

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 gestión 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 (sí); MODULA2 y PYTHON (no)

- Lenguajes con Estructura de Bloques (LEB)

Un *bloque* es cualquier construcción que pueda contener declaraciones
⇒ regla de anidación más próxima

- Ámbitos de las variables en LEB

- Recursividad de funciones

- Anidamiento de funciones

Lenguajes	LEB	Recursividad	Anidamiento
FORTTRAN	no	no	no
PASCAL	sí	sí	sí
C	sí	sí	no

TABLA DE SÍMBOLOS

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

TDS

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

Tabla de Nombres

m	i	V	a	r	i	a	b	l	e	⊗	i	⊗	o	t	r	a	⊗	j	⊗		
↑										↑	↑						↑				
δ_1										δ_2	δ_3						δ_4				

TABLA DE SÍMBOLOS

- Estructura e implementación de una TDS

- array de registros;
- listas ordenadas (doblemente) enlazadas;
- árboles ordenados equilibrados;
- tablas de dispersión (*hash*).

- Operaciones sobre una TDS

- **insertar** la información de un objeto (comprobando que no existe otro con el mismo nombre en ese ámbito)
- **obtener** la información asociada con un determinado objeto (comprobando que el objeto existe en la TDS)

TABLA DE SÍMBOLOS PARA LEB

TDS para un Lenguaje con Estructura de Bloques

> Estructura e implementación

- > **Compiladores en una pasada:** la TDS se implementa como una pila
(la información de los objetos de un bloque ya no se necesitará más a la salida de ese bloque)
- > **Compiladores en más de una pasada:** la TDS se implementa como una lista ligada
(la información de los objetos se necesita en las sucesivas pasadas)

> Operaciones adicionales

- > **cargar** la información asociada con un nuevo bloque
- > **descargar** la información de un determinado 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 {
        int b = 3;
        B3 printf("%d %d\n", a, b);
    }
    printf("%d %d\n", a, b);
}
```

EJEMPLO DE LEB: PASCAL (2/4)

