

# Capítulo 2

## Memoria dinámica

### 2.1. División de la memoria

A la memoria del ordenador la podemos considerar como una tira de celdas dividida en 4 segmentos lógicos que pueden variar en tamaño. Estos segmentos son los que vemos en la figura 2.1.1. Veamos cada uno:

- El Code Segment, o segmento de código, es donde se localiza el código resultante de compilar nuestra aplicación. Es decir, la algoritmia en código máquina.
- El Data Segment, o segmento de datos, almacena el contenido de las variables definidas como externas (variables globales) y las estáticas .
- El Stack Segment, o pila, almacena el contenido de las variables locales en la invocación de cada función, incluyendo las de la función main.
- El Extra Segment, o heap, es la zona de la memoria dinámica.

Cada celda corresponde a un byte, la unidad mínima de memoria. Recordemos que un byte es la conformación de 8 bits. En cada bit podemos almacenar dos intensidades distintas de corriente, en donde la intensidad más alta se interpreta como el valor 1, y la más baja como el valor 0. Por este motivo, en cada byte se pueden almacenar hasta  $2^8 = 256$  valores diferentes. Cada celda se identifica con un número que llamamos *dirección*, este número es correlativo. En general se representa en formato hexadecimal por ser múltiplo de 8, quedando como en el gráfico 2.1.2.

Cada variable ocupa una o más de estas celdas, dependiendo del tipo de variable y la plataforma de la máquina. Por ejemplo, si una variable de tipo *int* ocupa 4 bytes, se reservan cuatro celdas contiguas, siendo la dirección

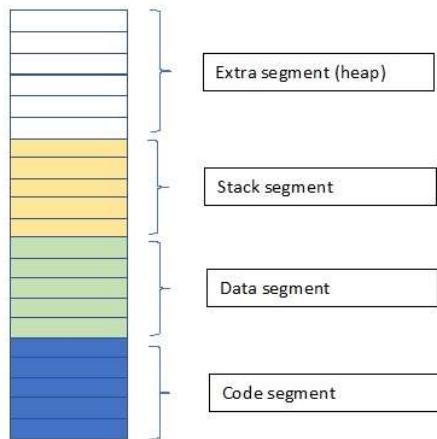


Gráfico 2.1.1: División lógica de la memoria



Gráfico 2.1.2: Direcciones de memoria

de la primera celda la dirección correspondiente a esa variable. Una variable de tipo *bool* ocupa un byte entero, aunque con un bit alcanza para guardar todos sus valores (0 o 1). Los otros 7 bits los rellena con ceros. Si queremos saber en qué dirección está determinada variable debemos usar el operador `&`, pero para imprimir su valor a veces, dependiendo de la plataforma, debemos indicar que lo tome como un entero sin signo. Por ejemplo:

```
int a = 8;
cout << (unsigned)&a << endl;
```

En este código, la variable entera *a* almacena el valor 8, y se encuentra en determinada dirección. Esta dirección se mantendrá fija durante todo el tiempo de vida de la variable, es decir, hasta que encuentre una llave de cierre de bloque, pero puede variar en las distintas ejecuciones del programa. En general, las direcciones en dónde están las variables no nos interesarán conocerlas, salvo a modo ilustrativo.

De la gestión de los tres primeros bloques de memoria: el segmento de código, el de datos y el de pila, se encarga el compilador, quien reserva y libera los bloques de memoria de manera automática.

La gestión del bloque identificado como heap será responsabilidad del desarrollador. Es decir, los bloques de memoria que se soliciten al heap deben hacerse con una determinada instrucción en el programa, y cuando la variable no se necesite más debemos liberar la memoria ocupada con otra instrucción, en caso de no hacerlo esa porción de memoria queda inutilizada hasta que apaguemos la máquina.



Nota: debemos ser cuidadosos con el uso de la memoria, ya que si no la liberamos no se recupera ni siquiera cuando finaliza el programa.

## 2.2. Punteros

Un puntero es un tipo especial de variable que almacena direcciones de memoria. Es decir, en lugar de guardar datos como una edad, un sueldo, una nota, etc, almacena direcciones de memoria de la propia máquina. Cuando una variable de tipo puntero guarda una dirección de memoria decimos que *apunta* a esa dirección.

El lector puede preguntarse cuál es la necesidad de guardar esta información. Esta pregunta será respondida más adelante, primero veamos cómo se definen y cómo es el funcionamiento.

