5. Representación de las Informaciones en Memoria

- Aspectos relacionados con los Lenguajes de Programación

5.1. Gestión de la Tabla de Símbolos (TDS)

- Estructura y operaciones con la TDS
- TDS para un LP con estructura de bloques

5.2. Gestión Estática de Memoria

- Introducción: noción de segmento
- Gestión estática de memoria en los segmentos

5.3. Gestión Dinámica de Memoria

- Introducción: necesidad de una gestión dinámica de memoria
- Gestión dinámica para los segmentos: basado en `pila`
- Introducción a la gestion de memoria para Lenguajes Orientados a Objetos
- Gestión dinámica para los objetos de talla desconocida: basado en `montículo`

- Declaraciones antes del uso

  Por ejemplo: JAVA, C++ y PASCAL (si); MODULA2 y PYTHON (no)

- Lenguajes con Estructura de Bloques (LEB)

  Un *bloque* es cualquier construcción que pueda contener declaraciones

  $\Rightarrow$ regla de anidación más próxima

- Recursividad de funciones
- Anidamiento de funciones

Lenguajes	LEB	Recursividad	Anidamiento
FORTRAN	no	no	no
PASCAL	si	si	si
C	si	si	no

- Ámbitos de las variables en LEB

La TDS permite relacionar los nombres (objetos) con sus atributos

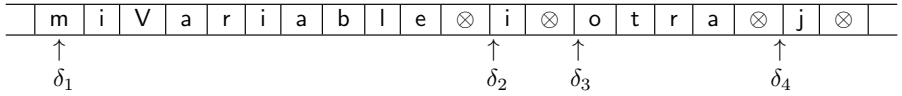
1) **TDS**

Nombre	Atributos
`miVariable`	...
`i`	...
`otra`	...
`j`	...

2) **TDS**

Nombre	Atributos
δ_1	...
δ_2	...
δ_3	...
δ_4	...

Tabla de Nombres



- Estructura de una TDS: Ejemplo *MenosC*
  - **TDSimbolos** (Una entrada por cada objeto definido por el usuario)
    - Nombre del objeto (lexema)
    - Categoría del objeto: *variable*, *parámetro*, *función*, ...
    - Tipo del objeto: *tentero*, *tarray*, *trecord*, *tvacio*, *terror*, ...
    - Desplazamiento relativo en el segmento de memoria correspondiente
    - Ámbito de las variables: *global* o *local*
    - otros campos de referencia de usos múltiples
  - **TDArray** (Una entrada por cada array definido)
    - Índices del array, número de elementos
    - Tipo de los elementos
  - **TDRecord** (Una entrada por cada campo de los registros)
    - Nombre del campo (lexema)
    - Tipo del campo
    - Desplazamiento relativo del campo
  - **TDArgumento** (Una entrada por cada dominio definido en los bloques)

Operaciones con la TDS

- Implementación de una TDS
  - array de registros;
  - listas (doblemente) enlazadas ordenadas;
  - árboles equilibrados ordenados;
  - tablas de dispersión (*hash*).
- Operaciones sobre una TDS

  insertar la información de un objeto (comprobando que no existe otro con el mismo nombre)

  buscar la información asociada a un objeto

  modificar la información contenida para un objeto

  En un LEB se deben considerar también:

  cargar la información asociada con un nuevo bloque

  descargar la información de un bloque

Ejemplo de LEB: C (1/4)

```
main()
{
.    int a = 0;
.    int b = 0;
.    {
.    .    int b = 1;
.    .    {
.    .    .  int a = 2;
.    .   B2  printf(``%d %d\n'', a, b);
.    .    .
B0   .    }
.   B1    {
.    .    .  int b = 3;
.    .   B3  printf(``%d %d\n'', a, b);
.    .    .
.    .    }
.    .    printf(``%d %d\n'', a, b);
.    }
.    printf(``%d %d\n'', a, b);
}
```


Ejemplo de LEB: PASCAL (2/4)

```
program B0;
var  a, b, c: ...
    procedure B1 (x: ...);
    var b: ...
        procedure B2 (e: ...);
        begin
          ...
          b := a + e;           <==== Punto-1
          ...
        end;
    begin
      ...
    end;
    procedure B3 (e: ...);
    begin
      ...
      b := a + e;               <==== Punto-2
      ...
    end;
begin
  ...
end.
```


Ejemplo de de LEB: C (3/4)

TDB

B0 → 0	0	1
B1 → 1	2	2
B2 → 2	3	3

TDS

0	a	t_a	δ_a	...
1	b	t_b	δ_b	...
2	b	t_b	δ_b	...
3	a	t_a	δ_a	...