```
program PP;
var a, b, c: ...
. procedure H1 (x: ...);
. var b: ...
. . procedure H1A (e: ...);
. . begin
. . | ...
. . | b := a + e;          <==== Punto-1
. . | ...
. . end;
. begin
. | ...
. end;
. procedure H2 (e: ...);
. begin
. | ...
. | b := a + e;          <==== Punto-2
. | ...
. end;
begin
| ...
end.
```

EJEMPLO DE DE LEB: C (3/4)

TDB			TDS				
B0 → 0	0	1	0	a	t_a	δ_a	...
B1 → 1	2	2	1	b	t_b	δ_b	...
B2 → 2	3	3	2	b	t_b	δ_b	...
			3	a	t_a	δ_a	...

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

TDB			TDS				
B0 → 0	0	1	0	a	t_a	δ_a	...
B1 → 1	2	2	1	b	t_b	δ_b	...
B3 → 2	3	3	2	b	t_b	δ_b	...
			3	b	t_b	δ_b	...

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

TDB			TDS				
B0 → 0	0	1	0	a	t_a	δ_a	...
B1 → 1	2	2	1	b	t_b	δ_b	...
			2	b	t_b	δ_b	...

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

TDB			TDS				
B0 → 0	0	1	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

PP → 0	0	3
H1 → 1	4	6
H1A → 2	7	7

TDS

0	a	t_a	δ_a	...
1	b	t_b	δ_b	...
2	c	t_c	δ_c	...
3	H1	t_{H1}	dir_{H1}	...
4	x	t_x	δ_x	...
5	b	t_b	δ_b	...
6	H1A	t_{H1A}	dir_{H1A}	...
7	e	t_e	δ_e	...

Punto-2 →

TDB

PP → 0	0	4
H2 → 1	5	5

TDS

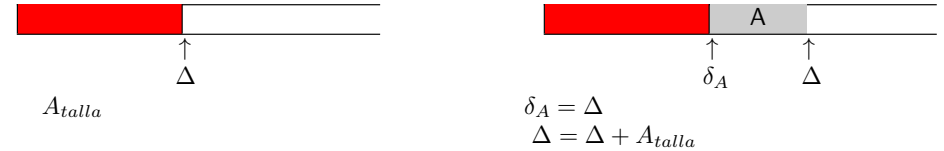
0	a	t_a	δ_a	...
1	b	t_b	δ_b	...
2	c	t_c	δ_c	...
3	H1	t_{H1}	dir_{H1}	...
4	H2	t_{H1}	dir_{H2}	...
5	e	t_e	δ_e	...

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 ⇒	$n = 0; \Delta = 0;$
LD	
LD ⇒ LD D	
⇒ D	
D ⇒ DV ;	insTdS (DV.n, "variable-global", DV.t, n, Δ); $\Delta = \Delta + DV.talla;$
DV ⇒ T id	DV.n = id.n; DV.t = T.t; DV.talla = T.talla;
⇒ T * id	DV.n = id.n; DV.t = tpuntero(T.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 $\neg [cte.t = tentero \wedge cte.num > 0]$ MenError(.); SINO DV.n = id.n; 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 ⇒ LC DV ;	LC.t = LC'.t ⊗ (DV.n, DV.t, LC'.talla); LC.talla = LC'.talla + DV.talla;
⇒ DV ;	LC.t = (DV.n, DV.t, 0); LC.talla = DV.talla;

Funciones y parámetros

(ver primero Registros de Activación)

Recursos en tiempo de ejecución

- La ejecución de un programa está inicialmente bajo el control del sistema operativo.
- Cuando se invoca un programa:
 1. El sistema operativo asigna espacio para el programa
 2. El código se carga en una parte de dicho espacio.
 3. El sistema operativo salta al punto de entrada (es decir, al "main")

Ambientes de Ejecución

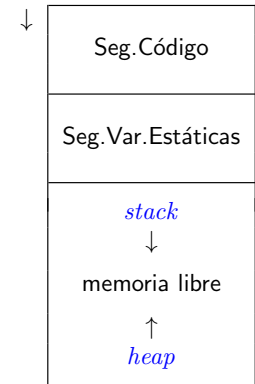
- Completamente estático [FORTRAN]
- Basado en una pila [C, C++, PASCAL, ADA, ...]
- Completamente dinámico [LISP, ...]

Estructura de la memoria de un proceso.

(en un ambiente de ejecución basado en una pila)

Memoria de un proceso

- **Seg.Código** contiene el código objeto.- Para la mayoría de los lenguajes, tamaño fijo y solo lectura.
- **Seg.Var.Estáticas** contiene los objetos con direcciones fijas (p.ej., variables globales)
- **Stack** contiene un **Registro de Activación (RA)** para cada una de las funciones activas. Cada RA suele ser de tamaño fijo y contener los segmentos de variables locales y temporales.
- **Heap** contiene todos los demás datos; por ejemplo en C, el heap se gestiona con `malloc` y `free`.



EJEMPLO DE EJECUCIÓN DE UNA FUNCIÓN

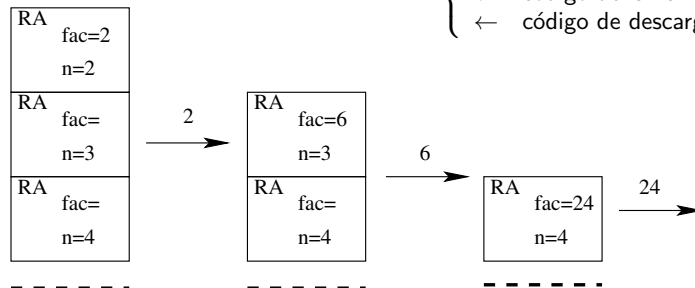
```
int fac (int n)
// Factorial de un número >0
{
  if ( n <3 ) return n;
  else return n*fac(n-1);
}
```

Código de la definición de "fac"

- ← código de carga del RA
- ← código de la función
- ← código de descarga del RA

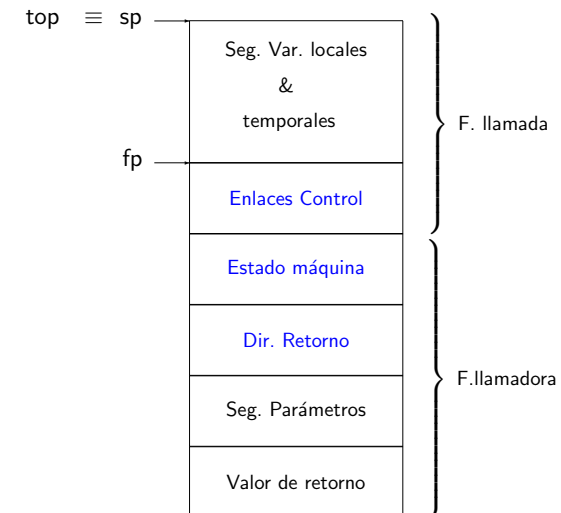
Código de la llamada a "fac"

- ← código de carga del RA
- ← código de la llamada a la función
- ← código de descarga del RA



GESTIÓN DINÁMICA DE MEMORIA

Registro de Activación



Funciones y parámetros

D \Rightarrow	$n++$; cargaCont (n); $D.aux = \Delta$; $\Delta = 0$;
T id (PF) { DL LI }	insTdS (id.n, "función", tfunción(PF.t, T.t, PF.talla), $n-1$, Δ); descargaCont (n); $n--$; $\Delta = D.aux$;
DL \Rightarrow DL DV ;	insTdS (DV.n, "variable-local", DV.t, n , Δ); $\Delta = \Delta + DV.talla$;
$\Rightarrow \epsilon$	
PF $\Rightarrow \epsilon$	PF.t = tvacio; PF.talla = 0;
\Rightarrow LF	PF.t = LF.t; PF.talla = LF.talla - TallaSegEnlaces;
LF \Rightarrow DV , LF	LF.t = DV.t \otimes LF'.t; LF.talla = LF'.talla + DV.talla;
\Rightarrow DV	insTdS (DV.n, "parámetro", DV.t, n , -LF.talla); LF.t = DV.t; LF.talla = TallaSegEnlaces + DV.talla;
	insTdS (DV.n, "parámetro", DV.t, n , -LF.talla);

cargaCont(n) y **descargaCont**(n): Carga y descarga el contexto asociado con el bloque de nivel n .

ejemplo: LEB sin anidamiento

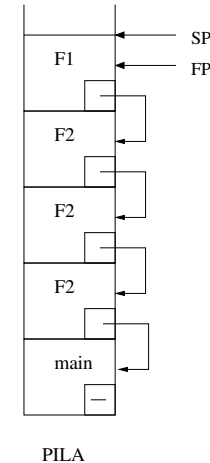
```