## Punteros a datos simples

Un puntero tiene que conocer, salvo alguna excepción que veremos pronto, de qué tipo será la variable que esté en la dirección de memoria que almacenará su valor. Dicho de otra forma, qué tipo de dato está guardado o se guardará en la dirección adonde apunta. Por ejemplo, si un puntero guarda la dirección de memoria *AB0012*, debe saber qué tipo de dato se almacenará en dicha dirección, si será un *int*, un *float* o cualquier otro. Esto se debe a dos razones: en primer lugar debe saber qué cantidad de celdas ocupará el dato que está en dicha dirección. En segundo lugar debe saber cómo interpretar esos datos. Tenemos que recordar que cada celda está formada por bits pero esos bits pueden ser interpretados de distinta manera, por ejemplo, como enteros, como flotantes, si representan un carácter en código ASCII, etc. Por lo tanto, cuando definimos una variable de tipo entero debemos decir a qué tipo de variable apuntará.

Para llevar a cabo su función de almacenamiento debe ocupar el tamaño que ocupa una dirección, es decir una palabra de la arquitectura de la máquina. En el caso de procesadores de 16 bits, son 2 bytes, para los de 32, 4 bytes y, para los de 64, 8 bytes.

¿Cómo se declara una variable de tipo puntero? Al igual que cualquier otra variable, indicando su tipo y un identificador de la variable, con la salvedad que luego de indicar el tipo se debe colocar el signo asterisco “\*” que indica que es un puntero. Ejemplos:

```
int * pi;
char * pc;
float * pf;
```

El símbolo asterisco puede ir inmediatamente después del tipo

```
int* pi;
inmediatamente antes del identificador de la variable
int *pi;
```

o como en los ejemplos indicados que no está contiguo a ninguna otra expresión. Hay que tener ciertos cuidados, por ejemplo:

```
int* pi, pi2, pi3;
```

no declara tres variables de tipo punteros a enteros, sino solo una, la primera. Tanto *pi2* como *pi3* son variables de tipo enteras.

Ahora bien, ¿cómo asignamos una dirección de memoria a una variable de tipo puntero? Las direcciones de memoria no podemos asignarlas de forma explícita, es decir, no podemos escribir algo de este estilo:



```
| int * p = 11206656;
```

Suponiendo que 11206656 es una dirección de memoria válida y que se encuentra disponible, el código anterior es erróneo. Sí podemos asignar la

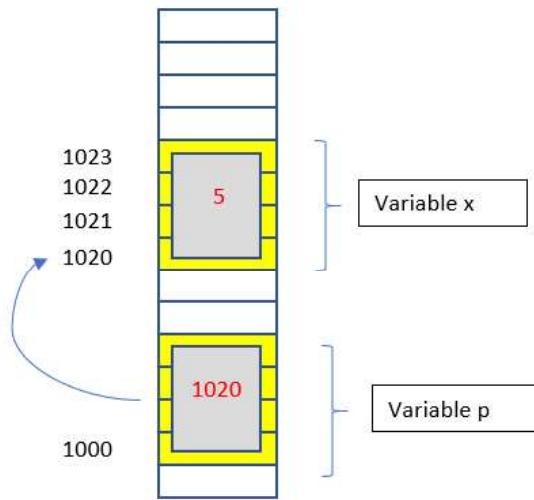


Gráfico 2.2.1: Ejemplo del contenido de una variable de tipo puntero

dirección de otra variable utilizando el operador “`&`”. Ejemplo:

```
int * p;
int x = 5;
p = &x;
```

Supongamos que la variable `p` está ubicada en la dirección 1000 y la variable `x` en la 1020. Entonces, la variable `p` contendrá como dato el valor 1020. Si un entero ocupa 4 bytes, la memoria quedará como vemos en el gráfico 2.2.1. La flecha indica la imagen gráfica del sentido de apuntar, de ahí el nombre de este tipo de variables.

Siguiendo con este ejemplo, si quisiéramos modificar el valor de `x` a, por ejemplo, 8 podríamos hacerlo indirectamente de la siguiente forma:

```
*p = 8;
```

La sentencia anterior equivale a

```
x = 8;
```

Hay que tener cuidado porque el operador `*` tiene distinta funcionalidad según el caso, decimos que está *sobre cargado*. En un caso sirve para declarar una variable de tipo puntero. Es decir, su significado es decir que la variable que sigue es un puntero. En cambio, cuando la variable ya está declarada se llama operador de desreferenciación. En este caso el significado es el siguiente “el contenido de la dirección a la que apuntas” o “dirígete a la dirección a la que apuntas”.

