động hay không. Giá trị .TRUE. (ngầm định) tạo ra lời gọi SETFOCUSQQ ngay lập tức trước khi thực hiện các lệnh READ, WRITE, hoặc PRINT.

iostat: Là tham số kết xuất, ngầm định là một số nguyên (INTEGER(4); cho giá trị bằng *0* nếu không xuất hiện lỗi mở file, giá trị *âm* nếu gặp lỗi kết thúc bản ghi, hoặc một số xác định mã lỗi.

pad: Có dạng ký tự (Character*(*)), dùng để chỉ ra việc có đệm các dấu cách vào bản ghi khi đọc hay không nếu định dạng bản ghi có độ dài lớn hơn độ dài dữ liệu. *pad* nhận giá trị bằng 'YES' (ngầm định) hoặc 'NO'. Tham số này chỉ tác động khi đọc dữ liệu.

position: Có dạng ký tự (Character*(*)), chỉ ra vị trí con trỏ file truy cập tuần tự, nhận một trong các giá trị 'ASIS' (ngầm định), 'REWIND', hoặc 'APPEND'. Nếu *position* nhận giá trị 'REWIND', con trỏ file sẽ định vị tại đầu file; nếu nhận giá trị 'APPEND', con trỏ file sẽ định vị tại cuối file; nếu nhận giá trị 'ASIS', con trỏ file không thay đổi vị trí (tức giữ nguyên vị trí hiện thời trong file). Vị trí con trỏ file đối với file mới luôn luôn ở đầu file.

recl: Ngầm định là số nguyên, để chỉ độ dài tính bằng byte của một bản ghi trong file truy cập trực tiếp, hoặc độ dài bản ghi cực đại đối với file tuần tự. Đối với file truy cập trực tiếp đây là tham số đòi hỏi phải có.

share: Có dạng ký tự (Character*(*)), chỉ ra quyền truy cập đối với file được mở. Các giá trị của *share* có ý nghĩa như sau:

'DENYRW': Cấm đọc/ghi

'DENYWR': Cấm ghi

'DENYRD': Cấm đọc

'DENYNONE': Cho phép cả đọc và ghi (ngầm định)

status: Có dạng ký tự (Character*(*)), dùng để mô tả trạng thái của file được mở. Sau đây là các giá trị có thể của tham số này.

status = 'OLD': File đang tồn tại. Mở thành công nếu file tồn tại, ngược lại sẽ xuất hiện lỗi;

status='NEW': File mới. Nếu file không tồn tại nó sẽ được tạo mới. Nếu file đang tồn tại sẽ xuất hiện lỗi;

status='SCRATCH' : Nếu bỏ qua tham số *file* thì giá trị của *status* ngầm định là 'SCRATCH'. File *Scratch* là file tạm thời, nó sẽ bị xóa khi đóng file hoặc khi chương trình kết thúc;

status='REPLACE' : File được mở thay thế file có cùng tên. Nếu file cùng tên không tồn tại thì một file mới được tạo ra.

status='UNKNOWN' (ngầm định): Trong lúc chương trình chạy hệ thống sẽ cố gắng mở file với *status* = 'OLD', và sau đó với *status* = 'NEW'. Nếu file tồn tại thì nó được mở, nếu không tồn tại thì nó được tạo mới. Sử dụng **STATUS=** 'UNKNOWN' để tránh các lỗi xảy ra trong lúc chạy chương trình liên quan đến việc mở một file đang tồn tại với **STATUS=**'NEW' hoặc mở một file không tồn tại với **STATUS =** 'OLD'.

Các giá trị của *status* chỉ ảnh hưởng tới các file trên đĩa, và được bỏ qua đối với các thiết bị như bàn phím và màn hình hoặc máy in.