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 :

$fp \leftarrow pop$

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}) : $fp + \delta_{vr}$

$$\delta_{vr} = -[SEC.talla + SP.talla + VR.talla]$$

$SEC.talla$ = talla del segmento de enlaces de control, del estado de la máquina y de la dirección de retorno.

$SP.talla$ = 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 }] $\{ push(p_i) \}$

> apila la dirección de retorno $push(...)$

> apila el estado de la máquina $\{ push(...); \}$

> 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. Ω = primera instrucción libre en el segmento de instrucciones.

➤ Bloque llamado:

- libera el segmento de variables locales y temporales $sp = fp$
- descarga de los enlaces de control $fp = pop$
- descarga el estado de la máquina $\{ \dots = pop; \}$
- desapila la dirección de retorno y devuelve el control $return(pop)$

➤ Bloque llamador:

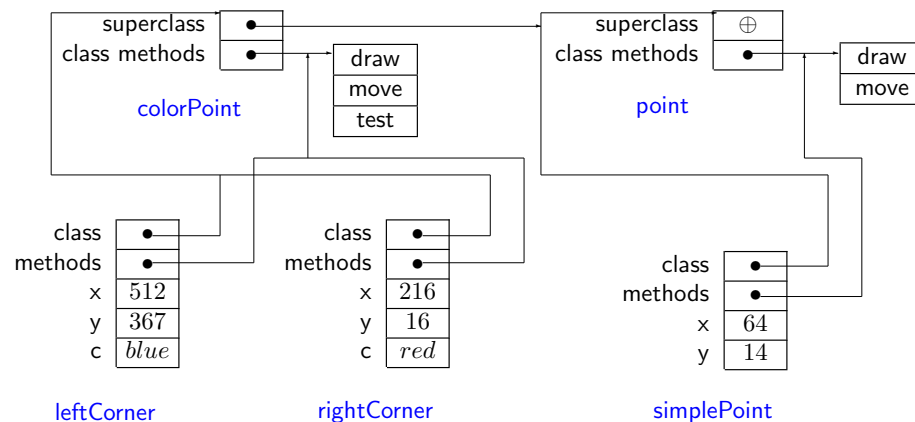
- [libera el segmento de parámetros] $sp = sp - SP.talla$
- [desapila el valor de retorno] $\dots = pop$

