**Definiciones.**

Sintaxis: Parte de los lenguajes que se encarga de estudiar cómo están escritas las cosas.

Semántica: Estudia el significado de las cosas que escribí, qué interpretación le doy.

Pragmática: Estudia cómo se hace lo que escribí de tal manera.

**Binding.**

**Binding (ligadura)**: Momento preciso en que se conoce un atributo de un elemento de un lenguaje de programación.

*Atributo*: tipo, valor, almacenamiento, alcance.

*Elemento*: variables, constantes, palabras reservadas, funciones, métodos.

Estudiamos el binding de variables.

**Tipo.**

Teniendo la asignación: a=b;

* **Estático**: El tipo de “a” no va a cambiar. Modificación en tiempo de compilación.
* **Dinámico**: Tengo que esperar a la ejecución para que se modifique. Modificación en tiempo de ejecución.

Si para hacer esta asignación se cambia el tipo de a, para que se vuelva del tipo de b, entonces digo que el lenguaje tiene tipos dinámicos. O sea, que si el tipo se copia, se tiene tipos dinámicos. Si el tipo de b se convierte al tipo de a y luego se copia, o sea que se adapta, entonces tienen tipos estáticos.

**Valor.**

Siempre el binding de valor es dinámico (el valor de la variable varía todo el tiempo).

**Alcance.**

Conjunto de instrucciones dentro de un programa que puede usar una variable.

* *Alcance local*: Puedo usar las variables dentro de delimitadores (como llaves).
* *Alcance global*: Es alcanzado por todos.
* **Estático**: Veo el programa y ya sé dónde puedo usar esa variable.
* **Dinámico**: Tengo que esperar a la ejecución para saber dónde puedo usar la variable.

Ejemplo de lenguaje GW Basic:

10 A=51

20 INPUT K

30 IF K>0 GOTO 50

40 C=A+K

50 X=A+C

En Basic no se declaran las variables. Una variable tiene alcance si toma valor. El alcance es dinámico porque una variable sólo se puede usar si tiene valor. Entonces tengo que esperar a la ejecución para saber si “C” tiene alcance o no. Porque según el valor de “K”, se le asigna valor o no. Si K es mayor que cero el programa da error, porque Basic no puede usar las variables si no tienen valor. Entonces sólo se sabe en tiempo de ejecución si la sentencia 50 puede usar la variable C, porque a veces sí la puede usar y a veces no. Basándose en las reglas de alcance que impone ese lenguaje, puedo decir si “A” puede usarse de vuelta o no.

**Almacenamiento.**

Momento preciso en que se liga la ubicación o dirección a una variable en un lenguaje de programación.

* **Estático**: La variable siempre está en memoria.
* **Dinámico**: La variable a veces está en memoria y a veces no.

¿Cuándo una variable tiene memoria? Depende del lenguaje.

* C asocia la dirección de sus variables sólo cuando la función que la tiene es invocada.
* Cobol y Fortran tiene a sus variables siempre en la memoria (almacenamiento fijo).

Si tengo un SO con particiones fijas:



Cobol y Fortran: No sabe cuanto va a usar. No eficiente.



Mayoría de los lenguajes: Carga el ejecutable y después tiene una frontera móvil de datos (depende de las funciones que se invocan).

Es **más óptimo**. Favorece a la recursividad (llamada a sí misma). Con direccionamiento absoluto no podría hacer recursividad porque cada vez que la función se invoque, necesito una nueva copia de las variables.

*Cuadro resumen.*

|  |  |  |
| --- | --- | --- |
|  | **Estático** | **Dinámico** |
| **Tipo** | Mayoría de los lenguajes | J, smalltalk, APL |
| **Valor** | Sólo en constantes con nombre | Mayoría de los lenguajes |
| **Alcance** | Mayoría de los lenguajes | Basic (versiones), APL, Lisp |
| **Almacenamiento** | Sólo Cobol y Fortran | Mayoría de los lenguajes |

**Clasificación de los lenguajes de programación.**

* **Lenguaje estático**: Almacenamiento estático. (Cobol y Fortran). Los lenguajes estáticos son *no recursivos*. La recursividad es una noción de que una unidad puede llamarse a sí mismo. En ese llamado a sí mismo se supone que arranca con todas las variables en su estado inicial, pero además supone que cuando vuelva, recupere los valores anteriores, entonces debe tener varias copias de las variables y no se puede hacer esto si las variables están en un lugar fijo.
* **Lenguaje tipo Algol**: Almacenamiento dinámico, tipo estático y alcance estático. (C, C++, Algol, Pascal, Java, Delphi).
* **Lenguaje dinámico**: Almacenamiento dinámico, tipo o alcance dinámico. (Smalltalk, J, APL, Basic).

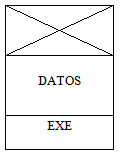
**Modelos de ejecución.**

* **Compilado**: Existe un programa llamado *compilador* que traduce (cambia) un lenguaje en otro lenguaje. Cambia el lenguaje original (no totalmente). No quedan las variables, no quedan nombres. Es más rígido y es mucho más eficiente, más rápido.
* **Interpretado**: Existe un programa *intérprete,* que entiende las instrucciones de un lenguaje y las lleva a cabo. No cambia el lenguaje original (alguito cambia). Es más flexible y en consecuencia, más lento.

Nada es completamente compilable ni completamente interpretable. Ej.: printf de C, que es interpretable.

Los lenguajes dinámicos no se pueden compilar, necesitan un intérprete siempre.

**Lenguaje estático: Usa modelo compilado.**

a = b + c; → lenguaje fuente

10 20 30 → direcciones absolutas

Compila, le da las direcciones y genera código (que es lo que después ejecuta):

MOV R1 20

ADD R1 30

MOV 10 R1

Al compilar, pierdo los nombres, y manejo sólo direcciones de memoria en la ejecución. Hay una parte que no se usa.