7.4.2 Lệnh đóng file

Cú pháp lệnh đóng file CLOSE có dạng:

```
CLOSE ([UNIT=] unit [, ERR=err] [, IOSTAT= iostat]
[, STATUS=status] )
```

Trong đó: nếu **UNIT=** bị bỏ qua thì *unit* phải là tham số đầu tiên, ngược lại các tham số có thể xuất hiện theo thứ tự bất kỳ.

unit: Là một số nguyên chỉ thiết bị lôgic liên kết với file được mở. Sẽ không xuất hiện lỗi nếu file không mở.

err: Là nhãn của câu lệnh thực hiện trong chương trình sẽ được chuyển điều khiển tới nếu lỗi xuất hiện khi đóng file. Nếu bỏ qua, hiệu ứng lỗi vào/ra được xác định bởi sự có mặt hoặc không có mặt của tham số *iostat*.

iostat: Là tham số kết xuất, ngầm định là một số nguyên (INTEGER*4), nhận giá trị bằng 0 nếu không xuất hiện lỗi khi đóng file, hoặc một số chỉ mã lỗi.

status: Tham số vào, là một biểu thức ký tự (CHARACTER*(*)), có các giá trị "KEEP" hoặc "DELETE"; ngầm định là "KEEP", ngoại trừ file nháp. Các file được mở không có tham số **FILE=** được gọi là các file nháp ("scratch" files). Đối với những file này giá trị ngầm định của *status* là 'DELETE'. Nếu đặt STATUS='KEEP' đối với những file nháp sẽ gây nên lỗi run-time.

7.5 Các lệnh vào ra dữ liệu với file

7.5.1 Lệnh đọc dữ liệu từ file (READ)

Cú pháp lệnh READ làm việc với file có dạng:

```
READ {{ fmt , | nml } | ( [UNIT=] unit [ , { [FMT=]
fmt | [NML=] nml } ] [ , ADVANCE=advance] [ , END=end]
[ , EOR=eor] [ , ERR=err] [ , IOSTAT=iostat] [, REC=rec]
[ , SIZE = size ])} iolist
```

Trong đó dấu gạch đứng (|) dùng để phân chia các tham số thuộc một nhóm, có nghĩa là chỉ được phép chọn một trong các tham số của nhóm . Ví dụ, nếu đã chọn `[FMT=] fmt` thì không được phép chọn `[NML=] nml`. Sau đây là ý nghĩa các tham số.

Nếu bỏ qua **UNIT=**, thì *unit* phải là tham số đầu tiên. Nếu bỏ qua **FMT=** hoặc **NML=**, thì *fmt* hoặc *nml* phải là tham số thứ hai. Nếu muốn sử dụng *fmt* không có **FMT**=, thì phải bỏ qua **UNIT**=. Trong các trường hợp khác các tham số có thể xuất hiện theo thứ tự bất kỳ. Hoặc *fmt* hoặc *nml* cần được chỉ ra nhưng không phải cả hai.

unit: Là tên thiết bị lôgic. Khi đọc từ một file ngoài *unit* là một biểu thức nguyên. Khi đọc từ một file trong *unit* là một xâu ký tự, một biến ký tự hoặc một phần tử của mảng ký tự,...

fmt: Là chỉ thị định dạng, có thể nhận một trong các trường hợp: Nhãn của lệnh **FORMAT**; Biểu thức ký tự (biến, thủ tục, hoặc hằng) xác định định dạng đọc vào, phân định bởi các dấu nháy đơn (') hoặc dấu nháy kép ("); dấu sao (*) đối với trường hợp định dạng tự do; hoặc một biến nguyên **ASSIGN**. Không được dùng *fmt*

đối với NAMELIST. Nếu lệnh **READ** bỏ qua các tùy chọn **UNIT=**, **END=**, **ERR=**, và **REC=**, và chỉ có *fmt* và *iolist*, thì câu lệnh sẽ đọc từ UNIT (*) là bàn phím.

nml: Chỉ tên của NAMELIST. Các tùy chọn *iolist* và *fmt* phải được bỏ qua. Lệnh đọc NAMELIST chỉ có thể được thực hiện đối với file truy cập tuần tự.

advance: Có dạng ký tự (Character*(*)), dùng để chỉ ra cách đọc vào từ file tuần tự có định dạng, nhận giá trị hoặc 'YES' (ngầm định) hoặc 'NO'.

