

Gestión dinámica de la memoria

Programación II

Facultade de Informática



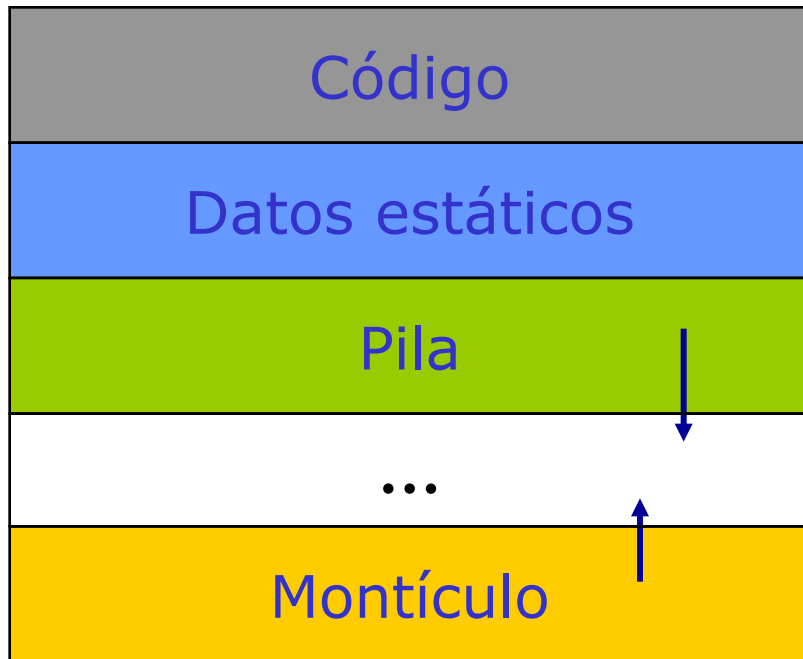
UNIVERSIDADE DA CORUÑA

Índice

1. Organización de la memoria de un programa
2. Definición de variables de tipo puntero
3. Reserva y destrucción dinámica de memoria
4. Asignación y comparación de punteros

Organización de la Memoria

- Existen dos lugares en la memoria de un computador que pueden utilizarse para almacenar elementos: la pila (stack) y el montículo (heap).



El tamaño del código y de los datos estáticos es determinado por el compilador

El tamaño de la pila y el montículo varían en tiempo de ejecución

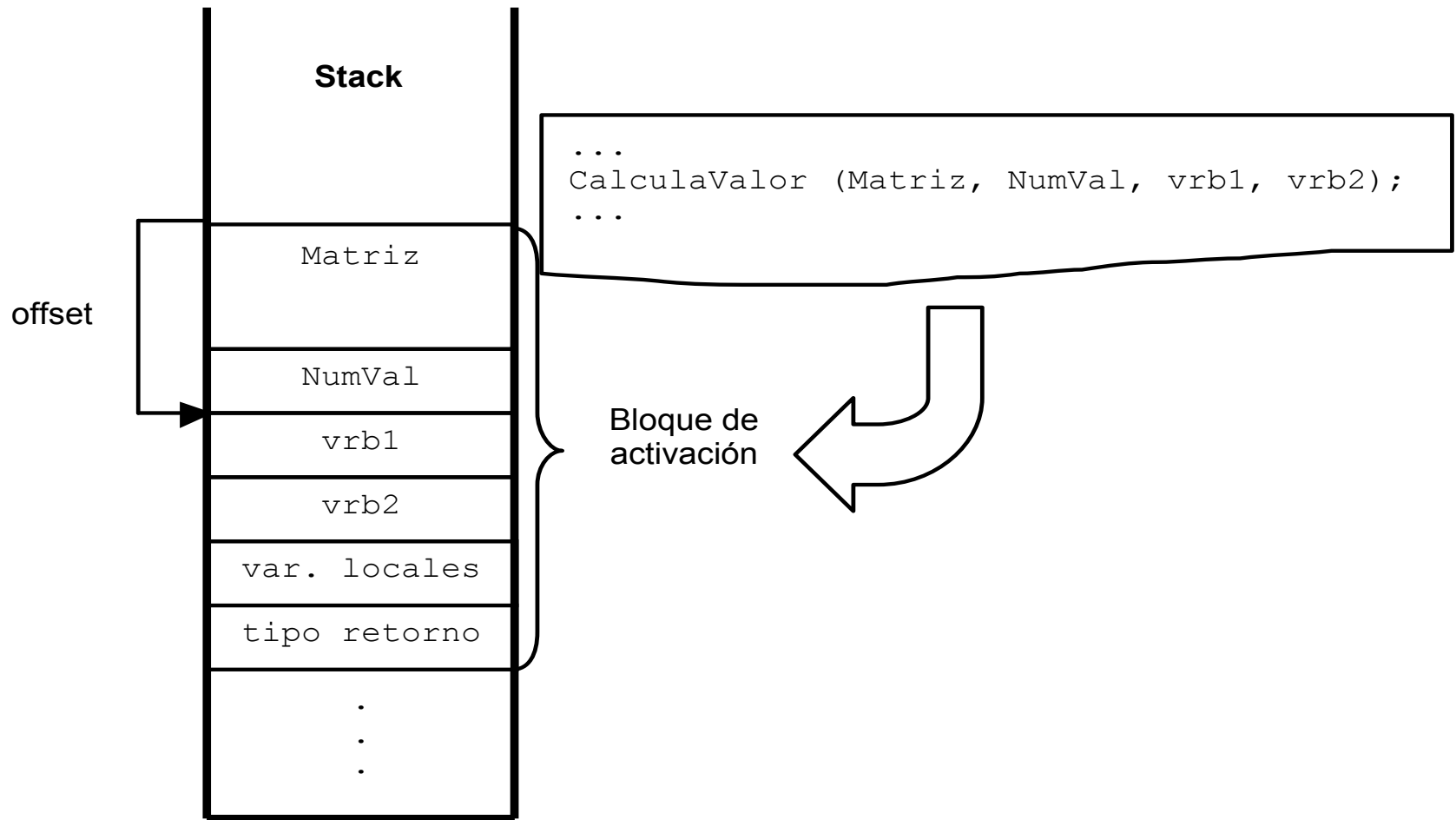
Generalmente la pila crece hacia abajo y el montículo al revés (aunque pueden estar intercambiados)

