Lenguajes de Programación

Dr. Carlos A. Coello Coello
Departamento de Computación
CINVESTAV-IPN
ccoello@cs.cinvestav.mx

Subprogramas

- Los subprogramas motivan el diseño modular, el cual es un principio de ingeniería de software muy importante.

- Bajo este principio, la idea es partir un problema grande en varios subproblemas pequeños y escribir un módulo para resolver cada uno de ellos como una entidad separada.

Subprogramas

- Un buen módulo debe ocultar detalles innecesarios y comunicarse sólo a través de sus interfaces (es decir, sus parámetros y valores de retorno). Asimismo, sus modificaciones no debieran afectar el resto del programa.

Subprogramas

- George A. Miller publicó en 1956 el resultado de un estudio que indicaba que el cerebro humano no es capaz de retener en su memoria de corto plazo más de 7 cosas a la vez (realmente es 7±2, lo que significa que algunas personas pueden retener hasta 9 cosas, pero otras sólo pueden retener 5).

Subprogramas

- Por lo tanto, otro principio importante al escribir subrutinas es que no deben pasarse más de 7 parámetros. Si se requieren más de 7, entonces puede ser necesario desarrollar más módulos.

Subprogramas

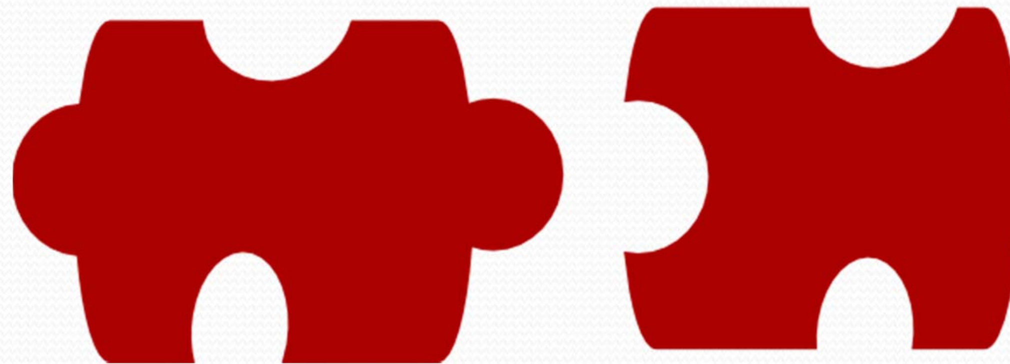
```
Subroutine ligd(n,x)
!DEC$ ATTRIBUTES DLLEXPORT::ligd
!DEC$ ATTRIBUTES C, REFERENCE, ALIAS:'ligd_'::ligd

 implicit none
 integer n,i
 real*8 x(n)

 do i=1,n
  x(i)=x(i)*x(i)
 end do
 return
End Subroutine ligd
```

- Los subprogramas motivan el desarrollo de bibliotecas, puesto que permiten la compilación separada. Esto proporciona a los programadores la ventaja de ser capaces de reutilizar sus propias funciones y algunas escritas por otras personas, en vez de tener que re-escribir todo de nuevo cada vez que se desarrolla un nuevo programa.

Subprogramas



- La idea del reuso de software (que algunos llaman "*programming in the large*") permite a los programadores desarrollar grandes sistemas de software usando módulos precompilados pequeños, los cuales fueron cuidadosamente depurados y refinados y que son totalmente independientes del software que se está desarrollando.

Mecanismos de Paso de Parámetros

- Supongamos que tenemos la siguiente subrutina:

```
SUBROUTINE SUMA(X,Y,Z)
Z=X+Y
RETURN
END
```

Mecanismos de Paso de Parámetros

- Supongamos que invocamos el subprograma anterior de la siguiente manera:

CALL SUMA(A1,B1,VALOR)

- Cuando se invoca esta subrutina, los parámetros originales (*formals*) se asocian con los parámetros de invocación (*actuals*).

- En este caso, la **X** se asocia con **A1**, la **Y** con **B1** y la **Z** con **VALOR**.

Mecanismos de Paso de Parámetros

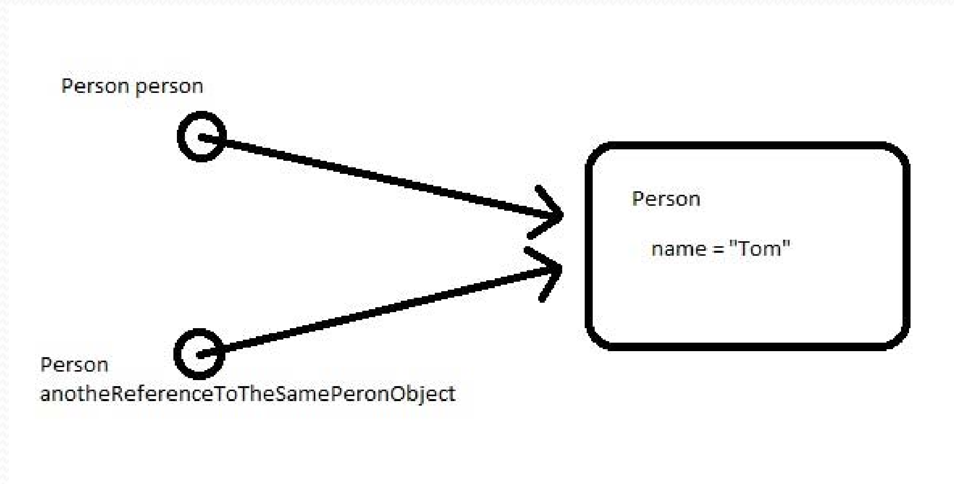
- FORTRAN pasa todos los parámetros que se envían a los subprogramas "por referencia".
- Esta es una mala decisión de diseño porque tiene algunas consecuencias peligrosas.

Mecanismos de Paso de Parámetros

- Usando paso de parámetros por referencia, todos los parámetros pueden ser usados para entrada o salida, dado que lo que se pasa no es una copia del valor de un parámetro, sino su dirección de memoria.

- El problema con esto es que una variable que debía ser sólo de entrada podría ser alterada accidentalmente. A esto se le llama “efecto colateral” (*side-effect*).

Mecanismos de Paso de Parámetros



- El paso de parámetros por referencia puede implementarse muy eficientemente en términos de tiempo y espacio. Esto es particularmente útil al pasar arreglos, porque no se tiene que pasar más que una dirección de memoria (la del primer elemento del arreglo).