Nếu *advance* = 'YES': Chương trình sẽ đọc theo định dạng *fmt* hết bản ghi này sang bản ghi khác và gán giá trị cho *iolist*. Nếu *advance* = 'NO': Chương trình sẽ đọc theo định dạng chỉ trên một dòng các giá trị và gán cho *iolist*. Nếu số giá trị chứa trong bản ghi ít hơn số biến trong *iolist* sẽ xuất hiện lỗi. Yêu cầu tham số định dạng *fmt* phải khác (*)

end: Nhãn của câu lệnh trong chương trình sẽ được chuyển điều khiển đến nếu con trỏ file đặt ở bản ghi kết thúc file. Nếu bỏ quan *end*, sẽ xuất hiện lỗi khi con trỏ file đã ở cuối file nhưng quá trình đọc vẫn cố gắng đọc tiếp, trừ trường hợp có chỉ ra *err* hoặc *iostat.*

eor: Nhãn của câu lệnh trong chơng trình sẽ được chuyển điều khiển đến nếu con trỏ file đặt ở cuối bản ghi. Tham số này chỉ dùng khi đưa vào tham số **ADVANCE**='NO'. Nếu bỏ qua *eor*, hiệu ứng lỗi vào/ra sẽ được xác định bởi *iostat.*

err: Nhãn của câu lệnh trong chương trình sẽ được chuyển đến nếu gặp lỗi trong quá trình đọc. Nếu bỏ qua *err*, hiệu ứng lỗi vào/ra sẽ được xác định bởi *iostat.*

iostat: Là tham số kết xuất, ngầm định là một số nguyên (INTEGER(4)). *iostat* = 0 nếu không có lỗi, = –1 nếu gặp kết thúc file (end-of-file), hoặc bằng một số chỉ thị thông báo lỗi.

rec: Tham số vào, ngầm định là một số nguyên dương (INTEGER(4)) chỉ số thứ tự bản ghi cần đọc đối với file truy cập trực tiếp. Sẽ xuất hiện lỗi khi sử dụng tham số này cho file truy cập tuần tự hoặc file trong. Khi sử dụng tham số *rec* cần bỏ qua các tham số *end* và *nml*. Con trỏ file sẽ được định vị đến bản ghi có số thứ tự *rec* trước khi số liệu được đọc. Số thứ tự bản ghi bắt đầu từ 1. Ngầm định của *rec* là số thứ tự bản ghi hiện thời.

size: Ngầm định là một số nguyên (INTEGER(4)), trả về số lượng ký tự được đọc và chuyển cho các biến nhận số liệu vào. Các dấu cách chèn đệm vào sẽ không được tính. Nếu sử dụng tham số này thì cần phải đặt tùy chọn **ADVANCE**='NO'.

iolist: Danh sách các biến sẽ được nhận số liệu từ file.

7.5.2 Lệnh ghi dữ liệu ra file (WRITE)

Cú pháp câu lệnh như sau.

```
WRITE ([UNIT=] unit [, {[ FMT=] fmt  | [ NML=] nml}]
[, ADVANCE=advance] [, ERR=err] [, IOSTAT= iostat]
[, REC=rec] ) iolist
```

Trong đó, dấu gạch đứng có ý nghĩa phân cách các tham số trong một nhóm mà chỉ có thể một trong chúng được phép xuất hiện.

Nếu bỏ qua **UNIT=** thì *unit* phải là tham số đầu tiên và *fmt* hoặc *nml* phải là tham số thứ hai (**FMT=** hoặc **NML=** có thể được bỏ qua). Ngược lại, các tham số có thể xuất hiện theo thứ tự bất kỳ. Trong hai tham số *fmt* và *nml* chỉ được phép xuất hiện một.

unit: Là tên thiết bị lôgic. Khi ghi ra file ngoài *unit* là một biểu thức nguyên gắn với định danh UNIT. Khi ghi ra file trong *unit* phải là xâu ký tự, biến ký tự, mảng hoặc phần tử mảng ký tự,... Nếu *unit* chưa liên kết với một file cụ thể thì lệnh mở file ẩn (implicit) được thực hiện, như câu lệnh sau:

```
OPEN (unit, FILE = ' ', STATUS = 'OLD',&
ACCESS = 'SEQUENTIAL', FORM = form)
```

trong đó *form* là 'FORMATTED' đối với lệnh đọc có định dạng hoặc 'UNFORMATTED' đối với lệnh đọc không định dạng.