Organización de la Memoria


- Pila (stack)

- Es una estructura en la cual solo se pueden incorporar o eliminar elementos por un solo punto denominado cima.
- Se utiliza para las llamadas a funciones:
 - Cuando se llama a una función se reserva espacio en la pila para las variables y constantes locales, los parámetros, el valor de retorno, etc.
- Estos valores existen mientras se esté ejecutando la función y se eliminan (recuperando la memoria que ocupaban) cuando la función termina.
- Si una función llama a otras funciones sus valores se van amontonando en la cima de la pila.

Organización de la Memoria: La Pila



Organización de la Memoria

- Montículo (heap)
 - Parte de la memoria que no está ligada a la llamada de funciones
 - Se utiliza para almacenar variables dinámicas (cuya memoria se reserva en tiempo de ejecución)
 - La memoria se asigna cuando se solicita expresamente (a través de la función malloc)  permite asignar una cantidad concreta de memoria
 - Las variables dinámicas son accesibles a través de un tipo básico, el puntero, cuya memoria sí se reserva en tiempo de compilación ya que lo único que hace es almacenar una dirección de memoria correspondiente al montículo

Definición de punteros

- Punteros en C

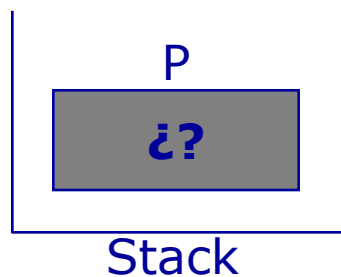
- Un puntero es un tipo básico en C (como integer, bool, char, float, etc.) y como tal ocupa en memoria una cantidad determinada conocida en tiempo de compilación (generalmente 4 bytes)
- Una variable de tipo puntero contiene una dirección en memoria en la cual se almacena una variable de otro tipo
- Las variables de tipo puntero en C son “tipadas” lo que quiere decir que un puntero se declara para que apunte a un tipo particular de datos (integer, boolean, float, etc.) y no puede apuntar a ningún otro

Definición de punteros

- **Sintaxis:** `typedef tipoApuntado* tipoPuntero`
↳ nombre del puntero
- Estamos declarando que `tipoPuntero` es una variable puntero que apunta a elementos del tipo `tipoApuntado`
- El símbolo “*” podría leerse como “puntero a”
- Por ejemplo:
 - `typedef int* tPEntero`
 - Se leería como “tPEntero es un puntero a int”

Declaración de variables tipo puntero

- Igual que cualquier otra variable
- Sintaxis: `tipoPuntero variablePuntero`
- Ejemplo: `tPEntero P`
- ¿qué ocurre cuando se declara?