El printf del B2 dará: $a = 2$ y $b = 1$

TDB

B0 → 0	0	1
B1 → 1	2	2
B3 → 2	3	3

TDS

0	a	t_a	δ_a	...
1	b	t_b	δ_b	...
2	b	t_b	δ_b	...
3	b	t_b	δ_b	...

El printf del B3 dará: $a = 0$ y $b = 3$

TDB

B0 → 0	0	1
B1 → 1	2	2

TDS

0	a	t_a	δ_a	...
1	b	t_b	δ_b	...
2	b	t_b	δ_b	...

El printf del B1 dará: $a = 0$ y $b = 1$

TDB

B0 → 0	0	1

TDS

0	a	t_a	δ_a	...
1	b	t_b	δ_b	...

El printf del B0 dará: $a = 0$ y $b = 0$

Ejemplo de de LEB: PASCAL (3/4)

Punto-1 →

TDB

B0 → 0	0	3
B1 → 1	4	6
B2 → 2	7	7

TDS

0	a	t_a	δ_a	...
1	b	t_b	δ_b	...
2	c	t_c	δ_c	...
3	B1	t_{B1}	dir_{B1}	...
4	x	t_x	δ_x	...
5	b	t_b	δ_b	...
6	B2	t_{B2}	dir_{B2}	...
7	e	t_e	δ_e	...

Punto-2 →

TDB

B0 → 0	0	4
B3 → 1	5	5

TDS

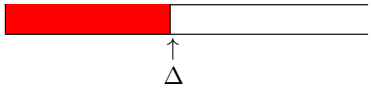
0	a	t_a	δ_a	...
1	b	t_b	δ_b	...
2	c	t_c	δ_c	...
3	B1	t_{B1}	dir_{B1}	...
4	B3	t_{B3}	dir_{B3}	...
5	e	t_e	δ_e	...

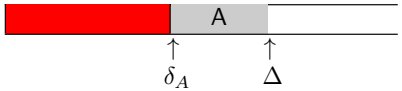
Gestión de memoria

Gestión Estática de Memoria

(Talla de los objetos conocida en tiempo de compilación)

⇒ Asignación estática de memoria de los objetos en segmentos de memoria





Gestión Dinámica de Memoria

⇒ Gestión de memoria para los segmentos

⇒ Gestion de memoria para los objetos de talla desconocida

Gestión estática de memoria

Objetos simples

P ⇒ LD	$n = 0$; $\Delta = 0$;
LD ⇒ LD D	
⇒ D	
D ⇒ DV ;	InsertarTds(DV.nom,"variable-global",DV.t,n,Δ); Δ = Δ+DV.talla;
DV ⇒ T id	DV.nom=id.nom; DV.t=T.t; DV.talla=T.talla;
⇒ T * id	DV.nom=id.nom; DV.t=tpuntero(DV.t); DV.talla=Talla-Entero;
T ⇒ char	T.t=tcarácter; T.talla=Talla-Carácter;
T ⇒ int	T.t=tentero; T.talla=Talla-Entero;
⇒ float	T.t=treal; T.talla=Talla-Real;
⇒ bool	T.t=tlógico; T.talla=Talla-Lógico;

Δ = primera posición libre en el *segmento de datos*. n = nivel del bloque actual.

Gestión estática de memoria

Objetos estructurados: *array*

DV ⇒ T id [cte]	si **not** (cte.t=tentero **and** cte.num>0) MenError(.) DV.nom=id.nom; DV.t=tarray(cte.num, T.t); DV.talla=cte.num ∗ T.talla;

Objetos estructurados: *registro*

T ⇒ struct { LC }	T.t=tregistro(LC.t); T.talla=LC.talla;
LC ⇒ DV ;	LC.t=(DV.nom, DV.t, 0); LC.talla=DV.talla;
⇒ LC DV ;	LC.t=LC'.t⊗(DV.nom, DV.t, LC'.talla); LC.talla=LC'.talla+DV.talla;

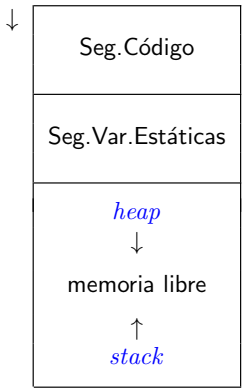
Funciones y parámetros

D	⇒	n++; D.aux= Δ; Δ =0;
	T id (PF) { DL LI }	InsertarTds(id.nom, "función", tfunción(PF.t, T.t), n-1, ◁); n−−; Δ =D.aux;
DL	⇒ ϵ	
	⇒ DL DV ;	InsertarTds(DV.nom, "variable_local", DV.t, n, Δ); Δ = Δ+DV.talla;
PF	⇒	LF.h = TallaSegEnlaces;
	LF	PF.t = LF.t;
	ϵ	PF.t=tvacio;
LF	⇒ DV	insertarTds(DV.nom, "parámetro", DV.t, n, -(LF.h + DV.talla)); LF.t=DV.t;
	⇒ DV , LF	LF'.h = LF.h + DV.talla; InsertarTds(DV.nom, "parámetro", DV.t, n, -LF'.h); LF.t=LF'.t⊗DV.t;