**Lenguaje tipo Algol: Usa modelo compilado.**

a = b + c; → lenguaje fuente

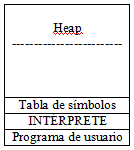
100 200 300 → direcciones relativas

El compilador asocia a cada uno una distancia (Offset), se transforma en una dirección relativa a algo.

La zona de datos es una pila ordenada. El stack crece y decrece en forma de pila.

La ejecución avanza o disminuye ocupando más o menos memoria.

**Lenguaje dinámico: Usa modelo interpretado.**

a = b + c → así se carga en la memoria

Programa de usuario: Es el programa que hizo el programador.

Heap (montículo): Región alejada de la memoria donde están los datos.

En los lenguajes dinámicos, como los tipos cambian continuamente, necesito un lugar dinámico donde poner y sacar (heap). Este está desordenado. Los nombres permanecen en la memoria. Siempre mantengo los nombres (más complicado). Siempre usa heap.

No soporta el modelo compilado porque no sabe cuánto van a ocupar los datos.

**Tabla de símbolos**: Maneja el dinamismo de tipos. Tiene el nombre de variables con su tipo que se van a ir modificando y un puntero al heap.

Los lenguajes compilables también tienen tabla de símbolos, pero la tienen en tiempo de compilación. Luego se destruye.

**Administración de memoria.**

Los programas del programador se dividen en unidades o bloques.

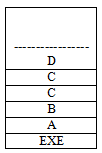
**Unidad o bloque**: Conjunto de instrucciones delimitado por dos delimitadores donde es posible definir o declarar variables locales (donde hay variables locales). Ejemplo: En C, las sentencias encerradas entre llaves conforman una unidad. En Pascal, las sentencias que se encuentran entre begin y end, no conforman una unidad porque no se pueden declarar variables allí (no existen bloques anónimos en Pascal).

Existen dos tipos de unidades:

* **Anónimas**: Se ejecutan cuando el mismo flujo del programa cae adentro.
* **Con nombre**: Se ejecutan sólo si se las invoca. Ej.: funciones en C, procedimientos en Pascal.

Las unidades  *anónimas* no pueden ser recursivas, las unidades *con nombre* sí pueden serlo.

Suponemos que tenemos una serie de bloques que se llaman unos a otros. Luego de compilar, nos queda código en assembler (en nuestro caso, no siempre). Nuestra sentencia de ejecución (que depende de lo que ocurre a medida que se va ejecutando el programa) es:

A → B → C → C → D (sólo se cargan las que se usan).

Primero se carga A. Cuando A llama a B, se carga B en memoria y las variables locales de B. Cuando B llama a C se carga en memoria las variables locales de C. Cuando C llama a C, se carga otra copia de C (del mismo tamaño. Si C tiene una variable x, se tiene una copia de x por cada instancia de C). C llama a D y se carga en memoria D y sus variables locales. O sea que a medida que las variables empiezan a ejecutar, se crean espacios para las variables.

C no puede terminar antes de que termine D (y así con los demás). Este conjunto de datos tiene un ordenamiento LIFO. Es una pila. El límite (línea punteada) va variando subiendo y bajando de acuerdo a qué unidad se llame o no se llame.

**Registro de activación.**

El registro de activación es el conjunto de datos necesarios para comenzar la ejecución de una unidad. Los registros de activación son una porción de memoria reservada para los datos que manipula una unidad o bloque en ejecución. Cuando en el programa se llama a una función, inmediatamente se reserva en memoria el espacio donde contendrá las variables locales de la función, los parámetros de entrada y de salida y todos aquellos elementos que se necesiten para su ejecución. Cuando esta función concluye su ejecución, se procede a liberar la memoria que ocupa el registro de activación y se retorna al bloque llamador. Cada unidad en ejecución tiene un registro de activación. (Aclaración: El registro de activación tiene sólo datos. El código está en el ejecutable, en la memoria también pero en otra zona).

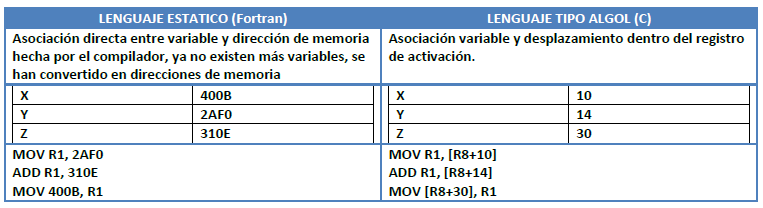
Tengo la secuencia: A → Q → Z → B

El compilador guarda previamente una variable estática (fija) en una dirección que sirve para manejar la ejecución.

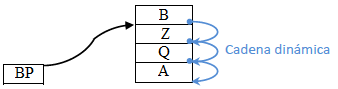
Se carga el registro de activación de A y el BP apunta a la base de A. A llama a Q y se carga el registro de activación de Q y el BP apunta a la base de Q. Q llama a Z y se carga el registro de activación de Z y el BP apunta a la base de Z. Z llama a B y se carga el registro de activación de B y el BP apunta a la base de B.

En un lenguaje tipo Algol, las variables no tienen dirección que cambia. Tienen almacenamiento dinámico, pero dentro del registro de activación tienen un lugar fijo. Es decir, las variables tienen dirección interna fija, no dirección real fija. La dirección real se compone por el comienzo del registro de activación + lugar interno.

Diferencia entre un lenguaje estático y un lenguaje tipo Algol para: **X:=Y+Z;**



En lenguajes tipo Algol, las direcciones son relativas a la base del registro de activación. En la ejecución cambia la base (BP o R8). Por eso puedo tener varias copias de lo mismo y nunca se va a pisar. Si tengo dos copias de lo mismo, lo que fueron X, Y, Z van a estar a la misma distancia de la base del registro de activación. Pero cada copia tiene un registro de activación distinto.