fmt: Chỉ thị định dạng, có thể là nhãn câu lệnh FORMAT; biến, hàm hoặc hằng ký tự trong đó kiểu định dạng được chỉ ra trong các cặp dấu nháy đơn (') hoặc nháy kép ("); biến nguyên ASSIGN; hoặc dấu sao (*).

nml: Chỉ ra tên của NAMELIST. Nếu tham số này được chọn thì các tham số *iolist* và *fmt* phải được bỏ qua. File chứa NAMELIST phải là file truy cập tuần tự.

advance: Có dạng ký tự (Character*(*)), chỉ ra cách ghi ra file là tiến (advancing) hay không. Nếu *advance* = 'YES' (ngầm định) sẽ tạo ra đánh dấu vị trí ở cuối bản ghi; nếu *advance* = 'NO' sẽ ghi một phần của bản ghi (tức chưa tạo ra kết thúc bản ghi).

err: Nhãn của câu lệnh thực hiện trong chương trình sẽ được chuyển điều khiển đến nếu gặp lỗi. Nếu bỏ qua tham số này, hiệu ứng lỗi vào/ra sẽ được xác định bởi tham số *iostat*.

iostat: Là tham số kết xuất, ngầm định là một số nguyên (INTEGER(4)), bằng 0 nếu không có lỗi, hoặc bằng một số xác định mã lỗi.

rec: Tham số vào, ngầm định là một số nguyên dương (INTEGER(4)), chỉ số thứ tự bản ghi trong file sẽ được ghi vào file truy cập trực tiếp. Khi sử dụng tham số *rec* thì các tham số *end* và *nml* cần phải bỏ qua. Con trỏ file phải được định vị tại bản ghi *rec* trước khi số liệu được ghi. Giá trị ngầm định của *rec* là số thứ tự bản ghi hiện thời.

iolist: Danh sách các biến sẽ được ghi, liệt kê cách nhau bởi dấu phẩy (,).

7.5.3 Vào ra dữ liệu với NAMELIST

Vào ra dữ liệu bằng NAMELIST là một phương pháp rất hữu hiệu của Fortran. Bằng cách chỉ ra một hoặc nhiều biến trong nhóm tên danh sách ta có thể đọc hoặc ghi các giá trị của chúng chỉ với một câu lệnh đơn giản.

Nhóm NAMELIST được tạo bởi câu lệnh **NAMELIST** có dạng:

```
NAMELIST / namelist / variablelist
```

Trong đó *namelist* là tên nhóm và *variablelist* là danh sách các biến.

Lệnh NAMELIST khi đọc vào sẽ kiểm duyệt tên nhóm trong file NAMELIST. Sau khi tìm thấy tên nhóm nó sẽ kiểm duyệt các câu lệnh gán giá trị cho các biến trong nhóm.

Trong file NAMELIST, các nhóm được bắt đầu bởi dấu và (&) hoặc dấu đôla ($), và kết thúc bằng dấu gạch chéo (/), dấu và (&), hoặc dấu đôla ($).

Từ khóa END có thể xuất hiện liền ngay sau các dấu kết thúc (&) hoặc ($) (không có dấu cách) nhưng không được phép xuất hiện sau dấu gạch chéo (/).

Ví dụ, giả sử có:

```
INTEGER a, b
NAMELIST /mynml/ a, b
...
```

Các lệnh gán trong file NAMELIST sau đây là hợp lệ:

```
&mynml a = 1 /
$mynml a = 1, b = 2, $
$mynml a = 1, b = 2, $end
&mynml a = 1, b = 2, &
&mynml a = 1, b = 2, $END
&mynml
   a = 1
   b = 2
/
```

a. Lệnh đọc nội dung NAMELIST

Cú pháp:

READ (UNIT=*unit*, [NML=] *namelist*)

Trong đó *unit* là thiết bị lôgic lưu trữ thông tin của NAMELIST, có thể là file trên đĩa hoặc bàn phím, *namelist* là tên của NAMELIST. Nếu *unit* là dấu sao (*) thì NAMELIST được nhận từ bàn phím. Trong trường hợp này bạn phải gõ vào theo đúng qui cách. Ví dụ:

```
INTEGER a, b
NAMELIST /mynml/ a, b
read(*,mynml)
WRITE(*,mynml)
END
```