Mecanismos de Paso de Parámetros

```
FUNCTION MEDIA(ARR, N)
DIMENSION ARR(N)
SUMA=0.0
DO 100I=1,N
        SUMA=SUMA+ARR(I)
100 CONTINUE
MEDIA=SUMA/FLOAT(N)
RETURN
END
```

Mecanismos de Paso de Parámetros

- Para invocar la función anterior hacemos:

```
DIMENSION DATOS(50)

PROM = MEDIA(DATOS, 50)
```

- Un problema con lo anterior es que el tamaño del arreglo debe pasarse como parámetro. Esto se debe a que ese valor es necesario para el ciclo **DO** que se efectúa. Sin embargo, el usuario puede pasar un valor equivocado.

Mecanismos de Paso de Parámetros

- Si se pasa un valor muy pequeño, se producirá un error que será un tanto difícil de rastrear.

- Si se pasa un valor muy grande, puede comprometerse la seguridad del lenguaje, ya que FORTRAN permite que los programadores usen índices que estén más allá del límite de un arreglo definido.

Mecanismos de Paso de Parámetros

- Otro peligro mayor del paso por referencia es que es posible alterar valores de constantes si éstas se pasan como parámetros. Esto produce errores sumamente difíciles de encontrar.

Mecanismos de Paso de Parámetros

```
SUBROUTINE CAMBIO(A)
A=5
RETURN
END
```

- Este subprograma simplemente cambia el valor del parámetro que se le pasa y lo hace **5**.

Mecanismos de Paso de Parámetros

Por ejemplo:

CALL CAMBIO(X)

Hace que **X** tome el valor de **5**.

Sin embargo, ¿qué pasa si hace lo siguiente?

CALL CAMBIO(3)

Esto cambiará el valor de la constante 3 por 5. De tal forma, si hace M=3+3, ¡ la respuesta será 10!

Mecanismos de Paso de Parámetros

- Otro ejemplo clásico es el siguiente:

```
SUBROUTINE EJEMPLO(X,Y)
   -              .
   -              .
RETURN
END
```

Mecanismos de Paso de Parámetros

Si ahora hacemos:

```
CALL EJEMPLO(A,A)
```

los 2 parámetros X, Y sirven como alias para el mismo valor.

Éste y otros problemas todavía más complejos pueden surgir cuando se usa el paso de parámetros por referencia.

Mecanismos de Paso de Parámetros

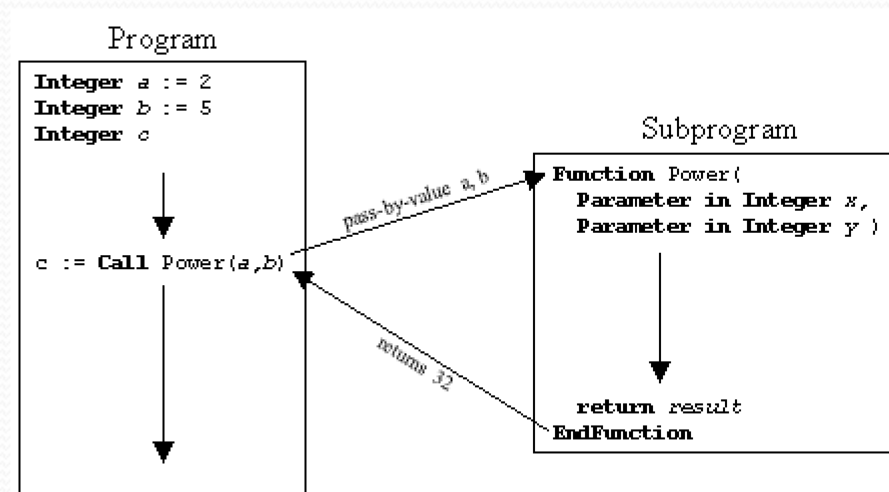
- Debido a sus problemas potenciales, el paso de parámetros por referencia no es una buena elección a menos que se permitan otros mecanismos de paso de parámetros en el lenguaje.
- La mayor parte de los lenguajes de programación modernos permiten que el programador distinga entre variables de entrada y de salida, a fin de evitar los problemas de paso de parámetros del FORTRAN.

Mecanismos de Paso de Parámetros

- Aunque la mejor solución es la mencionada anteriormente, otra solución posible al problema de FORTRAN es usar otro mecanismo de paso de parámetros que se conoce como "paso por valor-resultado" (también llamado "paso por copia").

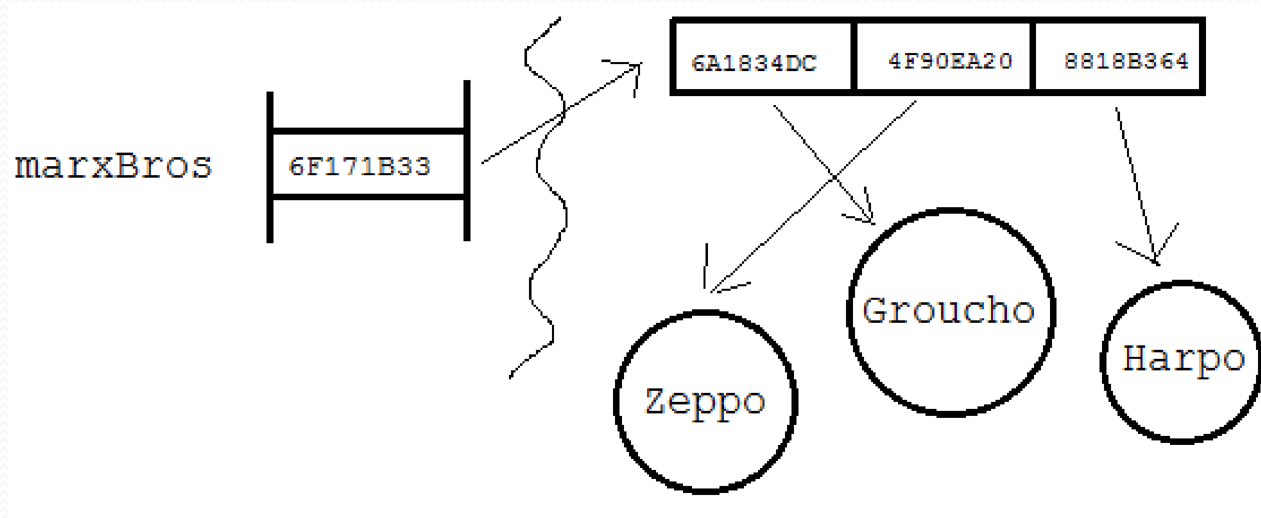
- Bajo este esquema, se pasa una copia del valor del parámetro con que se invoca una subrutina. Dentro de la subrutina, ese valor actúa entonces como una variable local. Al terminar la ejecución del subprograma, se regresa el valor final del parámetro interno al parámetro invocador.