Para realizar la cadena dinámica, el compilador primero tuvo que:

1. Resguardar la posición del BP (base de Z) en un registro.
2. Cuando Z llama a B, se le agrega el tamaño del llamador (tamaño de Z).
3. Mover el BP al nuevo lugar (el nuevo lugar es la base del llamado, en este caso, la base de B). El llamador sabe a quién llama pero el llamado no sabe quién lo llamó. Cada registro de activación, para su administración, tiene una dirección de retorno (para que el llamado sepa donde tiene que volver).
4. Resguardar la dirección de la base del llamador (en el registro de activación de B se guarda la dirección de la base de Z). Esta dirección se guarda en una variable que el compilador reserva y que está siempre a la misma distancia de la base.  
   Dirección de retorno: **CD = #1** (esta dirección no se puede usar para otra cosa).

Ahora, la misma secuencia pero en assembler:

MOV R7 BP **(1)**

ADD BP TAMAÑO(Z) **(2)**

MOV BP+1 R7 **(3)**

Siendo R7 el registro donde guardo el BP. Esta secuencia es la que genera el compilador cuando el programador, estando en Z dice “**call B**”. Cuando B termina, se hace el **return**:

MOV BP BP+1

Aquí se libera B (las variables de B ya no están).

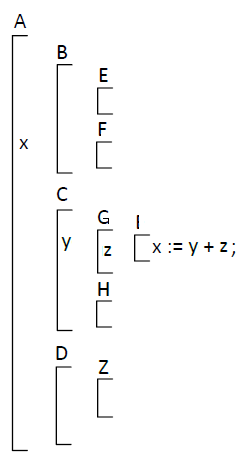
**Cadena dinámica**: Sucesión de punteros al registro llamador. Si no estuviera, no se podría retornar nunca.

Esta forma de administración de memoria funciona si las variables son locales a cada registro de activación (si está en el bloque en que se está ejecutando).

Si una variable está definida en otro bloque, hay que administrarlo de otra forma.

Las unidades de un programa pueden estar anidadas y puedo moverme dentro de ellas con variables mientras éstas tengan alcance. Este asunto de poder usar variables de las unidades que la contienen implica que debe estar reflejado de alguna manera en la pila de registros de activación. Entonces en la pila de registros de activación hay otro puntero que describe la estructura estática o de anidamiento del programa. Eso es lo que permite manejar las variables globales.

**Regla de alcance**: *Buscá las variables en el ámbito local, si ahí no las encontrás, andá al ámbito más externo hasta el último.* (Alcance estático. Esta regla se aplica en tiempo de compilación ya que las variables no se buscan en tiempo de ejecución).



En el ejemplo vemos que x, y, z no están declaradas en I. x está declarada en A, y en C y z en G.

El compilador, además de darle un offset a cada variable, le da un par ordenado, donde la primera componente es el **offset** y la segunda indica el **nivel de anidamiento** (en variables locales, el nivel de anidamiento es cero):

Variable(offset, nivel de anidamiento)

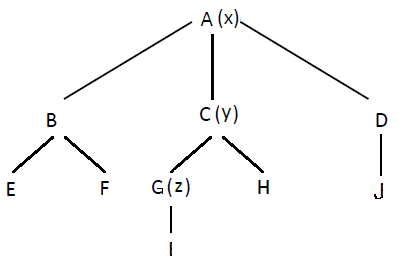
a(#20,3)

b(#17,2)

x(#12, 1)

Durante la compilación, el compilador además crea un árbol con la estructura del programa, el **árbol de anidamiento**. Sólo existe durante la compilación y luego se destruye (no llega a la ejecución).

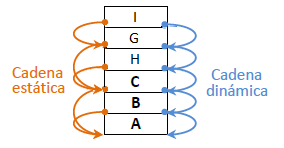
Cada vez que se compila una instrucción, se busca en la tabla de símbolos (que existe en tiempo de compilación) y se deduce en qué lugar está dentro del registro de activación.

Entonces tener en cuenta:

* Regla de alcance.
* Árbol de anidamiento.
* Ubicación de la instrucción.

Suponemos que tenemos la siguiente secuencia de llamados:

A → B → C → H → G → I.



Entonces cada registro de activación tiene dos punteros, uno al que lo llamó (*cadena dinámica*) y el otro a su padre en el árbol de anidamiento (*cadena estática*).

Cadena estática: **CE = #2**.

MOV R7 BP //*Reservo el* BP

MOV BP BP#2 //*Colocar la instrucción tantas veces como niveles de*

MOV BP BP#2 //*anidamiento haya (hay 2 niveles), busco* y

MOV R1 BP#17 //*Obtengo* y

MOV BP R7 //*Restauro el* BP *(vuelvo a I)*

MOV BP BP#2 //*1 nivel de anidamiento, busco* z

ADD R1 BP#12 //*Sumé* z

MOV BP R7 //*Restauro el* BP *(vuelvo a I)*

MOV BP BP#2 //*Hay 3 niveles de anidamiento, voy a A para buscar* z

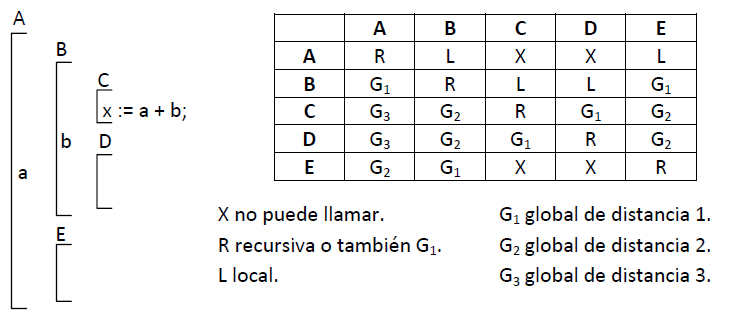
MOV BP BP#2

MOV BP BP#2

MOV BP#20 R1 //*Guardo el resultado en* x

MOV BP R7 //*Restauro el* BP *(vuelvo a I)*

El alcance de una unidad es global a sí misma y local a quien la contiene. Esto conlleva a la pregunta de quien puede llamar a quien, para lo cual se realiza la siguiente matriz, donde en vertical se pone la unidad que llama y en horizontal, la unidad llamada.