Observen un momento el siguiente código y piensen qué hace cada instrucción y si son válidas, antes de ver los comentarios.

```

1 int main() {
2     int x = 5, y = 8;
3     int *p1;
4     int *p2;
5     p1 = &x;
6     p1 = &x;
7     p2 = &y;
8     cout << p1 << endl;
9     p1 = p2;
10    *p1 = *p2;
11    return 0;
12 }

```

En la línea 2 definimos dos variables de tipo enteras,  $x$  e  $y$ , asignándoles los valores de 5 y 8, respectivamente. Luego, en la línea 3 y 4 declaramos dos variables de tipo puntero a entero.

La línea 5 da error al momento de compilar ya que  $p1$  espera una dirección, no el valor de  $x$  que contiene 5. También sería inválido hacer

$*p1 = x;$

dado que  $p1$  aún no apunta a ningún lado.

Las líneas 6 y 7 son válidas:  $p1$  pasa a tener la dirección de  $x$  o simplemente decimos que  $p1$  apunta a  $x$ , y  $p2$  apunta a  $y$ . La línea 8 podría dar error dependiendo de cómo tengamos configurado nuestro compilador, dado que intenta imprimir el valor que tiene  $p1$ , y este valor es una dirección. Puntualmente, la dirección de  $x$ . En caso de error podríamos castearlo a una variable de tipo entera sin signo (unsigned). Sin embargo, salvo por la finalidad de investigar cuáles son las direcciones de las variables no tendremos el deseo o la necesidad de imprimir una dirección. Sí nos interesaría imprimir el contenido que hay en esa dirección. Es decir, si reemplazamos esa línea por  $*p1$  imprime el valor 5.

Las líneas 9 y 10 son ambas correctas pero pueden llegar a confundirnos. En la línea 9 estamos diciendo que  $p1$  apunta al mismo lugar que  $p2$ , es decir, a la variable  $y$ . De esta forma,  $p1$  dejará de tener la dirección de  $x$  para pasar a tener la de  $y$ . En cambio, en la línea 10 lo que estamos igualando son los contenidos. Supongamos que la línea 9 no se ejecutó, por lo que  $p1$  conserva la dirección de  $x$  y  $p2$  la de  $y$ . De esta manera, cada puntero seguiría apuntando al mismo lugar pero se cambiaría el contenido de la variable  $x$  por un 8.

### 2.3. Punteros y vectores

Cuando declaramos un vector el compilador reserva la memoria solicitada y guarda solo la dirección del primer elemento. Por ejemplo, si declaramos un vector de 10 enteros

```
int vec[10];
```

el compilador reserva 10 lugares contiguos para almacenar los enteros. De estos 10 lugares solo guarda la dirección del primero en la variable que llamamos *vec*. Luego, es responsabilidad del programador no excederse de los 10 lugares pedidos. Supongamos que en dicho vector guardamos los números 100, 200, 300, etc. La siguiente línea

```
cout << vec[5] << endl;
```

imprime 600, la sexta posición del vector (hay que recordar que la primera posición es la 0). En cambio, la siguiente instrucción

```
cout << vec << endl;
```

debería imprimir (si el compilador no se queja) la dirección en donde está el número 100, ya que es el primer dato. Por lo tanto, como en la variable *vec* estamos guardando una dirección la podemos pensar como un puntero. Esto es cierto si lo pensamos como un puntero constante, ya que una variable de tipo puntero podemos cambiarle su contenido, lo cual la haría apuntar a otro lugar. En cambio, a una variable de tipo vector, no. Con esto en mente, la siguiente instrucción es válida:

```
int * p = vec;
```

O también como parámetro de una función:

```
void cargar (int * p, int n);
```

Entonces, si manejamos un vector a través de punteros, ¿cómo accedemos a sus posiciones? La primera respuesta a esto es mantener la lógica que vimos de punteros con el operador de desreferenciación:

```
*p // accede a la primera posición  
*(p+1) // accede a la segunda posición  
// etc
```

Sin embargo, tenemos otra forma más natural de acceder que es utilizando los corchetes con índices como con cualquier variable de tipo vector:

```
p[0] // accede a la primera posición  
p[1] // accede a la segunda posición  
// etc
```

Por lo visto, los vectores y los punteros no presentan diferencias, salvo que los primeros son constantes. Cuidado: los valores de los vectores pueden modificarse pero siempre en la misma porción de memoria.