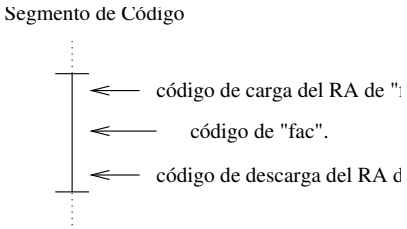
Ambientes de Ejecución

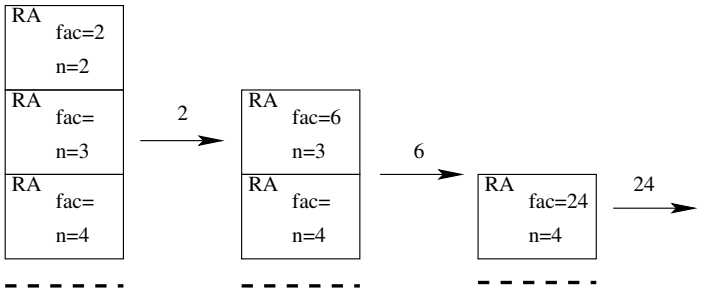
- Completamente estático
  [FORTRAN]
- Basado en una pila
  [C, C++, PASCAL, ADA, . . .]
- Completamente dinámico
  [LISP, . . .]

Memoria de un proceso

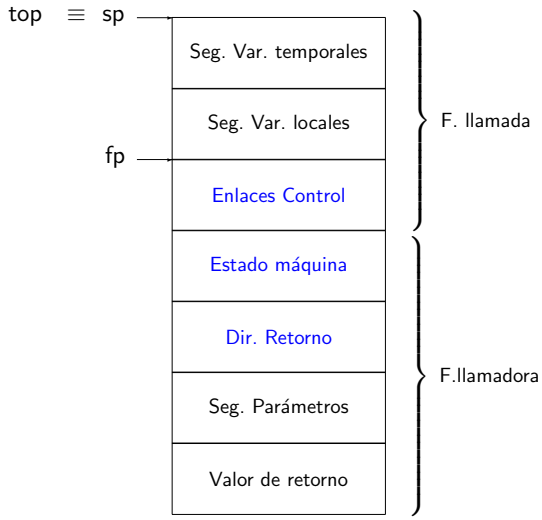



```
int fac (int n)
// Factorial de un número > 0
{
  if ( n < 3 ) return n;
  else return n * fac (n−1);
}
```





Registro de Activación

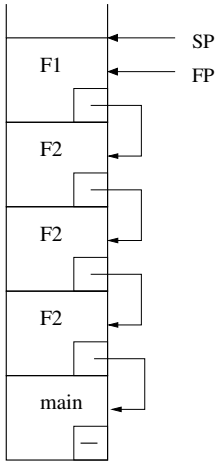


Ejemplo: LEB sin anidamiento

```
int F1 (...)
{
  ...
}

int F2 (...)
{
  ...
}

int main ()
{
 ...
}
```



Carga de los enlaces de control

- apila el fp anterior:

  $push(fp)$

- actualiza el fp:

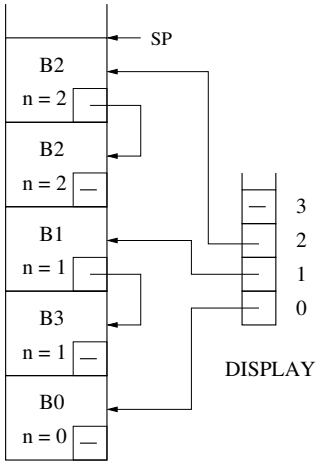
  $fp \leftarrow sp$

Descarga de los enlaces de control

- actualiza el fp:

  $fp \leftarrow pop$

Ejemplo: LEB con anidamiento



Carga de los enlaces de control

- apila el $display$ anterior:

  $push(display[n])$

- actualiza el $display$:

  $display[n] \leftarrow sp$

Descarga de los enlaces de control

- actualiza el $display$:

  $display[n] \leftarrow pop$

Acceso a los objetos en memoria

- Basado en fp
  - Acceso a variables locales $x(\delta_x)$: $fp + \delta_x$
  - Acceso a parámetros $p(\delta_p)$: $fp + \delta_p$
  - Acceso al valor de retorno (δ_{vr}): $fp + \delta_{vr}$

  $\delta_{vr} = -[SEC.talla + SP.talla + VR.talla]$
- Basado en $display$
  - Acceso a variables locales $x(n_x, \delta_x)$: $\texttt{display}[n_x] + \delta_x$
  - Acceso a parámetros $p(n_p, \delta_p)$: $\texttt{display}[n_p] + \delta_p$
  - Acceso al valor de retorno: $\texttt{display}[n] + \delta_{vr}$

$SEC.talla =$ talla del *segmento de enlaces de control* $SP.talla =$ talla del *segmento de parámetros.*
$VR.talla =$ talla del *valor de retorno.*

Carga del registro de activación

- Bloque llamador:
  - [reserva espacio para el valor de retorno] $sp = sp + VR.talla$
  - [{ apila el parámetro actual }] $\{push(p_i.pos)\}$
  - apila la dirección de retorno $push(\Omega + 2)$
  - llamada $call(dir_f)$
- Bloque llamado:
  - carga de los enlaces de control $push(fp);\ fp = sp$
  - reserva de espacio para el segmento de variables locales y temporales $sp = sp + SV.talla$

$\{VR, SV\}.talla =$ talla del *valor de retorno* y del *segmento de variables.* $dir_f =$ dirección del segmento de código asociado al bloque llamado. $\Omega =$ primera instrucción libre en el *segmento de instrucciones.*

Descarga del registro de activación

- Bloque llamado:
  - libera el segmento de variables locales y temporales $sp = fp$
  - descarga de los enlaces de control $fp = pop$
  - desapila la dirección de retorno y devuelve el control $return(pop)$
- Bloque llamador:
  - [libera el segmento de parámetros] $sp = sp - SP.talla$
  - [desapila el valor de retorno] $\ldots = pop$

$SP.talla =$ talla del *segmento de parámetros*.

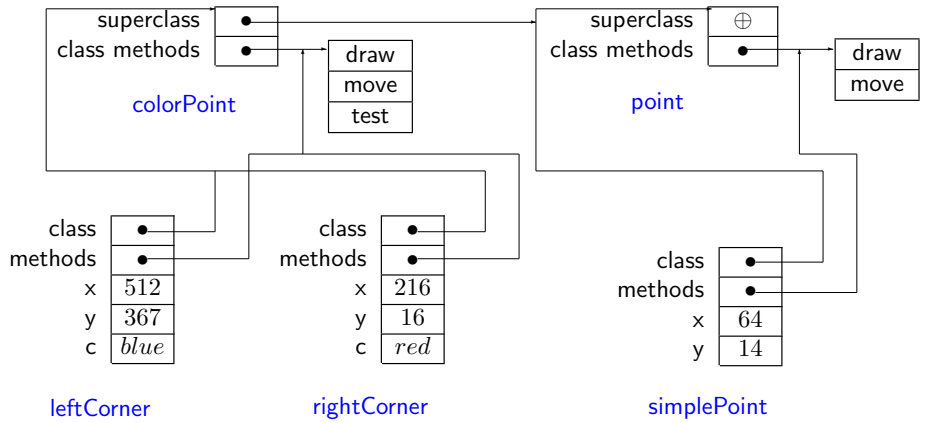
Gestión de Memoria para Lenguajes Orientados a Objetos

- Visibilidad de los objetos (ej. del [Cooper & Torczon, 2012])

```
class point {
   public int x, y;
   public void draw ( ) {...};
   public void move ( ) {...};
}
class colorPoint extends point {   // hereda:   x, y, move
   color c;                        // campo local
   public void draw ( ) {...};     // oculta el  draw de  point
   public void test ( ) {...};     // método local
}
class A {
   int x, y;                       // campos locales
   public void m ( ) {             // método local
      int y;                       // variable local de  m
      point p = new colorPoint( ); // utiliza  colorPoint, y por
      y = p.x                      //   herencia,  point
      p.draw( );      }
}
```


Gestión de Memoria para Lenguajes Orientados a Objetos

- Estructuras de datos en tiempo de ejecución: **Registro de Objeto**



Gestión dinámica de memoria: montículo

- Operaciones de Gestión Dinámica de Memoria
  - *Peticiones y liberaciones implícitas*:

    SNOBOL-4 y lenguajes lógicos y funcionales.
  - *Peticiones y liberaciones explícitas*:

    C (malloc, free); PASCAL (new, dispose); C++ (new, delete), ...
  - *Peticiones explícitas y liberaciones implícitas*:

    Java; C# y lenguajes ".net".
- Criterios de diseño del gestor de memoria:
  - Optimizar el espacio,

    $\Rightarrow$ minimizando la fragmentación
  - Optimizar el tiempo de ejecución del programa

    $\Rightarrow$ minimizando el sobrecoste de la gestión de memoria

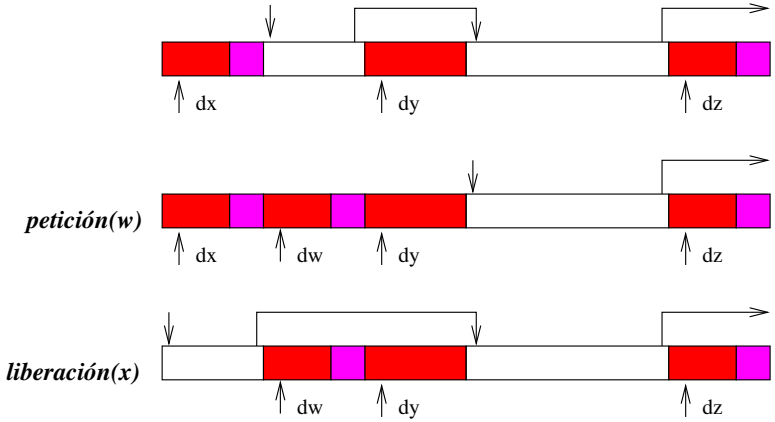
Gestión dinámica de memoria: montículo

- Gestión de bloques de talla fija:
  - \+ Poca o nula fragmentación
  - \+ Fácil implementación
  - – Uso poco adecuado de la memoria

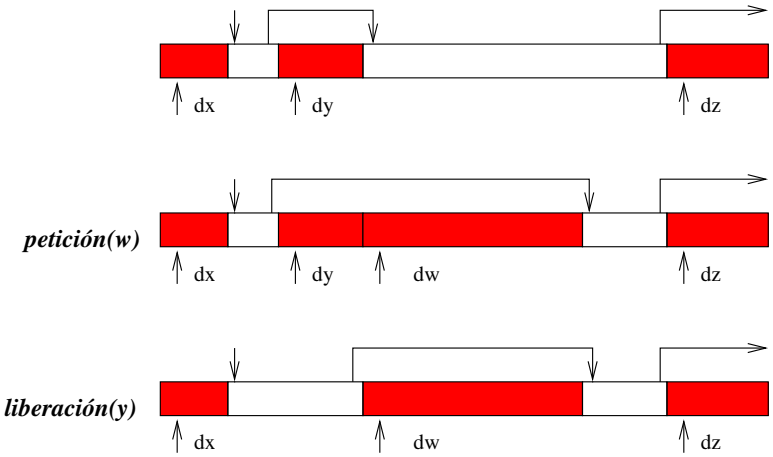
    Ejemplo: `Lisp`
- Gestión de bloques de talla variable:
  - – Posible fragmentación de la memoria
  - – Dificultad de implementación
  - \+ Uso adecuado de la memoria

    Ejemplo: `PASCAL`, `C`, `C++`, ...

Bloques de talla fija



Bloques de talla variable

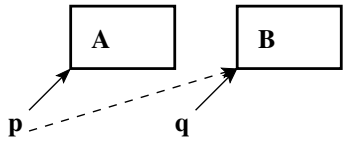


Gestión dinámica de memoria: montículo

	indicador de bloque libre	talla bloque	enlace bloque anterior	datos	enlace bloque siguiente	talla bloque	indicador de bloque libre	

- Estrategias de petición/selección de bloques
  - primer bloque
  - mejor (peor) bloque ⇒ ordenación
- Estrategias de liberación de bloques
  - **explícita** → referencias suspendidas [`PASCAL`, `C`, `C++`]
  - **implícita** → "Garbage-Collection" [`Java`, `C#`, lógico-funcionales]

➢ Desocupación sobre la marcha "free-as-you-go"



$p = q$

Se necesita un contador (punteros apuntando) en cada zona.

- acceder al contador zona A
- Si es 1, liberar zona A
- si no decrementar contador zona A
- p apunta zona B e incrementa su contador

➢ Marcar y barrer "marck-and-sweep"

Cuando queda poca memoria, cuando finaliza un módulo o bajo ciertas condiciones:

➢ buscar todas las referencias vivas y "marcar" los bloques accesibles.

➢ liberar ("barrer") todos los bloques no marcados