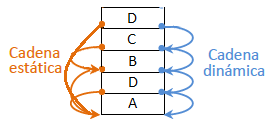
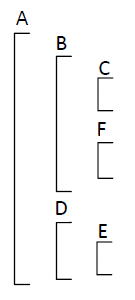


Esta matriz no es armada por el compilador, el compilador sólo hace el árbol de anidamiento. Esta matriz permite ver qué cosas están ocultas y qué cosas están a la vista en el conjunto de unidades anidadas.

*Cuando la llamada es local, el puntero de la cadena estática coincide con el puntero de la cadena dinámica (el padre llama al hijo).*

*La cadena estática se construye usando la cadena estática*. Pasos para crear la cadena estática:

1. Crear el puntero de la cadena dinámica.
2. Copiar el puntero de la cadena dinámica en el puntero de la cadena estática. O sea, suponemos que la llamada es local.
3. Mirar el árbol de anidamiento y calcular la distancia n en el árbol de anidamiento.
4. Bajar por la cadena estática construida tantos arcos como indica el nivel de anidamiento del llamado.



MOV R7 BP

MOV BP BP#1 **(2)**

MOV BP BP#2 **(4)**

MOV BP BP#2 **(4)**

MOV R7#2 BP //*enlaza la CE de D*

MOV BP R7 //*restaura*

Si en el ejemplo anterior, D llamara a A, se generaría la CD, pero la CE no. Después si A llamara a B, ahí sí se crearía la CE, pero con respecto a la última A llamada, no a la del principio.

Las cadenas estática y dinámica se refieren sólo a variables, no a las partes ejecutables de las unidades. Las partes ejecutables están en lugares permanentemente fijos.

**Clasificación de variables.**

La siguiente clasificación es según la ubicación o tamaño de las variables.

* **Estáticas**: Poseen ubicación fija y tamaño fijo. La variable tiene dirección absoluta y no cambia de tipo, por eso mantiene su tamaño. Pertencen a lenguajes estáticos como Fortrand o Cobol. Puede haber también variables estáticas que se fijen en compilación, por ejemplo, a través de la sentencia *static* de C.
* **Semi-estáticas**: Poseen ubicación variable y tamaño fijo. La dirección es relativa al bloque donde se declaran: *base + desplazamiento*. Estas variables están dentro del registro de avivación pero tienen siempre el mismo tamaño, por lo que su offset es fijo. Ejemplos: int, reales, float, vectoress cuyos límites se conozcan en tiempo de compilación.
  + Fórmula de acceso para un **vector**:

**V(Li,Ln) → V(j) = dir(V(Li)) + [j-Li] \* Tamaño del componente**

* + En el caso de una matriz, que se almacena en forma lineal, tenemos dos formas de organizarla (por filas y columnas), por lo tanto, dos formas de acceder a ella.
    - Matriz organizada por filas.

**V(Fi…Fn, Ci…Cn) → Z(i,j) =**

**dir(Z(Fi,Ci)) + [(i-Fi)(Cn-Ci+1)+(j-Ci)]\*Tam. del componente**

* + - Matriz organizada por columnas.

**V(Fi…Fn, Ci…Cn) → Z(i,j) =**

**dir(Z(Fi,Ci)) + [(j-Ci)(Fn-Fi+1)+(i-Fi)]\*Tam. del componente**

Para acceder a un componente necesito:

* + - Por filas: Fi, Ci, Cn.
    - Por columnas: Ci, Fi, Fn.

En el caso de que el lenguaje no verifique los límites, necesita 3 límites (según la forma de organización); si los verifica, necesita los 4 límites (Fi, Fn, Ci, Cn).

* **Semi-dinámicas**: Poseen una ubicación variable y un tamaño variable, el cual queda fijo al comenzar el bloque. Una vez que se ubican en el registro de activación, se quedan quietas, no cambian de tamaño (no pueden, porque tienen otro registro atrás). Las variables semi-dinámicas son arreglos con límite variable. Sus límites son una incógnita en tiempo de compilación (ese límite desconocido es una variable global a ese procedimiento, es decir, conocido por un proceso padre).

Para acceder a una variable semi-dinámica, se realiza a través de un **descriptor**, que se encuentra en una porción dentro del registro de activación. El descriptor tiene los límites (3 o 4 según si se verifican o no) y un puntero a los datos donde comienza el vector. El descriptor es una variable *semi-estática*.

Dentro del registro de activación, se ponen primero todas las variables semi-estáticas, y luego para las semi-dinámicas lo que se hace es poner los descriptores de las variables. Se van poniendo las distintas variables semi-dinámicas y se van cargando los punteros. Entonces no importa que las variables semi-dinámicas cambien de tamaño porque con base + desplazamiento llego al descriptor y el descriptor me dice dónde está el dato. Las variables semi-dinámicas son más costosas de utilizar, pero más flexibles.

Mecanismo de acceso:

* + - Base + desplazamiento.
    - Seguir puntero.
    - Acceder a los datos.
* **Dinámicas**: Poseen una ubicación variable y un tamaño variable. Pueden cambiar de tamaño en cualquier instrucción. Ejemplos:
  + p = malloc(100) //C++ **Variable dinámica anónima**

Los 100 bytes son los variables. Por medio de un puntero (variable semi-estática), accedo a la zona de memoria donde están esos 100 bytes, en el heap. La característica esencial de las variables anónimas es que están fuera de la pila.

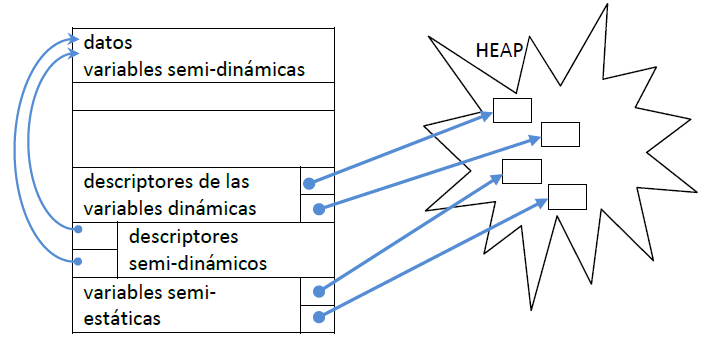
* + z Flex [1:0] int //Algol **Variable dinámica con nombre**

Casi desaparecieron. La forma de declarar arreglos flexibles en Algol es poniendo límites sin sentido.