Khi chạy chương trình này bạn phải gõ vào, chẳng hạn:

```
&mynml a = 1, b = 2, /
```

Một ví dụ khác, giả sử NAMELIST của bạn có tên *example* chứa trong file liên kết với thiết bị lôgic 4 (*unit*=4) với nội dung:

```
&example
      Z1 = (99.0,0.0)
      INT1=99
      array(1)=99
      REAL1 = 99.0
      CHAR1='Z'
      CHAR2(4:9) = 'Inside'
      LOG1=.FALSE.
/
```

Bạn có thể dùng câu lệnh sau để đọc nội dung NAMELIST này:

```
READ (UNIT = 4, example)
```

Hoặc, giả sử bạn có chương trình

```
INTEGER, DIMENSION(4) :: A = 7
NAMELIST/MYOUT/A, X, Y
X = 1
Y = 1
PRINT*,'Cho noi dung NAMELIST (A(1:4),X,Y):'
READ( *, MYOUT )
WRITE( *, NML = MYOUT )
END
```

Khi chạy chương trình này bạn nhập vào (từ bàn phím):

```
&MYOUT A(1:2) = 2*1 Y = 3 /
```

và sẽ nhận được (trên màn hình):

```
&MYOUT A = 1 1 7 7, X = 1.0000000, Y = 3.0000000
```

b. Lệnh in nội dung NAMELIST

Để in nội dung các nhóm NAMELIST có thể sử dụng lệnh:

```
WRITE (UNIT=unit, [NML=] namelist)
```

NML= là một tùy chọn, chỉ đòi hỏi phải có nếu các từ khóa khác được sử dụng (chẳng hạn, **END=**).

Ví dụ: Chương trình sau đây gán các giá trị của NAMELIST và in nội dung lên màn hình.

```
INTEGER(1) int1
INTEGER int2, int3, array(3)
LOGICAL(1) log1
LOGICAL log2, log3
REAL       real1
```

```
REAL(8)    real2
COMPLEX    z1, z2
CHARACTER(1)  char1
CHARACTER(10) char2
NAMELIST /example/ int1, int2, int3, &
     log1, log2, log3, real1, real2, &
        z1, z2, char1, char2, array
int1     =  11
int2     =  12
int3     =  14
log1     = .TRUE.
log2     = .TRUE.
log3     = .TRUE.
real1    =  24.0
real2    =  28.0d0
z1       = (38.0, 0.0)
z2      = (316.0d0, 0.0d0)
char1    = 'A'
char2   = '0123456789'
array(1) = 41
array(2) = 42
array(3) = 43
WRITE (*, example)
END
```

Khi chạy chương trình bạn sẽ nhận được trên màn hình:

```
&EXAMPLE
INT1 =          11
INT2 =          12
INT3 =          14
LOG1 =  T
LOG2 =  T
LOG3 =  T
REAL1 =       24.000000
REAL2 =       28.000000
Z1 =            (38.000000,0.000000E+00)
Z2 =          (316.000000,0.000000E+000)
```

```
CHAR1 =  A
CHAR2 =  0123456789
ARRAY =            41           42           43
/
```

Bạn có thể sửa đổi chương trình để ghi NAMELIST vào file.

7.5.4 Một số ví dụ

Ví dụ 1: Chương trình sau đây tạo một file có tên file do bạn xác định, sau đó đọc nội dung từng bản ghi trong file và lần lượt hỏi bạn có xóa hay không. Kết quả trung gian được ghi vào một file nháp. Nội dung của file được tạo sẽ được phục hồi lại từ file nháp này.

```
CHARACTER(80) Name, FileName, Ans
WRITE( *, '(A)', ADVANCE = 'NO' ) "Name of file: "
READ*, FileName
OPEN( 1, FILE = FileName )
OPEN( 2, STATUS = 'SCRATCH' )
write (1,'(A)') 'TEST1 Only'
write (1,'(A)') 'TEST2 Only'
write (1,'(A)') 'TEST3 Only'
rewind(1)
IO = 0
DO WHILE (IO == 0)
  READ( 1, '(A)', IOSTAT = IO ) Name
  print*,Name
  IF (IO == 0) THEN
    WRITE(*,'(A)',ADVANCE='NO')“Xoa khong (Y/N)?"
    READ*, Ans
    IF(Ans/='Y'.AND.Ans/='y') WRITE( 2, * ) Name
  END IF
END DO
REWIND( 2 )
CLOSE( 1, STATUS = 'DELETE' )
OPEN( 1, FILE = FileName )
IO = 0
DO WHILE (IO == 0)
  READ( 2, '(A)', IOSTAT = IO ) Name
  IF (IO == 0) WRITE( 1, * ) Name
```

```
END DO
CLOSE( 1 )
CLOSE( 2 )
END
```