Mecanismos de Paso de Parámetros



- De esta manera, puede omitirse el copiado de un parámetro en la etapa final, si el valor que se envió originalmente era constante. Asimismo, el subprograma nunca tiene acceso directo a las variables del ambiente de invocación, sino únicamente a una copia del valor con que se le llama.

Mecanismos de Paso de Parámetros



- Este mecanismo no está exento de fallas, ya que todavía puede presentarse la alteración accidental de una variable dentro de un subprograma. Sin embargo, elimina el riesgo de comprometer la integridad del lenguaje de programación.

Mecanismos de Paso de Parámetros

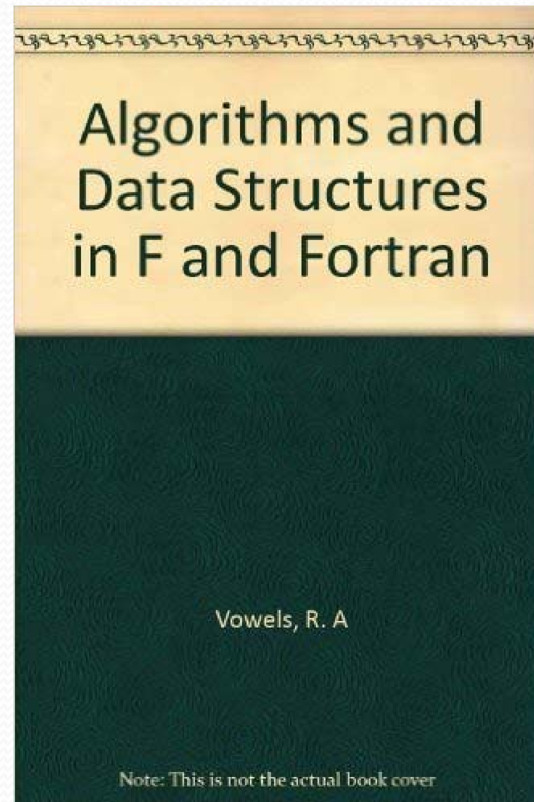
- Algunos de los problemas principales con el paso de parámetros por referencia son los siguientes:
- 1) Los accesos a los parámetros originales del subprograma serán más lentos debido a que se requiere un nivel más de direccionamiento indirecto cuando se transmiten datos.
- 2) Si se requiere una comunicación unidireccional hacia el subprograma invocado, pueden efectuarse cambios accidentales al valor de la variable que se transmite a éste.

Mecanismos de Paso de Parámetros

- 3) Pueden crearse fácilmente “alias” porque se crean rutas a los subprogramas invocados, lo que amplía el acceso a variables no locales.

- 4) El pasar una dirección de memoria da acceso a ésta, y cualquier cambio a dicha posición podría afectar otras partes del programa, produciéndose un efecto similar al uso de variables globales.

Estructuras de Datos

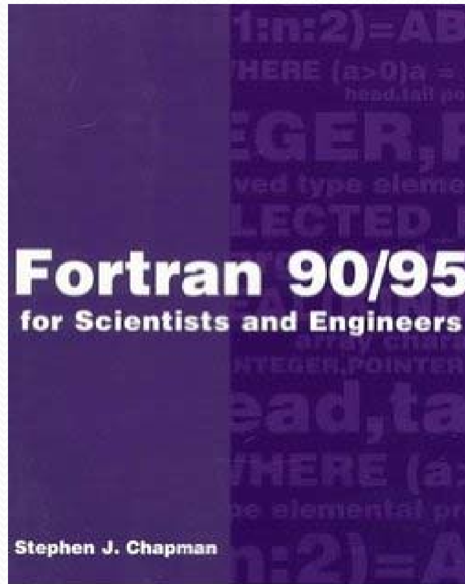


- Las estructuras de datos de FORTRAN fueron inspiradas en las matemáticas. Por ello, no debe sorprendernos que su estructura de datos principal sean los escalares numéricos.

Estructuras de Datos

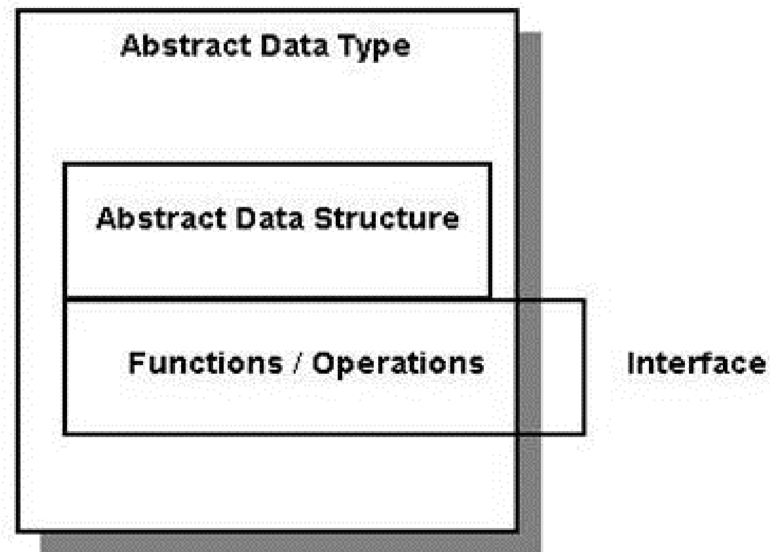
- Además de los números tradicionales de punto flotante y los enteros, FORTRAN proporcionaba:
  - Números de doble precisión (DOUBLE PRECISION)
  - Números complejos (COMPLEX)
  - Valores Booleanos (LOGICAL)

Estructuras de Datos



- Es importante tener en mente que todas las operaciones numéricas en FORTRAN son independientes de la representación. Esto significa que dependen en las propiedades lógicas o abstractas de los valores de los datos y no en los detalles de su representación en una computadora en particular.

Estructuras de Datos



- Este es un concepto muy importante en los lenguajes de programación modernos. A un conjunto de valores de datos junto con un conjunto de operaciones que actúan sobre dichos valores sin alusión a su representación se le conoce como tipo de datos abstracto (*abstract data type*).