Se guarda un descriptor (que tiene un puntero interno) con la dirección donde se encuentra esa variable dinámica con nombre (en el heap).

* **Súper-dinámicas**: Poseen ubicación variable, tamaño variable y pertenecen a lenguajes dinámicos, es decir que su tipo puede cambiar. Posee tabla de símbolos (porque el tipo puede cambiar).

Un registro de activación completo sería:



*Cuadro resumen:*

|  |  |  |  |
| --- | --- | --- | --- |
| **Tipo** | **Ubicación** | **Tamaño** | **Ejemplos** |
| **Estática** | Fija | Fijo | Cobol, Fortran, static de C |
| **Semi-estática** | Variable | Fijo | C++, Pascal, int, real, long, vectores y matrices con límites conocidos. |
| **Semi-dinámica** | Variable | Variable (fija al comenzar el bloque) | ADA, arrays con límites incógnitas. |
| **Dinámica** | Variable | Variable | malloc/new |
| **Súper-dinámica** | Variable | Variable / Tipo variable | Smalltalk, J |

*Variables en función de los tipos de lenguajes:*

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
|  | **E** | **SE** | **SD** | **D** | **Super D** |
| L. Estáticos | X |  |  |  |  |
| L.T.A. | X | X | X | X |  |
| L. Dinámicos |  |  |  |  | X |

**Problemas con punteros manejados.**

**Caso 1: Pila a heap.**

**Variables dinámicas anónimas**: Área que no tiene nombre pero es una variable, Sólo puede ser apuntada por una semi-estática que adentro tiene una dirección. Este puntero se encuentra dentro del registro de activación.

Entonces puedo tener un puntero p que apunta a un tamaño de 100 bytes. Es anónimo y está en el heap, pero lo más importante es que el programador puede modificar el puntero. Se decimos p=q, copiamos el puntero y ese p va a apuntar a otro lado y ya no ve los 100 bytes que tenía antes. Esta situación se denomina **garbage**. La creación de garbage es tener algo en el heap inaccesible porque se ha perdido el puntero que me permitía acceder (sólo en el heap). Pero si hago free(p), entonces hago desaparecer el bloque de 100 bytes y el puntero se queda apuntando a algo que ya no existe más. En este caso se suele hablar de **dangling referente** o **punteros colgados**.

Estas situaciones sólo se dan cuando tengo variables dinámicas anónimas, porque tengo un puntero que administra el programador y un bloque de datos que administra el programador. Esto no puede pasar con variables dinámicas con tipos dinámicos, porque siempre la tabla de símbolos los está apuntando. En los lenguajes tipo Algol pueden aparecer punteros que apunten a algo que no existe y puede haber bloques de memoria inaccesibles.

Ejemplos:

*Puntero colgado.*

int \*p, \*q;

p=malloc(50);

q=malloc(50);

p=q;

free p;

*Garbage.*

int \*p, \*q;

p=malloc(100);

q=malloc(100);

p=q;

**Caso 2: Pila a pila.**

Ocurren sólo punteros colgados. En Algol, si el alcance del lado izquierdo es:

* + menor al del lado derecho → OK.
  + mayor al del lado derecho → ERROR, no lo deja pasar.

Regla de alcance: *En una asignación de referencias, el alcance del lado izquierdo debe ser menor o igual al alcance del lado derecho.*

**Administración del heap.**

Puede pasar que por mal uso de la memoria, por haber fabricado muchos huecos, no me alcance la memoria. La memoria dejó de ser compacta para pasar a estar muy fragmentada. Entonces esto implica cierta necesidad de administrar el heap.

* **Lenguajes estáticos**: No existe el heap, entonces no hay problemas.
* **Lenguajes dinámicos**: Sólo hay heap, no hay pila. Entonces el lenguaje administra correctamente el heap (asignación y liberación de bloques), pero pueden surgir huecos, lo cual se soluciona con la *compactación* (mal llamado garbage collection). Los lenguajes dinámicos sólo tienen fragmentación porque todos los bloques usados están apuntados por la tabla de símbolos. Lo que se ve en la tabla de símbolos es lo único que sirve.
* **Lenguajes tipo Algol**: Hay pila y hay heap, hay punteros generados por el usuario, entonces hay garbage y punteros colgados. *Garbage collector* es sinónimo de recolección de basura + compactación.

Para la recolección de basura, existen dos algoritmos:

* **Escaneado y marcado**: El compilador mantiene dos listas: lista de usados y lista de libres. Existen 3 pasos en este algoritmo:
* Escaneado y marcado: El compilador recorre la lista de usados y marca todas las variables dinámicas usadas. (Cuando se fabricó el bloque, se hubo que haber reservado un espacio para la marca).
* Busca en la pila todos los punteros que apuntan a algún lugar del heap y les saca la marca (a los que tienen referencia). Seguir los punteros del heap, borrando la marca.
* Recorre nuevamente la lista de usados. Si alguno quedó marcado, quiere decir que es basura, y lo pasa a la lista de espacio libre.

Algunos lenguajes corren este algoritmo:

* + cada vez que hay asignación dinámica,
  + a pedido del usuario,
  + en modo pánico (cuando no hay nada libre).

Los bloques de memoria tienen una zona reservada para la administración, donde hay un lugar para almacenar la marca y el tamaño de del bloque.

* **Conteo de referencias** (o garbage collections): Cuando un puntero apunta a un bloque, en el momento de su creación, se fabrica un contador (ubicado en la variable dinámica anónima) que indica que un puntero apunta allí. Cada vez que se solicita un bloque de memoria, aumenta el contador de referencias. Cada vez que hay una desvinculación de referencias o liberación, se resta el contador.