En la pila se reservarán 4bytes para la variable puntero P. El valor que tiene al declararla es un número "basura"

Ejemplo de utilización de punteros

- Explicación

CODIGO

```
typedef int* tPEntero;  
  
tPEntero P1;  
int*     P2;
```

→ es lo mismo

Pero si luego
queremos cambiar
el tipo al que
apunta, es más fácil
hacerlo si esta
declarada como un
tipo de dato

- Se define un tipo `tPEntero` indicando que las variables declaradas de ese tipo contendrán direcciones de memoria en las que estarán variables de tipo entero
- La variable `P1` se declara del tipo `tPEntero` por lo cual se reserva memoria en la pila para albergar una dirección de memoria que apuntará a un entero (4 bytes) almacenado en el montículo
- Para usar los punteros no es necesario definir un tipo. Pueden usarse directamente, tal y como hace `P2`. Sin embargo, crear un tipo aumenta la abstracción del código y elimina problemas de comprobación de tipos entre los punteros ¿`P1` es del mismo tipo que `P2`?

Inicialización de punteros

- La declaración de variables reserva una zona de memoria para ellas, pero no elimina los contenidos que hubiese en esa zona ni inicializa las variables, por lo tanto `P` contendrá inicialmente un valor “basura”
- Utilizar un puntero cuyo contenido es “basura” generalmente tendrá efectos fatales para la ejecución del programa
- Se recomienda inicializar los punteros a un valor “nulo”
- Este valor se representa por la constante “NULL” en C
- Ejemplo: `P = NULL;`

Creación de una variable dinámica

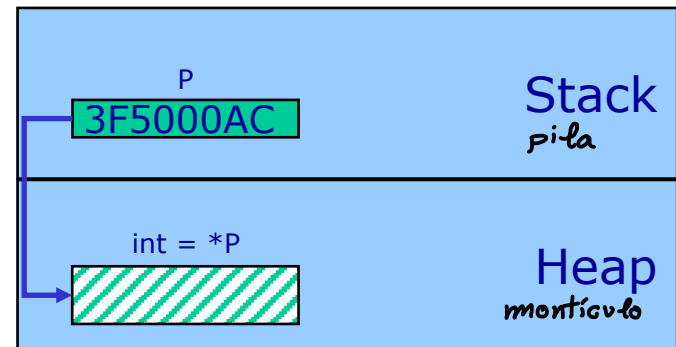
- Para crear la variable dinámica apuntada por un puntero se utiliza el operador “malloc” definido en el archivo de cabecera “stdlib”.

Ejemplo:

```
P=malloc(sizeof(int));
```

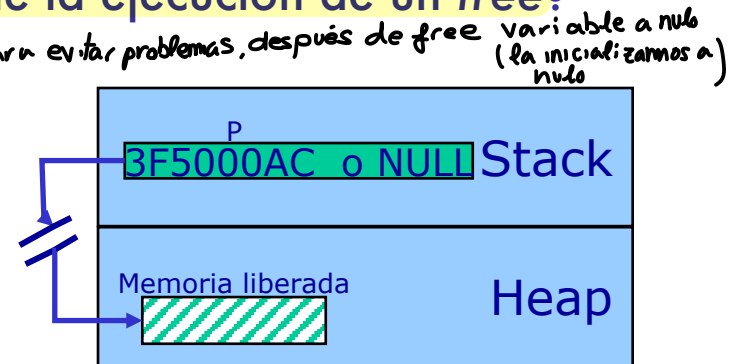
↳ stdlib.h

- Funcionamiento
 - Reserva memoria para el tipo de la variable apuntada (por el puntero), cuyo tipo se pasa por parámetro (en este caso, `int`), e inicializa el valor del puntero a la dirección de esa zona de memoria
 - Si no existe memoria suficiente para reservar el operador asigna al puntero el valor “NULL”