Representando Números de Punto Flotante

- Una de las representaciones binarias más populares de los números reales es el estándar del IEEE:

Signo: 1(-) o 0 (+) 8 bits exponente + 12 (8 bits) mantisa (23 bits)

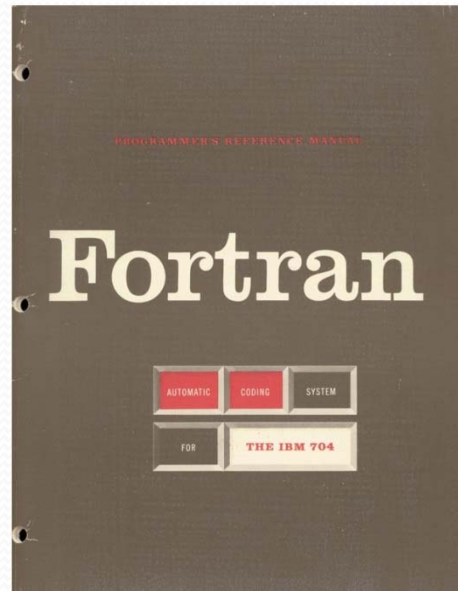
Representando Números de Punto Flotante

- En el estándar del IEEE, un número de simple precisión consiste de 32 bits, de los cuales uno se usa para el signo, ocho se usan para el exponente (en notación exceso-127) y 23 bits se usan para la mantisa, la cual se normaliza en la forma 1.bb…b, donde sólo se almacena bb…b.

Representando Números de Punto Flotante

- Sin embargo, existen otras representaciones posibles y éstas pueden variar de computadora a computadora.
- Los tipos de datos abstractos nos permiten precisamente el despreocuparnos de esa representación interna de los números y concentrarnos en las operaciones a realizarse entre ellos.

Sobrecarga de Operadores Aritméticos



- En FORTRAN I no había soporte para expresiones mixtas, lo que quiere decir que no era posible mezclar números de tipos diferentes en la misma expresión matemática. Se requería del uso de un mecanismo de "coerción" para reducir un tipo numérico a otro.

Sobrecarga de Operadores Aritméticos

- Un ejemplo de “coerción” es el siguiente:

  - X + **FLOAT**(A)

donde suponemos que X es un real y A es un entero. **FLOAT** convierte el entero a real.

Sobrecarga de Operadores Aritméticos

- Sin embargo, en versiones posteriores de FORTRAN se incorporó un mecanismo de coerción implícita, de manera que el usuario pudiese mezclar tipos en una expresión y éstos pudiesen ser convertidos automáticamente a tipos más elevados sin ninguna intervención humana.

Sobrecarga de Operadores Aritméticos

- Los siguientes son ejemplos de coerción implícita (X es real, A es entero):
  - X + A ➔ X + FLOAT(A)
  - A + X ➔ FLOAT(A) + X

  - X = A ➔ X = FLOAT(A)
  - A = X ➔ A = IFIX(X)

Sobrecarga de Operadores Aritméticos

- La coerción implícita sigue las reglas convencionales de las matemáticas, ya que los enteros son considerados un subconjunto de los reales y los reales son considerados un subconjunto de los números complejos.

- De tal forma, si se mezclan reales y enteros en una expresión algebraica, los enteros se convierten a reales.

Sobrecarga de Operadores Aritméticos

- De hecho, en versiones posteriores de FORTRAN era posible realizar coerciones más complejas. Por ejemplo (A es entero, X es real):
  - A=X+A ➔A = **IFIX**(X+FLOAT(A))

Sobrecarga de Operadores Aritméticos

- Algunos lenguajes de programación como C y Pascal no soportan coerción implícita, mientras que otros como el BASIC, el PL/I y el Algol-68 sí lo hacen.
- Cuando se define un operador de manera que pueda operar con distintos tipos de datos, decimos que está "sobrecargado".

Sobrecarga de Operadores Aritméticos

- La sobrecarga de operadores es una forma de "polimorfismo".

- Este tema lo estudiaremos en más detalle cuando hablemos del paradigma de orientación a objetos.

Caracteres

- Hasta antes de FORTRAN 77, los compiladores de FORTRAN no contaban con un tipo **CHARACTER**, y se usaban enteros para representarlos. Esto violaba el **Principio de Seguridad**:

> Ningún programa que viole la definición del lenguaje o su propia estructura supuesta, debiera escapar detección.

Caracteres

- Las cadenas de caracteres se definían a través de las llamadas "constantes de Hollerith". Por ejemplo:

6HCARLOS

representa la cadena de 6 caracteres 'CARLOS'.

Caracteres

- La sintaxis es la siguiente:

1) El número de caracteres de la cadena

2) La letra H

3) La cadena de la longitud pre-especificada

Caracteres

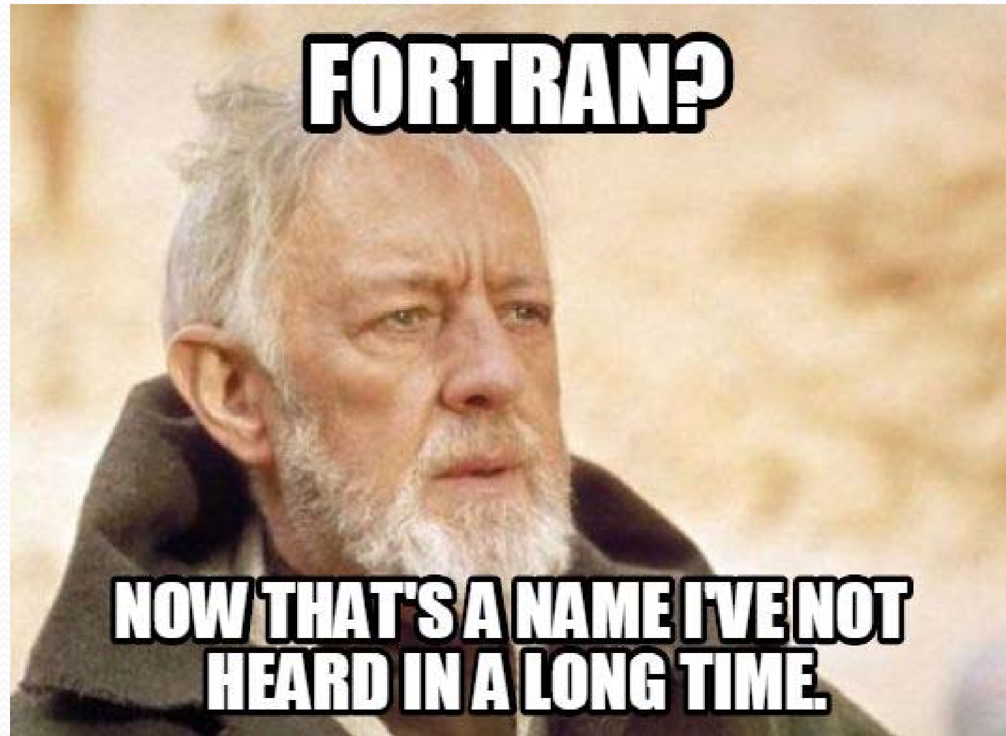
- Las cadenas de caracteres no eran ciudadanos de primera clase en FORTRAN. Esto significa que no podían utilizarse de todas las maneras en las que esperaríamos que se hiciera. Esto viola el **Principio de Regularidad**:

Las reglas regulares, sin excepciones, son más fáciles de aprender, usar, describir e implementar.