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

➤ 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( );
    }
}
```

➤ Estructuras de datos en tiempo de ejecución: Registro de Objeto



➤ 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,
⇒ minimizando la fragmentación
- Optimizar el tiempo de ejecución del programa
⇒ 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

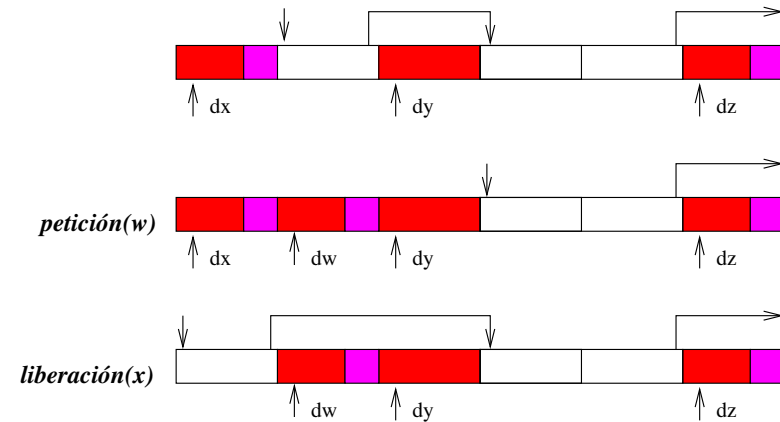
Ejemplo: Lisp

> Gestión de bloques de talla variable:

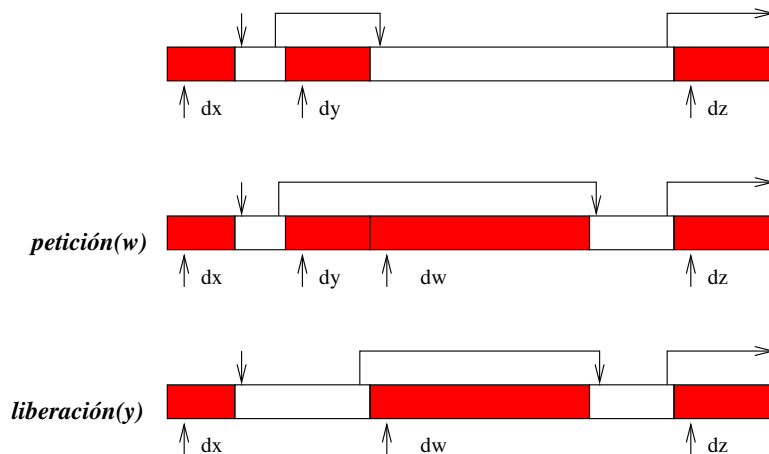
- Posible fragmentación de la memoria
- Dificultad de implementación
- + Uso adecuado de la memoria

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

BLOQUES DE TALLA FIJA



BLOQUES DE TALLA VARIABLE



GESTIÓN DINÁMICA DE MEMORIA: MONTÍCULO

contador de accesos	indicador bloque libre	talla bloque	enlace bloque anterior	datos	enlace bloque siguiente	talla bloque	indicador bloque libre	contador de accesos
---------------------------	------------------------------	-----------------	------------------------------	-------	-------------------------------	-----------------	------------------------------	---------------------------

> Estrategias de petición/selección de bloques

- > primer bloque
- > mejor (peor) bloque \Rightarrow ordenación

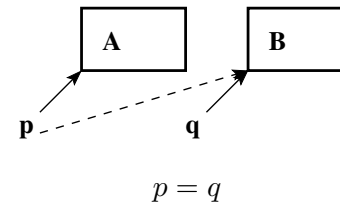
> Estrategias de liberación de bloques

- > **explícita** \rightarrow referencias suspendidas [PASCAL, C, C++]
- > **implícita** \rightarrow "Garbage-Collection" [Java, C#, lógico-funcionales]

- Un objeto X es accesible si y solo si:
 - existe un puntero a X ó
 - otro objeto accesible Y contiene un puntero a X .
- Un objeto no accesible nunca podrá ser usado y por tanto es *basura* (“*garbage*”)
- Una estrategia de *recolección de basura* (“*Garbage Collection*”) debe considerar:
 1. Asignar espacio a los nuevos objetos según se necesite
 2. Cuando el espacio libre se agote:
 - a. detectar los objetos accesibles
 - b. liberar el espacio asignado a los objetos no accesibles (que no están en (a))

Algunas estrategias activan la recolección de basura antes de que el espacio realmente se agote.

➤ Desocupación sobre la marcha “free-as-you-go”



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 “mark-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