Ví dụ 2: Chương trình sau đây tạo một file tuần tự không định dạng. Chú ý cách truy cập đến các bản ghi của nó.

```
INTEGER, DIMENSION(10) :: A = (/ (I, I = 1,10) /)
INTEGER, DIMENSION(10) :: B = (/ (I, I = 11,20) /)
OPEN( 1, FILE = 'TEST', FORM = 'UNFORMATTED' )
WRITE (1) A     ! ghi A trước
WRITE (1) B      ! ghi B sau
REWIND (1)
A = 0
B = 0
READ (1) B     !  đọc B trước
PRINT*, B
READ (1) A      !  đọc A sau
PRINT*, A
CLOSE (1)
END
```

Ví dụ 3: Chương trình sau đây tạo file truy cập trực tiếp.

```
CHARACTER (20) NAME
INTEGER I
LEN=20
OPEN( 1, FILE = 'TEST.txt', STATUS = 'REPLACE',&
        ACCESS = 'DIRECT', RECL = LEN )
DO I = 1, 6
  PRINT*,' Cho xau ky tu thu ', I
  READ*, NAME
  WRITE (1, REC = I) NAME
END DO
PRINT*,' Cac xau da nhap:'
DO I = 1, 6
  READ( 1, REC = I ) NAME
```

```
  PRINT*,'Xau thu ',I,': ', NAME
END DO
! Ghi de (thay doi) noi dung ban ghi thu 3
WRITE (1, REC = 3) 'Ban ghi thu 3'
PRINT*,' Cac xau sau khi sua doi:'
DO I = 1, 6
  READ( 1, REC = I ) NAME
  PRINT*, NAME
END DO
CLOSE (1)
END
```

Ví dụ 4: Chương trình sau đọc từng ký tự trong từng bản ghi và in ra nội dung và số ký tự của bản ghi. Bạn hãy lưu ý cách sử dụng các tham số trong lệnh READ.

```
CHARACTER ch*1, ST(100)
INTEGER IO, Num, EOR
OPEN( 1, FILE = 'TEST.TXT' )
DO I=1,10
   WRITE(1,'(10I3)') (J,J=I+1,I+10)
END DO
REWIND (1)
IO = 0
DO WHILE (IO /= -1)       ! EOF
! DO WHILE (IO == 0)      ! Khong co loi
Num = 0
do
  READ (1, '(A1)', IOSTAT = IO,&
        ADVANCE = 'NO',EOR=10) ch  ! Doc tung ky tu
  Num = Num + 1
  ST(Num) = ch  ! Luu vao bien ST
end do
10 PRINT*,Num   ! In so ky tu
PRINT*,ST       ! In noi dung ban ghi
END DO
CLOSE (1)
END
```

Ví dụ 5: Chương trình sau đây nhập vào một mảng số nguyên và một mảng số thực với số phần tử tùy ý (trong chương trình chỉ hạn chế tối đa là 10 phần tử). Sau đó in các phần tử của mỗi mảng này trên cùng dòng. Bạn chú ý cách tạo lệnh định dạng FORMAT sao cho co thể in được số phần tử và độ rộng trường tùy ý.

```
INTEGER WI, N(10), ICOUNT, XCOUNT
REAL X(10)
ICOUNT=1
DO
   PRINT*,' CHO GIA TRI N(',ICOUNT,&
          ') (-999=THOAT):'
   READ*,N(ICOUNT)
    IF (N(ICOUNT)==-999.OR.ICOUNT==10) EXIT
   ICOUNT=ICOUNT+1
END DO
XCOUNT=1
DO
   PRINT*,' CHO GIA TRI X(',XCOUNT,
          ') (-999=THOAT):'
   READ*,X(XCOUNT)
   IF (X(XCOUNT)==-999.OR.XCOUNT==10) EXIT
   XCOUNT=XCOUNT+1
END DO
PRINT*,' CHO DO RONG TRUONG SO NGUYEN:'
READ*, WI
WRITE(*, 11) (X(I),I=1,XCOUNT-1)
11 FORMAT(2X,<XCOUNT-1>F8.1)
WRITE(*, 10) (N(I),I=1,ICOUNT-1)
10 FORMAT(<ICOUNT-1>I <WI>)
WRITE(*,*)
END
```