Una vez que el contador llega a cero, automáticamente ese bloque se incorpora a la lista de espacios libres. De esta forma, no permite que haya basura ni punteros colgados (en realidad no recolecta nada). Este algoritmo no funciona con referencias circulares.

En el espacio de administración se guarda el tamaño del bloque y el contador de referencias.

**Compactación**: Una vez que se aplica el algoritmo de recolección de basura, pueden quedar agujeros libres en la memoria, por lo que se suele aplicar la compactación, que consiste en juntar los bloques de memoria usados al principio de la memoria (posición cero). Para llevarla a cabo se siguen 3 pasos:

* Calcular las nuevas direcciones, de derecha a izquierda. Las nuevas direcciones calculadas se almacenan en la zona de administración.
* Modificar los punteros con las nuevas direcciones.
* Trasladar las variables dinámicas a las nuevas direcciones.

**Implementación de objetos.**

La relación entre la pila y el heap es a través de punteros.

Los objetos se tratan desde el punto de vista de la administración de memoria como variables, es decir, en la pila puede haber objetos y en el heap puede haber objetos que obligatoriamente van a estar referenciados por punteros. El acceso y almacenamiento de objetos y variables es indistinguible.

Lo mismo pasa en Algol y en los lenguajes dinámicos, pero una variable no es lo mismo que un objeto. Los objetos se diferencian de las variables en su estructura interna y en el uso de esa estructura interna (no en el mecanismo de acceso). Una variable de cualquier naturaleza tiene en su interior datos. Un objeto tiene datos más punteros al objeto.

Si una función normal llama a otra, la llamada es *estática*, es decir, el llamador sabe donde está el llamado y cuando termina, ejecuta la próxima instrucción del llamador.

Ahora, si tenemos un objeto en el heap accedido por un puntero en la pila, y una función le quiere mandar un mensaje al objeto, se hace a través del puntero. Entonces, cuando quiero mandar un mensaje uso al objeto para llegar al método. Acá la vinculación no es estática, sino que es *dinámica* más o menos dependiendo del lenguaje.

En **Algol**, todos los objetos de una clase comparten los mismos métodos, aunque tengan datos distintos. Por eso, en cada objeto ponemos un puntero que apunta a una tabla de despacho. Tenemos entonces una tabla de despacho por cada clase, que es la parte de la zona ejecutable.

**Lenguajes dinámicos.** Los que tienen sólo alcances dinámicos con tipos estáticos no se diferencian del lenguaje tipo Algol. Los que tienen tipos dinámicos sí se diferencian: El objeto se mueve (migra de la clase), se le cambian los métodos, entonces pierde la tabla de despacho.

El sistema operativo provee memoria, la forma de la caja, y el lenguaje adentro hace lo que quiere. Estas cajas sólo están en un sistema operativo con particiones fijas. No todos los SO se llevan bien con los lenguajes.

**Interacciones con el SO.**

**Caso básico: Una partición.**

* **Lenguaje estático**: Usan una parte de la memoria (para programa y datos) y lo demás lo dejan libre para siempre.
* **Lenguaje tipo Algol**: El compilador sólo calcula el espacio para el ejecutable. La pila crece y nos podemos quedar sin memoria cuando la pila y el heap se chocan, entonces termina la ejecución.
* **Lenguaje dinámico**:También ocurre que en un punto del programa no pueda seguir más por falta de memoria.

**Varias particiones.**

La diferencia es que tengo algunos bloques de memoria grandes y otros más chicos, por lo que hay que decidir dónde se ubica el programa.

* **Lenguaje estático**: Se conoce el tamaño del programa, entonces se le da una partición en la que quepa. Se elije la partición más chica donde entre.
* **Lenguaje tipo Algol**: Son engañosos para el SO porque no le dicen cuánta memoria necesitan. El mayor consumo de memoria está en la zona de datos más que en el ejecutable, pero lo que el SO ve es sólo el ejecutable. El lenguaje sabe las variables que declaró el usuario pero no conoce los niveles de recursividad, ni la recursividad mutua (el lenguaje tampoco sabe el tamaño). Entonces el lenguaje pone un tamaño por defecto, dándole al programador la oportunidad de decir al compilador la cantidad de memoria que necesita el programa. El programador sabe el tamaño si es que analizó el programa (sino, no).

SO pregunta al lenguaje **→** lenguaje pregunta al programador. Si este no sabe, tamaño por defecto.

El programador le dice el tamaño al lenguaje a través de: cláusulas de compilación, variables de ambiente, pragma.

El lenguaje le dice al SO dentro de los 128 bytes iniciales de información que hay dentro del ejecutable.

**Un segmento.**

La diferencia entre particiones fijas y segmentos es que con segmentos puedo elegir dinámicamente el tamaño de la partición. El SO no sabe de qué tamaño crear el segmento. Entonces el programa pone un tamaño por default. La ventaja es que se crea un segmento del tamaño del programa.

**Varios segmentos.**

Cuando el SO puede manejar varios segmentos por programa, especialmente la línea Intel se caracteriza por tener 4 segmentos de tamaño fijo (no mismo tamaño):

* Code segment: Parte ejecutable (tamaño fijo).
* Data segment: Parte de los datos, variables estáticas.
* Stack segment: Pila.
* Extra segment: Heap.

La pila y el heap se construyen como se ha visto, salvo que cada uno tendrá su espacio de direccionamiento independiente.

**Memoria paginada o memoria virtual.**

La memoria *virtual* (inmensa) le importa al lenguaje. La memoria *real* (más chica) no le importa al lenguaje sino que de ella se ocupa el SO.

El programa ve un “rectángulo” muy grande. La gran diferencia es que los tamaños relativos son muy despreciables.

La pila y el heap están tan lejos que no se chocan casi nunca, por lo que no hay problemas de garbage collection. La memoria virtual va de la mano con lenguajes de tipo Algol y lenguajes dinámicos (LTA y LD se llevan mal con los que no tienen MV). Los lenguajes estáticos se llevan bien con todos los SO.