El caso de las matrices lo veremos un poco más adelante, dado que es más complejo: hay que pensarlas como un vector de punteros, pero el primer vector también lo podemos pensar como un puntero. Todo esto implica un doble puntero que veremos en las otras secciones.

## 2.4. Pedidos de memoria y liberación

Lo que vimos hasta el momento sobre las variables de tipo puntero no tendría sentido si no tuviéramos la necesidad de utilizar memoria dinámica. Es decir, memoria que se solicita por el programador en el momento de ejecución. Esta zona de la memoria es la que llamamos *heap*.

¿Cómo solicitar memoria al heap?

Mediante la instrucción *new tipo*. Por ejemplo

```
new int;
```

crea una nueva variable de tipo int en la zona del heap. Sin embargo, esta variable anónima la perderíamos si no tuviéramos la dirección en donde fue creada. El operador *new* hace algo más que crear la variable solicitada: devuelve su dirección, la cual debemos guardarla en una variable adecuada. Una variable adecuada para guardar direcciones es un puntero, por lo tanto, la instrucción correcta es:

```
int * p = new int;
```

De esta forma *p* guarda la dirección de esa variable anónima y puede acceder mediante el operador *\** para guardar / recuperar valores.

Cuando trabajemos con memoria dinámica debemos tener cuidado porque la memoria solicitada hay que liberarla. C++ no cuenta con un recolector de basura como el lenguaje Java que libera la memoria automáticamente, por lo que la memoria que no se libere quedará inhabilitada para su uso, aún cuando el programa hubiera finalizado.

La instrucción para liberar la memoria es con el operador *delete*, de la siguiente forma:

```
delete p;
```

### Solicitud y liberación de un bloque de memoria

La potencia de los punteros se aprecia en las estructuras dinámicas. Si quisiéramos crear un vector de enteros en forma dinámica podemos solicitar todo un bloque al *heap*, de la siguiente forma:

```
int * p = new int [100];
```

En la instrucción anterior estamos creando un bloque contiguo de 100 enteros del cual guardamos la dirección del primer entero en la variable *p*. Ya vimos cómo podemos acceder a cada uno de esos enteros, trabajando con la variable *p* como si fuera un vector común y corriente. La única consideración es que al momento de liberar debemos indicarle al compilador que debe liberar todo el bloque colocando corchetes vacíos luego del operador *delete*, de esta forma:

```
delete [] p;
```

No debemos indicarle cuál es el tamaño del bloque a liberar ya que el mismo compilador se encarga de liberar todo el bloque solicitado.

## 2.5. Punteros a estructuras complejas

Supongamos que tenemos un struct Empleado:

```
struct Empleado {
    int legajo;
    string nombre;
    float sueldo;
};
```

Si queremos crear una variable de tipo Empleado en forma dinámica utilizaremos el operador *new* como en los demás casos:

```
Empleado * pe = new Empleado;
```

Para acceder a alguno de los campos deberíamos usar el operador de desreferenciación *\** sumado al punto ( . ). Por ejemplo:

```
(* pe).legajo = 145;
```

La línea anterior se interpreta:

*\* pe* es la variable a la cual apunta *pe*, en este ejemplo es una variable de tipo *Empleado*. Luego

( . ) accede al campo deseado.

Hay otra manera que se utiliza más: consiste en emplear el operador flecha -> para acceder a los campos, de esta manera:

```
pe->legajo = 145;
```

Para liberar la memoria es igual que con cualquier variable simple:

```
delete pe;
```

## 2.6. Doble punteros

Los punteros dobles, triples, etc. los usaremos cuando necesitemos direcciones de direcciones. El caso más común se da con las matrices dinámicas. Una matriz dinámica la podemos pensar de la siguiente forma: cada fila es un vector dinámico. Por lo tanto, la matriz entera es un vector de vectores dinámicos. En el gráfico 2.6.1 vemos en la parte *a*) una matriz de 4 filas por 5 columnas. En la parte *b*) se representa la idea de la misma matriz pensada como un vector de vectores. En cada posición del vector tenemos un puntero simple, en cambio, en la primera posición de la estructura necesitamos un puntero de punteros, representado como *pp*.