CHƯƠNG 8. MỘT SỐ KIẾN THỨC MỞ RỘNG

8.1 Khai báo dùng chung bộ nhớ

Trong nhiều lớp bài toán, vấn đề dùng chung bộ nhớ sẽ quyết định khả năng giải được của bài toán liên quan đến tài nguyên bộ nhớ và tốc độ của máy tính. Fortran hỗ trợ một vài phương thức dùng chung bộ nhớ, làm tăng khả năng chia sẻ dữ liệu giữa các đơn vị chương trình cũng như làm tăng tốc độ truy cập bộ nhớ. Trong mục này ta sẽ xét hai phương thức dùng chung bộ nhớ là sử dụng lệnh COMMON và lệnh EQUIVALENT.

8.1.1 Lệnh COMMON

Lệnh COMMON có dạng:

COMMON [/[*cname*]/]*objlist* [[,]/[*cname*]/*objlist*] ...

Trong đó:

cname: (Tùy chọn) là tên của khối dùng chung mà các biến trong *objlist* thuộc khối đó. Nếu bỏ qua ta nói đó là khối chung "trắng" ("blank common")

objlist: Là một hoặc nhiều tên biến, tên mảng được phép dùng chung vùng nhớ, chúng được liệt kê cách nhau bởi dấu phẩy (,)

Các biến, mảng trong các khối dùng chung giữa các đơn vị chương trình phải tương ứng về vị trí và độ dài. Các mảng phải có cùng kích thước. Các biến trong khối chung không thể là đối số hình thức, mảng động, tên hàm,... Chúng cũng không thể là những hằng được khai báo bởi lệnh PARAMETER.

Tác động của khối dùng chung là ở chỗ, khi các đơn vị chương trình khác nhau có khai báo dùng chung cùng một tên khối chung *cname* thì, mặc dù danh sách tên biến trong *cname* ở các đơn vị chương trình có thể khác nhau, chúng vẫn tham chiếu đến cùng những vùng bộ nhớ.

Ví dụ, giả sử ta có các đơn vị chương trình sau:

```
PROGRAM MyProg
COMMON i, j, x, k(10)
COMMON /mycom/ a(3)
A=5
CALL MySub
PRINT*,A
END

SUBROUTINE MySub
COMMON je, mn, z, idum(10)
COMMON /mycom/ B(3)
B = B + 5
```

```
END
```

Khi chương trình thực hiện, các biến `je, mn, z, idum` trong **MySub** sẽ tham chiếu đến các vùng bộ nhớ tương ứng của các biên `I, J, X, K` trong **MyProg**; biến `B` trong **MySub** và biến `A` trong **MyProg** cùng dùng chung một vùng bộ nhớ. Do đó, lời gọi **CALL MySub** trong đó `B` bị thay đổi cũng có nghĩa là `A` bị biến đổi.

8.1.2 Lệnh EQUIVALENT

Phương thức khai báo dùng chung EQUIVALENCE làm cho hai hoặc nhiều biến hoặc mảng chiếm cùng một vùng bộ nhớ. Cú pháp câu lệnh có dạng:

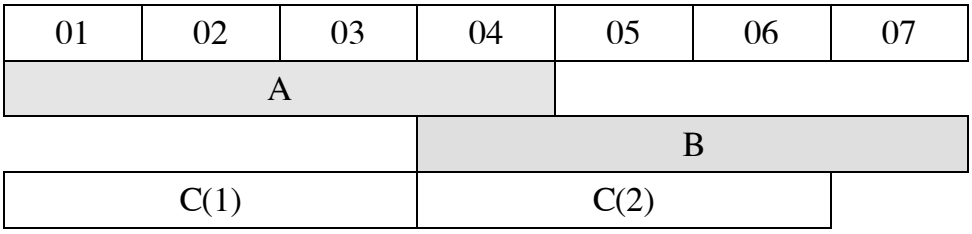
```
EQUIVALENCE (nlist) [ , (nlist)] ...
```