Caracteres

- Por ejemplo, las cadenas de caracteres no podían asignarse a variables, ni tampoco podían ser objeto de comparaciones entre sí.
- Esto motivó a muchos programadores a escribir sus propias funciones para manipular cadenas, sacrificando la portabilidad del lenguaje.

Caracteres



- Otro problema es que FORTRAN permitía pasar una cadena en lugar de un entero, dado que internamente se representaban de la misma manera. Esto daba lugar a efectuar operaciones que carecían por completo de sentido.

Caracteres

- Por ejemplo:

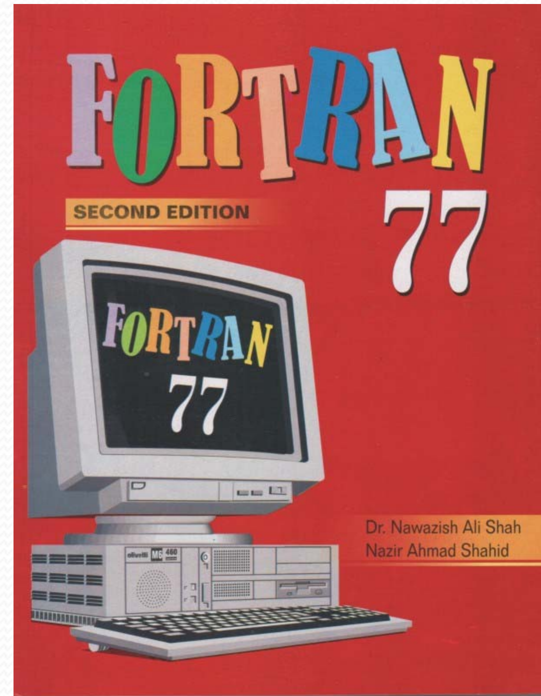
```
FUNCTION SUMA(X)
SUMA=X+5
RETURN
END
```

- FORTRAN nos permitía escribir:
  - VAL = SUMA(6HCARLOS)

Caracteres

- El sumar 5 a la cadena 'CARLOS' carece por completo de sentido, pero era permitido por el compilador de FORTRAN.
- Un lenguaje moderno que permite esta clase de cosas es C, debido a su sistema de tipos débil.

Caracteres

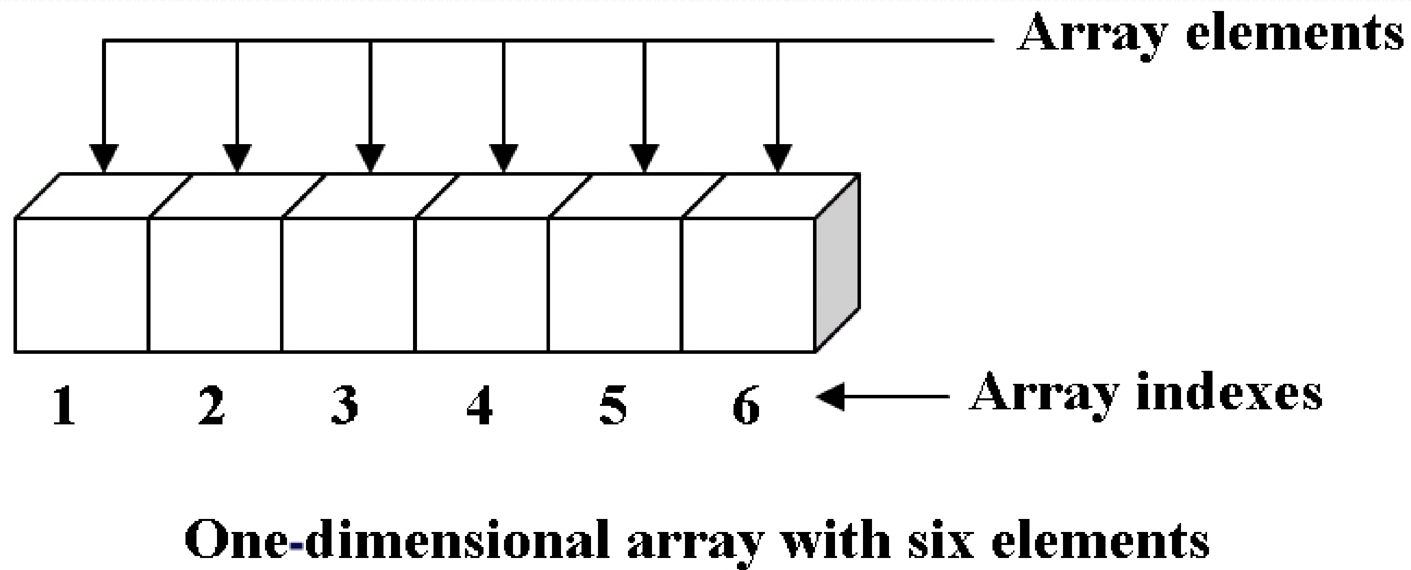


- FORTRAN 77 introdujo el tipo **CHARACTER**, haciendo a las cadenas de caracteres ciudadanos de primera clase. Esto resolvió el problema de seguridad que esta excepción introducía en versiones previas del lenguaje.

Arreglos

- Los arreglos en FORTRAN estaban implementados de manera muy eficiente, pero tenían severas limitaciones.
- Por ejemplo, los arreglos eran estáticos, lo que significa que su tamaño debía conocerse en tiempo de compilación y una vez definido, no podía modificarse.

Arreglos



- Los arreglos estáticos no son muy flexibles y tienen la desventaja adicional de desperdiciar memoria cuando tiene que usarse más de un arreglo en una cierta aplicación y ya no se necesitarán arreglos que fueron definidos previamente.

Arreglos

- Otra limitante de los arreglos es que sólo podían ser de un máximo de 3 dimensiones (en FORTRAN 77 estaban limitados a 7 dimensiones). Esto viola el **Principio del Cero-Uno-Infinito**:

> Los únicos números razonables en un lenguaje de programación son cero, uno e infinito.