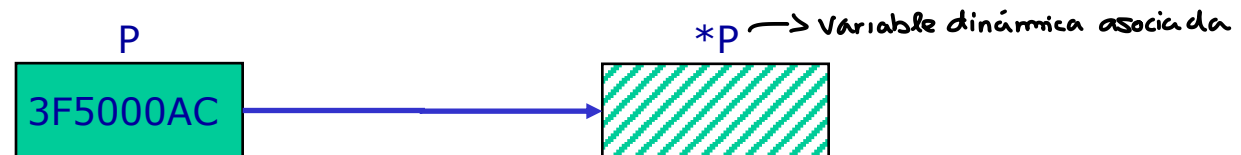
Destrucción de una variable dinámica

- Para liberar la memoria de la variable dinámica se utiliza el procedimiento “free” definido en el archivo de cabecera “stdlib”.
Ejemplo: `free(P);`
- Funcionamiento
 - Marca la memoria apuntada por el puntero como liberada, de forma que pueda volver a ser utilizada
 - ¿Qué ocurre con el valor de P después de la ejecución de un free?
 - Depende del compilador: bien P no varía de contenido (aún tiene la dirección de la variable dinámica); bien toma de nuevo el valor NULL
 - En cualquier caso, P ya no es válido para acceder a la variable apuntada y tratar de utilizarlo para acceder a ella podría en ocasiones producir un error de ejecución



Acceso a una variable dinámica

- Una vez creada, la variable dinámica se maneja como cualquier otra
- El acceso es a través del puntero y el operador de dirección `"*"`
- Toma el nombre `*P` (`*P`= "variable apuntada por P")



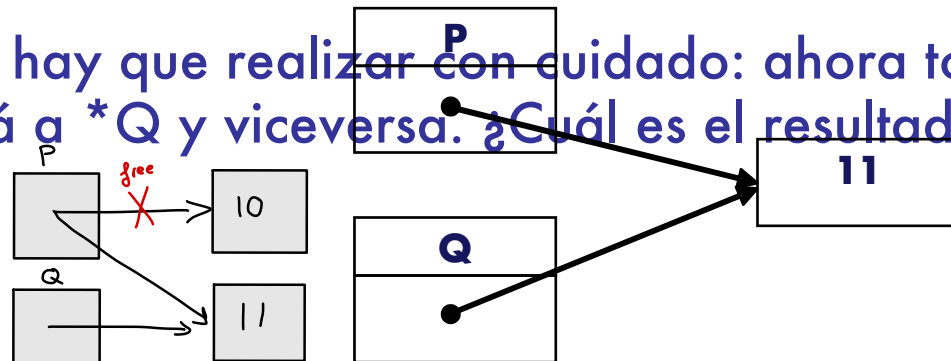
	Declaración	Creación	Escritura	Lectura
Estáticas	<code>int i;</code>	—	<code>i = 5;</code>	<code>j = i;</code>
Dinámicas	<code>int* p;</code>	<code>p = malloc(sizeof(int));</code>	<code>*p = 5;</code>	<code>j = *p;</code>

Asignación de valores a punteros

- `P = NULL;` `/* asignamos al puntero el valor nulo */`
- `P = malloc(sizeof(int));` `/* reservamos memoria y asignamos al puntero la dirección de memoria */`
- `P = Q;` `/* Asignamos al puntero el valor de otro puntero */`

Es una operación que hay que realizar con cuidado: ahora todo cambio en `*P` afectará a `*Q` y viceversa. ¿Cuál es el resultado de

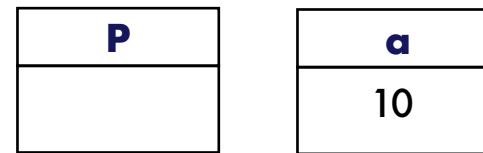
```
*P = 11;  
printf("%d", *Q);  
*Q = 4;  
P = Q;  
*P = 11;  
printf("%d", *Q);
```



Asignación de valores a punteros

Uso no recomendado

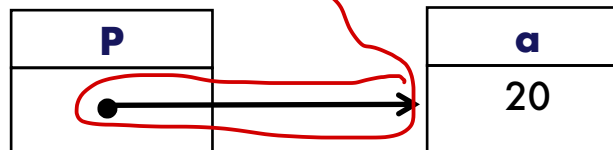
- `int a;`
- `int *P;`
- `a = 10;`



- `P = &a;`
entera */

/* Asignamos a P la dirección de la variable *a*

- `*P = 20`
- `printf(" %i",a);`



Operación de Comparación

- Comparación de punteros:
 - Dos punteros son iguales si contienen la misma dirección de memoria
- Comparación de variables dinámicas:
 - Identidad:
 - Dos variables dinámicas son idénticas si y solo si están en la misma dirección de memoria.
 - Igualdad:
 - Dos variables dinámicas son iguales si, a pesar de estar en direcciones de memoria distintas, su contenido es el mismo