Trong đó *nlist* là hai hoặc nhiều hơn các biến, mảng hoặc phần tử mảng, viết cách nhau bởi dấu phẩy (,). Các chỉ số mảng cần phải là những hằng số nguyên và phải nằm trong miền giá trị của kích thước mảng. Những tên biến mảng không có chỉ số được ngầm hiểu là phần tử có chỉ số đầu tiên của mảng. Tất cả các phần tử trong *nlist* đều có cùng vị trí đầu tiên trong vùng bộ nhớ. Để minh họa ta hãy xét ví dụ sau.

Giả sử có khai báo:

```
CHARACTER  a*4, b*4, c(2)*3
EQUIVALENCE (a, c(1)), (b, c(2))
```

Khi đó định vị bộ nhớ sẽ có dạng:



Nếu không khai báo dùng chung, dung lượng bộ nhớ phải cấp cho các biến A, B, C sẽ là 4 + 4 + 2*3 = 14 byte. Nhưng khi sử dụng khai báo dùng chung bằng EQUIVALENT, dung lượng bộ nhớ cấp cho 3 biến này chỉ cần 7 byte như trên hình vẽ. Từ hình vẽ, có thể hình dung rằng, 3 ký tự của phần tử C(1) dùng chung bộ nhớ với 3 ký tự đầu của xâu A; ký tự thứ tư của xâu A, ký tự thứ nhất của xâu B và ký tự thứ nhất của phần tử C(2) dùng chung 1 byte; hai cặp ký tự tiếp theo của xâu B và của C(2) dùng chung các byte thứ 5 và thứ 6; chỉ có ký tự cuối cùng của xâu B là không dùng chung. Do đó, tùy theo biến nào bị thay đổi giá trị cuối cùng mà các biến khác sẽ bị biến đổi theo. Ví dụ, trong khai báo trên, nếu tuần tự câu lệnh là:

```
A='abcd'
B='efgh'
C(1)='ijk'
C(2)='LMN'
```

thì kết quả cuối cùng sẽ là:

```
C(1)='ijk'      B='LMNh'
```

```
C(2)='LMN'     A='ijkL'
```

Một ví dụ khác, giả sử ta có chương trình:

```
INTEGER M(6), N(10), MN(8)
EQUIVALENCE (N,M), (N(7),MN)
N=4
M=3
MN=5
PRINT*,M
PRINT*,N
PRINT*,MN
END
```

Khi chạy chương trình này bạn sẽ nhận được:

```
MN=(5, 5, 5, 5, 5, 5, 5, 5)
N =(3, 3, 3, 3, 3, 3, 5, 5, 5, 5)
M =(3, 3, 3, 3, 3, 3)
```

8.2 Chương trình con BLOCK DATA

Chương trình con BLOCK DATA là một loại đơn vị chương trình cho phép xác định các giá trị khởi tạo cho các biến trong các khối dùng chung. Chương trình con này chứa các câu lệnh không thực hiện. Cấu trúc chương trình có dạng:

```
BLOCK DATA [name]
   [specifications]
END [BLOCK DATA [name]]
```

Thông thường các biến được khởi tạo bằng các lệnh DATA. Các biến trong khối dùng chung cũng có thể được khởi tạo. BLOCK DATA có thể chứa các câu lệnh: COMMON, USE, DATA, PARAMETER, DIMENSION, POINTER, EQUIVALENCE, SAVE, IMPLICIT.

Ví dụ:

```
COMMON /WRKCOM/ X, Y, Z (10,10)
PRINT*,X
PRINT*,Y
PRINT*,Z
END
BLOCK DATA WORK
   COMMON /WRKCOM/ A, B, C (10,10)
   DATA A /1.0/, B /2.0/ , C /100*5.0/
END BLOCK DATA WORK
```