Arreglos

- Aunque no es usual en la práctica utilizar arreglos de más de 3 dimensiones, esta limitación es una excepción que el usuario debe recordar y, por tanto, es indeseable.

Arreglos

- En las primeras versiones de FORTRAN las únicas expresiones permisibles como índices de un arreglo eran:

c

v

v+c o v - c

c*v

c*v+c' o c*v-c'

donde **c** y **c'** eran constantes enteras, y **v** era una variable entera.

Arreglos

- Por lo tanto:

  A(2)
  A(I)
  A(I-1)
  A(2*I-1)

eran expresiones válidas

Arreglos

- Pero:

  A(1+I)
  A(I-J)
  A(100-1)
  A(I*J)

no eran expresiones válidas.

Arreglos

- Esta restricción aparentemente arbitraria estaba motivada por el hecho de que de otra forma no habría sido posible optimizar el código en ensamblador que generaba el compilador de FORTRAN.

- Esa era la misma razón por la que se limitaban los arreglos a 3 dimensiones.

- El permitir más de 3 dimensiones complica (exponencialmente) la posibilidad de generar código eficiente en ensamblador para los índices.

Variables no declaradas

- Un aspecto peculiar de FORTRAN es que permitía el uso de variables no declaradas en un programa.
- La intención de este mecanismo era buena: hacer que un programador no tuviera que declarar las variables que usaría.

Variables no declaradas

- El problema inicial, sin embargo, es cómo saber el tipo de una variable no declarada.

- FORTRAN resolvía este problema, declarando como entera cualquier variable cuyo nombre comenzara con las letras comprendidas entre la I y la N. Todo lo demás se declaraba como **REAL**.

Variables no declaradas

- Esta convención seguía los lineamientos de las matemáticas, pero orillaba a los programadores a usar nombres esotéricos para sus variables, a menos que éstas fuesen declaradas.

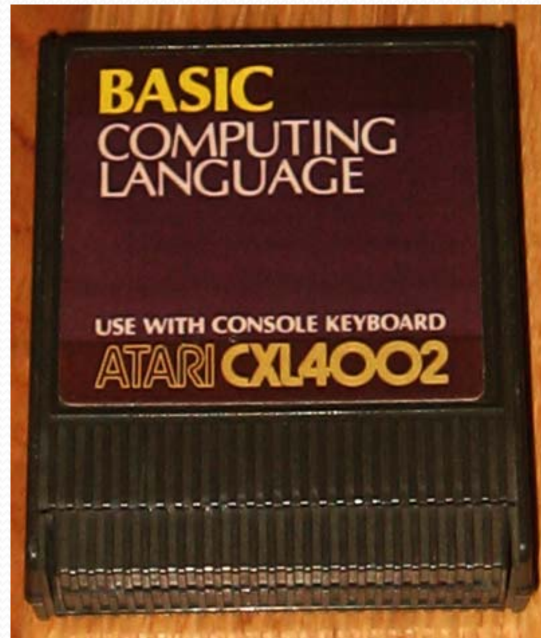
- Además, podían producirse errores bastante extraños. Por ejemplo, supongamos que la variable CONT ha sido declarada como entera y queremos incrementarla en uno. Sin embargo, nos equivocamos y escribimos:

```
CONT = COMT+1
```

Variables no declaradas

- Como la variable COMT no está declarada y su nombre empieza con C, se considerará como **REAL**. De tal forma, FORTRAN asignará a COMT el valor de una posición aleatoria de memoria, generando un error difícil de rastrear.

Variables no declaradas



- Debido a éste y otros problemas con las variables no declaradas, éstas no son permisibles en la mayoría de los lenguajes estructurados modernos (una excepción es el BASIC, que permitía variables no declaradas en varias de sus versiones).

Alcance global y local en los subprogramas

- El 'alcance' (*scope*) del nombre de una variable se refiere a la visibilidad de este dentro de un programa.

- En FORTRAN, los nombres de los subprogramas tienen un alcance global, ya que requieren ser vistos desde cualquier parte del programa principal (o desde otro subprograma).

Alcance global y local en los subprogramas

- En contraste, las variables declaradas dentro de un subprograma tienen un alcance local. Esto quiere decir que sólo pueden ser vistas dentro del mismo subprograma donde fueron declaradas.

Alcance global y local en los subprogramas

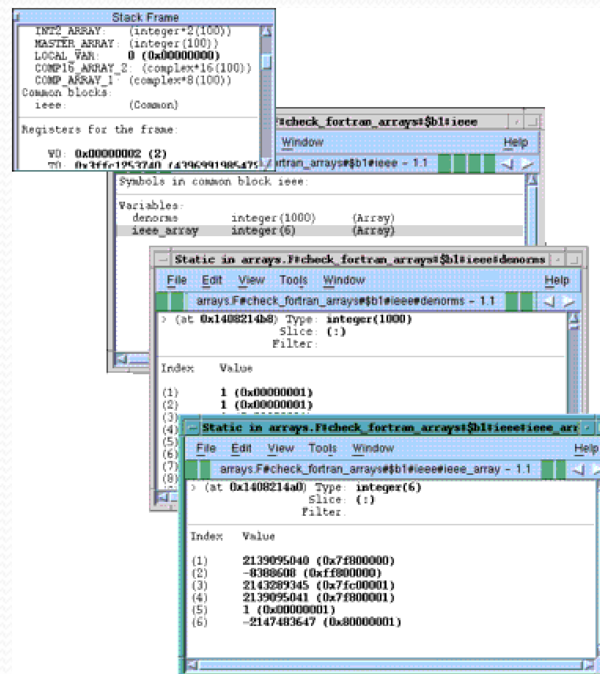
- Esto sigue un principio que originó la idea de la programación orientada a objetos: el **ocultamiento de información**.

- Sólo queremos dar acceso al usuario a la información que requiere para comunicarse con el subprograma (o sea, sus parámetros), pero queremos ocultar lo demás para evitar efectos colaterales indeseables.

Compartiendo Información entre Subprogramas

- Debe observarse, sin embargo, que resulta complicado poder compartir datos entre varios subprogramas si el único medio de comunicación con ellos son los parámetros.
- Por ejemplo, si quisiéramos compartir varios arreglos, tendríamos que pasarlos explícitamente a cada subprograma. Esto hace que la invocación de los subprogramas se vuelva confusa y podría ser fuente de errores serios.

Compartiendo información entre subprogramas



- Para remediar el problema anterior, es posible declarar bloques tipo **COMMON** en FORTRAN, los cuales permiten compartir áreas de memoria entre subprogramas.