**Problemas de concurrencia.**

La concurrencia está vinculada con las hebras de ejecución.

Tenemos:

* *Lenguajes concurrentes*: Pascal concurrente, ADA, Java, C (últimos estándar). La concurrencia se da entre funciones, procedimientos y métodos.
* *SO concurrentes*: Linux, Windows. La concurrencia se da entre programas.

La concurrencia implica la existencia de un administrador de concurrencia (una porción de programa que diga a quién le toca). En lenguajes concurrentes administra el lenguaje y en SO concurrentes, el SO.

|  |  |  |
| --- | --- | --- |
|  | **Lenguaje no concurrente** | **Lenguaje concurrente** |
| **SO no concurrente** | No hay concurrencia. | La concurrencia la administra el lenguaje. El SO no sabe que hay concurrencia. |
| **SO concurrente** | La concurrencia la administra el SO. | **?** |

Cuando el SO y el lenguaje son concurrentes, tengo 2 situaciones:

* El SO no sabe que el lenguaje es concurrente, entonces hay 2 administradores de concurrencia: uno del SO y otro del lenguaje. “Threads ad-hoc”
* El SO coordina con el lenguaje la administración de la concurrencia. “Threads verdaderos”. Hay un solo administrador.

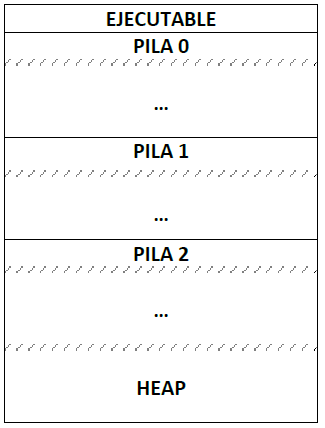
Estas dos situaciones se distinguen por las prioridades:

* 1. *Prioridades asignadas por el SO*. El programa le da distintas prioridades a las distintas funciones (dentro del programa). Si tengo dos programas no puedo intercalar prioridades, debo establecer sub-prioridades. Todas las prioridades de todas las unidades de un programa están por encima de todas las prioridades de todas las unidades de otro programa.
  2. *Acuerdo entre el SO y el lenguaje* por el que se pueden intercalar las prioridades de las distintas funciones de los distintos programas. Como el programa conoce sólo sus prioridades internas y no las de otros programas, esto lo administra el SO (las prioridades y la CPU). El lenguaje le cede al SO la administración de prioridades.

¿Cómo se administra la memoria? Si la memoria del programa es administrada por el lenguaje, el lenguaje no cede al SO la administración de memoria dentro de su espacio de direccionamiento.

En todos los casos en que hay lenguajes concurrentes, la administración de memoria la sigue haciendo el lenguaje. Cuando uno llama a una función, puede crear o no un nuevo hilo.

Si no lo hace, y tengo por ejemplo la secuencia A → B → C → D → E → F, sé que el primero que termina es el último que empieza, entonces tengo la prioridad *pila*.

En cambio, si tengo varios hilos de ejecución concurrentes, no sé cual termina primero, entonces uso una pila por cada hilo de ejecución.

En estos casos, corro el riesgo de que una pila choque con otra pila.

Si el SO es de particiones fijas, no anda. Sólo funciona con MV, ya que las distancias entre pilas son realmente grandes.

A veces falla cuando se carga mucho a un hilo de ejecución y a otro poco, entonces chocan las pilas.

La decisión de cómo colocar las pilas puede ser por default (se van dividiendo en dos los pedazos) o lo dice el programa.

El ámbito o alcance no se asocia a la unidad, sino al hilo de ejecución.

**Pasaje de parámetros.**

La **unidad llamadora** posee *parámetros reales*. Ej.: f(u,v).

La **unidad llamada** posee *parámetros formales*. Ej.: int f(int x, int y).

* **Asociación por posición**: x se asocia con u, y se asocia con v (de izquierda a derecha). Hay dos tipos:
* *Sin faltantes*: La más común.
* *Con faltantes*: f(3) en la unidad llamadora. Entonces x se asocia a 3, y no se asocia. Sé que a y no lo voy a usar.
* Al final.
* En cualquier lugar: Es peligroso. Ejemplo:

Unidad llamada: int g(int x1, int x2, int x3, int x4)

Unidad llamadora: g(4,\_,v,u)

* **Asociación explícita**:

Unidad llamada: g(x1=u1,x4=u3,x6=u6,x7=u7,x10=u2)

Unidad llamadora: g(u1,\_,\_,u3,\_u6,u7,\_,\_,u2)

Desventaja: Debo conocer el nombre del parámetro formal.

Ventaja: No me pierdo.

* **Asociación anónima**: Conocida en C como “cantidad de parámetros variable”.

Unidad llamada: int g(int x, …)

Unidad llamadora: g(3) g(3,4,5) g(u,v)

Ejemplo: printf de C: int printf(char\* formato, …)

**Tipos de semántica en los pasajes de parámetros.**

A → unidad llamadora (en realidad es la que contiene el llamado a B).

B → unidad llamada.

Teniendo la siguiente ejecución:

int B(int x, int y) *Parámetros formales*

en A: call B(u,v) *Parámetros reales*

Existen tres tipos de pasajes de parámetros:

* **Por referencia**: Las únicas variables que existen son las de los reales. Los formales no están en B, son sólo referencias a los parámetros reales.

No existen x e y, sino que son alias para u y v.

C no realiza pasaje por referencia (sólo hace copia de direcciones, no usa un alias).

* **Por nombre**: Cuando se hace el llamado a B, detiene el programa y genera en un lugar llamado **thunk** una copia de B y textualmente hace un reemplazo de los parámetros formales por los parámetros reales. Una vez que hace la copia, recién ahí ejecuta. Es como una macro.
* **Por copia**: Existen tres tipos:
  + **Valor (ida, in)**: El parámetro real se va a copiar en el formal. Cuando entro en el código de B, el copilador hace una copia de los parámetros reales en los formales siguiendo las posiciones de izquierda a derecha.
  + **Resultado (salida, out)**: Los parámetros formales se copian en los reales al final, justo antes del return.
  + **Valor-resultado (entrada-salida, in-out)**: Se copian los parámetros reales en los formales al principio. Antes del return se copian los parámetros formales en los reales.