Compartiendo información entre subprogramas

```
C  PROGRAMA PRINCIPAL
                .
CALL FUNCT1(NUMERO, POS, VAL, RESULTADO)
                .
END
SUBROUTINE FUNCT1(N,P,V,R)
COMMON/BLOCK/DIRECC(50), TELS(50), NOM(50)
                .
END

SUBROUTINE FUNCT2(A,B,C)
COMMON/BLOCK/DIRECC(50), TELS(50), NOM(50)
                .
END
```

Compartiendo información entre subprogramas

- Sólo las subrutinas que usen una declaración **COMMON** estarán compartiendo el mismo bloque de memoria (cuyo nombre se indica entre las diagonales).
- La principal motivación de **COMMON** es remediar el problema de compartir arreglos entre varios subprogramas, dado que FORTRAN no permite comunicarse con las subrutinas más que a través de sus parámetros.

Compartiendo información entre subprogramas

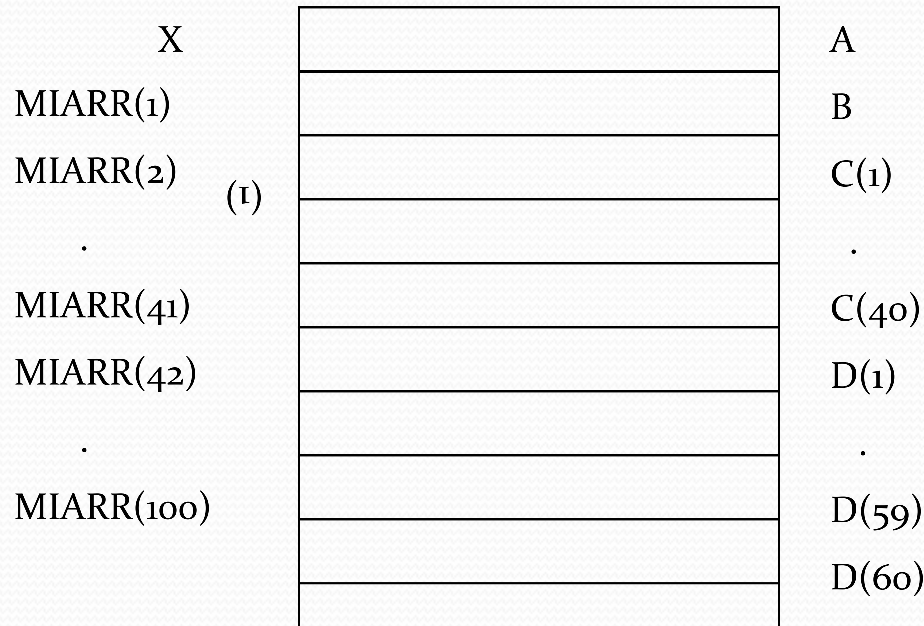
- Sin embargo, este tipo de mecanismo tiene varios problemas. Por ejemplo, FORTRAN no checa si existen diferentes declaraciones de **COMMON** que son iguales (la llamada "**equivalencia estructural**"), por lo que es posible hacer cosas como la siguiente:

En SUB1: **COMMON**/BL/X, MIARR(100)

En SUB2: **COMMON**/BL/A,B,C(40),D(60)

Compartiendo información entre subprogramas

- Internamente, el bloque de memoria BL sería asignado de la siguiente manera:



Compartiendo información entre subprogramas

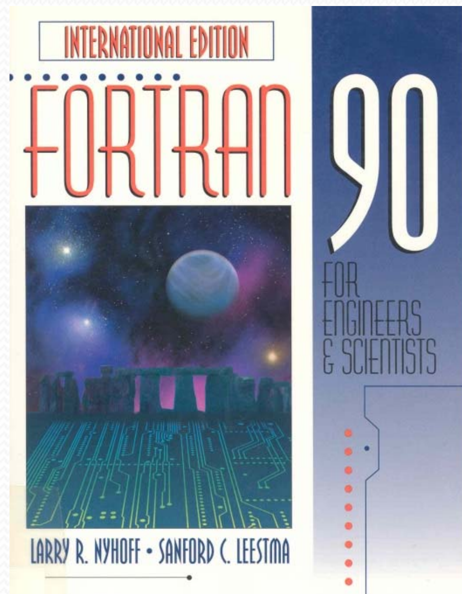
- Esto produce un problema al que se denomina "*aliasing*", el cual consiste en identificar dos regiones distintas de memoria bajo el mismo nombre.

- Como puede verse en el ejemplo anterior, el mapeo de memoria es totalmente caótico y no hay correspondencia entre las 2 declaraciones.

Compartiendo información entre subprogramas

- Aunque este tipo de cosas podrían hacerse intencionalmente para ahorrar memoria, es un mal hábito de programación y se considera una falla grave que viola el sistema de tipos de FORTRAN.

Compartiendo información entre subprogramas



- El problema se agrava si las declaraciones comunes consumen espacios distintos de memoria (por ejemplo, enteros contra números reales). Esa es la razón por la que el comité ANSI decidió eliminar este mecanismo del lenguaje a partir del FORTRAN 90.

Compartiendo información dentro de un subprograma

- Existe otra declaración similar, llamada **EQUIVALENCE**, la cual se usa para compartir datos dentro de un mismo subprograma.

- Su motivación principal era que dos arreglos pudiesen compartir la misma zona de memoria, con la finalidad de economizar espacio.

Compartiendo información dentro de un subprograma

- Ejemplo:

  DIMENSION ARR1(100), ARR2(300)
  EQUIVALENCE (ARR1(1), ARR2(2))

- Esto hacía que los arreglos ARR1 y ARR2 compartieran la misma zona de memoria.

Compartiendo información dentro de un subprograma

- Este mecanismo permite al programador realizar un "*aliasing*" explícito.
- Si bien su motivación era válida en los tiempos en que se originó FORTRAN, con la caída en los precios de las memorias en los 1970s, este mecanismo se volvió automáticamente obsoleto.

Compartiendo información dentro de un subprograma

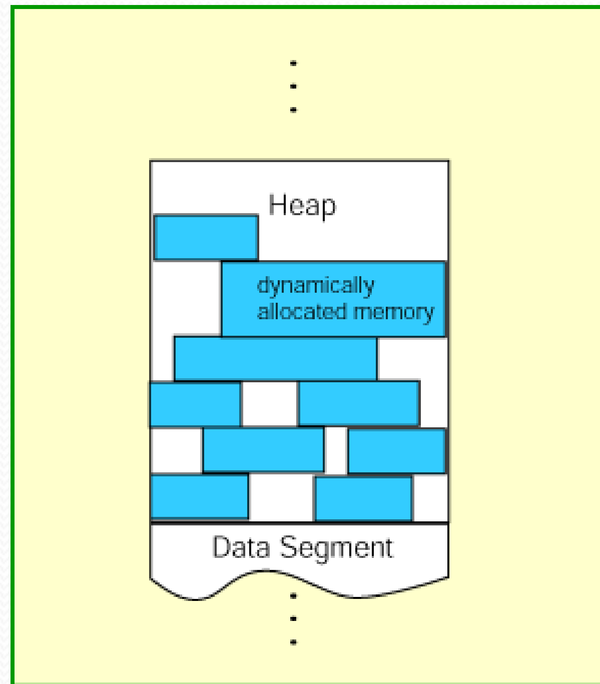
- El uso de **EQUIVALENCE** permite corromper el sistema de tipos.

- Puede hacerse, por ejemplo, un **EQUIVALENCE** entre una variable lógica y una real. Esto permitiría a un programador usar operaciones lógicas para acceder a porciones de la representación de punto flotante de un número.

Compartiendo información dentro de un subprograma

- Esto permitiría realizar una forma primitiva de programación de sistemas en FORTRAN, pero los programas resultantes no serían transportables a otra arquitectura, ya que dependerían de la representación interna de la computadora donde se escribieran.
- Asimismo, ponen en riesgo la integridad del sistema de tipos del lenguaje.

Compartiendo información dentro de un subprograma



- Esa es la razón por la que los esquemas de manejo dinámico de memoria que permiten los lenguajes estructurados modernos son la alternativa usada hoy en día para hacer un uso eficiente de la memoria.

Convenciones léxicas

- Existen 3 convenciones léxicas principales para las palabras de un lenguaje de programación:
- 1) **Palabras reservadas:** Las palabras usadas por el lenguaje están reservadas, lo que significa que no puede usarla el programador para denotar identificadores.

Convenciones léxicas

- Este es el enfoque más comúnmente adoptado en los lenguajes de programación modernos, aunque es un poco confuso para los programadores novatos, si bien hace que el código sea más legible.

Convenciones léxicas

- 2) **Palabras Clave:** Las palabras usadas por el lenguaje están marcadas de una manera no ambigua (por ejemplo, se les enmarca entre comillas). Esto suele ser difícil de escribir y no resulta muy legible a menos que se use un tipo de letra distinto para distinguirlas. Este fue el enfoque usado en ALGOL-60.

Convenciones léxicas

- 3) **<u>Palabras Clave en Contexto</u>:** Este es el enfoque usado en FORTRAN (y PL/I). Las palabras usadas por el lenguaje son sólo consideradas palabras clave en aquellos contextos en los que se espera que lo sean. De lo contrario, se les trata como identificadores.

Convenciones léxicas

- Esto puede producir algunos problemas. Por ejemplo:

  DO10I=1.100

- se interpretaría como que el usuario asignó a la variable DO10I el valor 1.100, aunque probablemente el programador quiso realizar un ciclo de 1 a 100.

Convenciones léxicas

- El usar un punto en vez de una coma hizo que el contexto cambiara. Esto puede producir errores muy difíciles de rastrear. Este es un ejemplo de lo que llamaremos “interacción de mecanismos”.
- En este caso, dos diferentes mecanismos de FORTRAN han interactuado (las variables no declaradas y las palabras clave en contexto) para generar problemas más serios.

Convenciones léxicas

- Esta es una clara violación del **Principio de Defensa en Profundidad**, ya que se han roto fácilmente dos líneas distintas de defensa del lenguaje ante la interacción de estos dos mecanismos.

Si un error logra escapar una línea de defensa, entonces debiera ser atrapado en la siguiente línea de defensa.

Registros de Activación

- Cuando un subprograma llama a otro, el estado del invocador debe preservarse antes de entrar al invocado.

- Asimismo, dicho estado deberá restaurarse cuando se concluya la ejecución del procedimiento invocado.

Registros de Activación

- Se denomina "estado del invocador" a toda la información que caracteriza al estado de procesamiento en progreso (es decir, el contenido de todas las variables, el contenido de los registros en uso, y el apuntador a la instrucción que se está ejecutando en ese momento).

Registros de Activación

- Toda esta información se almacena en una región de memoria que pertenece al invocador.

- A dicha región se le denomina "registro de activación" (*activation record*), puesto que contiene toda la información relevante a una **activación** de un subprograma.

Registros de Activación

- Para transmitir al procedimiento invocado un apuntador al registro de activación del invocador, normalmente se almacena dicho apuntador en el registro de activación del invocado.

- A este apuntador del registro de activación de un procedimiento invocado al registro de activación de su invocador, se le llama "liga dinámica" (*dynamic link*).

Registros de Activación

- Consecuentemente, una cadena dinámica es la secuencia de ligas dinámicas que van de cada procedimiento invocado a su invocador.
- Puesto que FORTRAN no permite recursividad, la implementación de la invocación de subprogramas es muy simple.

Registros de Activación

- Las tareas que deben completarse para poder efectuar la invocación de un subprograma son las siguientes:

- 1) Colocar los parámetros en el registro de activación del procedimiento invocado, de acuerdo al mecanismo de paso de parámetros utilizado (FORTRAN sólo permitía paso por referencia).

Registros de Activación

- 2) Almacenar el estado del invocador en su propio registro de activación (incluyendo el punto al cual se pasará el control cuando se regrese del subprograma invocado).
- 3) Colocar un apuntador al registro de activación del invocador en el registro de activación del procedimiento invocado (una liga dinámica).
- 4) Transferir el control al procedimiento invocado.

Registros de Activación

- Los pasos requeridos para regresar del procedimiento invocado al invocador son los siguientes:

1) Continuar la ejecución del invocador en el punto indicado por el registro apuntador de instrucción (IP).

2) Restaurar el estado del invocador (es decir, los valores de sus variables, contenido de sus registros, etc.) a partir de su registro de activación.

Registros de Activación

- A partir de lo anterior puede verse fácilmente que el registro de activación debe contar con suficiente espacio para almacenar la siguiente información:

1) Los parámetros que se pasen al subprograma.

2) La dirección de reinicio (valor del registro IP).

3) La liga dinámica.

4) Áreas temporales para almacenar el contenido de los registros y otra información volátil.

Registros de Activación

- Puesto que FORTRAN no permite subprogramas recursivos, sólo se requiere un registro de activación por cada subprograma.

- Este concepto es muy importante y lo usaremos varias veces más a lo largo del curso.