En ADA, existen las 3 semánticas por copia (además tiene referencia), y se especifica en la sintaxis: int B(IN x, OUT y, INOUT z).

Java y C sólo tienen por copia valor.

**Compatibilidad y conversiones.**

* **Compatibilidad por nombre**: Se puede operar con dos tipos que llevan el mismo nombre de tipo. Ej.: C++, ADA.
* **Compatibilidad por estructura**: Se puede operar con dos tipos que tienen la misma estructura (miden lo mismo en cuanto a cantidad de bytes). Ej.: C.

Ejemplo en ADA:

type reales is new float

A: reales

B: float

A+B //Da error de compatibilidad en ADA, porque son de

distinto tipo aunque hereden la misma estructura.

Ejemplo en C:

typedef float reales

reales A

float B

A+B //OK porque tienen la misma estructura (A y B son ambos

de 32 bits)

* **Conversión explícita**: Hay que escribir la conversión explícitamente (y se ve).
* **Conversión implícita**: No se escribe y no se ve. Hay muchas y es difícil de encontrarlas. El ejemplo anterior en ADA se puede solucionar haciendo:

float (A) + B o A + reales (B)

Si eso lo quisiera asignar a un integer (Z): Z = integer(A + reales (B))

**Conversiones implícitas** (proporciones numéricas).

**a = b \* c + d \* e**

float long float int long

↓ ↓ ↓

double double long

|-------------------------| |----------------------|

double long

|-------------------------------------------------------|

double

short int → int → long → double

float → double

Lenguaje C (tipo Algol) tiene binding estático, con lo cual manda el lado izquierdo y pierde precisión.

**Coprocesador matemático**: Procesador que hace operaciones matemáticas básicas y otras más, además realiza operaciones condicionales. Tiene una UC y una pila LIFO de 8 registros de 80 bits cada uno. Esta pila permite subir elementos y hacer cálculos allí mismo. Cada vez que suba algo, echa hacia abajo a los anteriores. Cualquier operación que haga, lo hace en el segmento ST(1). Las instrucciones las diferencio de las del procesador porque todas llevan una F delante.

**Conversiones implícitas en Algol.**

Conversión implícita = coerción (el programador no la ve).

* **Widening** (ensanchamiento): Conversión en la que el tipo del lado derecho es menor en cuanto a estructura a la del lado izquierdo, entonces ensancha su tamaño. Ej.:

real a

integer b

a:=b

* **Uniting**: Se refiere a las variables de tipo *union* (se manifiesta como una estructura con el tamaño tan grande como el tamaño más grande de los tipos que la involucran). Ej.:

mode b = union (int,real) //reserva 32 bits + espacio para el discriminante

b:=4; //es un entero, se ocupan 16 bits y los otros 16 se desperdician

b:=3.5; //real, se ocupan los 32 bits

El *discriminante* indica el tipo de dato que guardé.

Siempre que tengo del lado derecho una variable del tipo union, debo usar una cláusula de conformidad (hacer un case para preguntar por el discriminante).

mode b = union(int,real)

real x

int y

case b in:

when i: y:=b //i indica que es un int

when r: x:=b //r indica que es un real

* **Deproceduring**: Se crea una variable de tipo procedimiento, a la que se puede asignar directamente un procedimiento. El resultado del procedimiento se guarda en la variable. La variable en realidad es un puntero cuya dirección es la dirección del procedimiento (es como un puntero a una función).

proc b //variable de tipo proc

b:=random //random es un procedimiento

La coerción es ejecutar el procedimiento. No hay registro de activación de random, se va directamente al código.

* **Voiding**: Tirar el resultado de un procedimiento normal, porque no me interesa usarlo. Una variable tipo void es una variable que no es capaz de almacenar nada y no va a tener ningún valor.

void x

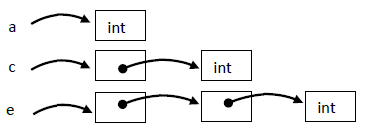
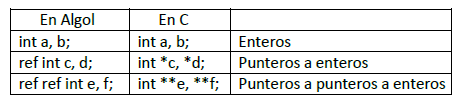
proc xx( ):int //devuelve un int

x=xx( ) //hago una conversión de int a void

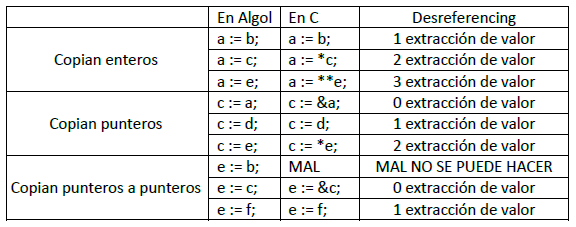
* **Desreferencing**: En una asignación, una variable es una celda de memoria (para Algol). Lo que dice Algol es que el lado izquierdo siempre debe actuar como una referencia a una celda y el lado derecho debe ser un valor almacenable en la celda referenciada por a.

a:=b

b es una celda a la que tengo que extraer un valor y meterlo en a. Convierto la dirección en el valor que necesito.



Vamos a poner todas las posibles asignaciones entre las variables de un tipo y otra de otro tipo:



Cuando hago a:=b, **b** es la dirección de un entero, pero **a** quiere el entero, no la dirección entonces se dice que se hace un desreferencing, que se lee como una extracción de valor. Entonces la conversión implícita desreferencing implica extracción de valor. Seguir el puntero y extraer el valor.

C siempre tiene un desreferencing implícito y el resto explícito y tengo que indicar todos los explícitos. & es un supresor de desreferencing, es necesario por el hecho de que C siempre hace uno implícito y hay casos en los que no tendría que hacer ninguno.