El código para armar una matriz de esta forma es el que apreciamos en el algoritmo 2.1. En primer lugar se crea el vector de punteros que tendrá la

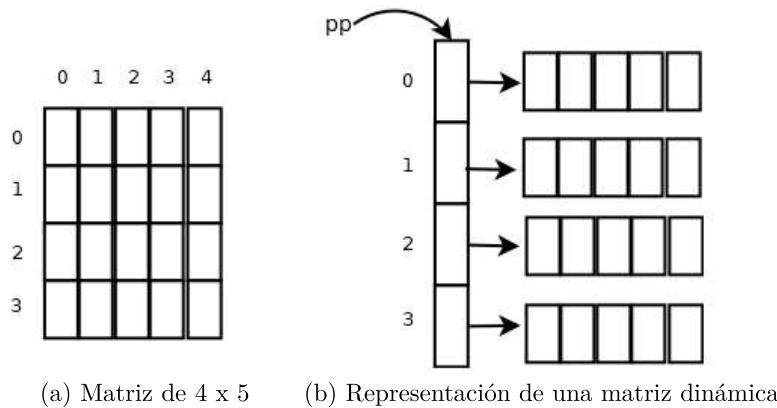


Gráfico 2.6.1: Matrices dinámicas

---

**Algoritmo 2.1** Creación de una matriz dinámica

---

```
int ** matriz;
matriz = new int* [4]; // se crea el vector de vectores
for (int i = 0; i < 4; i++) // para cada una de las filas
    matriz[i] = new int[5]; // se crea el vector fila
```

---

longitud de las filas de la matriz. Esta dirección se guarda en el doble puntero *matriz*. Luego se accede a cada una de las posiciones del vector y se crea el vector de enteros que representará cada una de las filas.

Para la eliminación el proceso es inverso:

- Se eliminan cada una de las filas.
- Finalmente se elimina el vector que sostiene la estructura.

El código de liberación es el siguiente:

```
for (int i = 0; i < 4; i++) // para cada una de las filas
    delete [] matriz[i]; // se elimina la fila
delete [] matriz; // se elimina el vector de vectores
```

La versatilidad de manejar las matrices de esta manera es que podemos crear matrices irregulares, en donde las filas tienen distinta longitud, como apreciamos en el gráfico 2.6.2.

## 2.7. Operaciones con punteros

Las operaciones que podemos hacer con los punteros están acotadas.

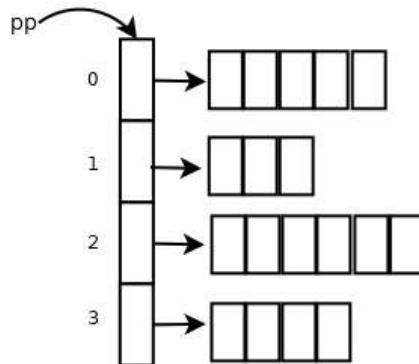


Gráfico 2.6.2: Matrices irregulares

- Asignación. Vimos que podemos asignarle direcciones de variables de dos formas:

- Si la variable no es dinámica con el operador `&`. Ejemplo

```
int x = 5;
int * p = &x;
```

Esta forma de asignación no es muy utilizada, salvo en la didáctica.

- Si la variable es dinámica se crea con el operador `new`. Ejemplo:

```
int * p = new int;
```

Esta forma es más utilizada, en especial cuando se crean bloques de memoria.

- Hay un solo valor válido que se puede asignar a un puntero que no sea la dirección de otra variable: el valor nulo. Se realiza poniendo un cero a la variable o un `NULL`:

```
int * p = 0;
int * p = NULL;
```

Las asignaciones anteriores son equivalentes pero se prefiere colocar un 0 porque la constante `NULL` no siempre está definida y puede dar error.

¿Para qué utilizar esta dirección nula? Se utiliza para saber si el puntero apunta a cierta variable o no.

- Suma y resta.

- La suma de un valor a un puntero hace que este puntero avance a las posiciones contiguas de memoria. Por ejemplo:

```
int * p = new int[100]; // p apunta al primer lugar del vector
p++; // p apunta al segundo lugar del vector
```

- Con la resta sucede al contrario que con la suma: el puntero retrocede. Hay que tener en cuenta que para liberar la memoria el puntero debe apuntar a la primera dirección.

**Ejemplo.** Utilización del valor nulo. En C++ al igual que en C el valor nulo representa un falso a la hora de evaluar una condición lógica y cualquier valor distinto de cero lo interpreta como cierto, por lo que podemos hacer lo siguiente:

```
if (p) // equivale a p != 0
    // p apunta a cierta dirección válida
    // realizar A
else // p no apunta a ninguna dirección
    // realizar B
```

## 2.8. Persistencia de datos

En el capítulo anterior habíamos visto que las variables nacen cuando se las declara y mueren cuando finaliza el bloque en donde fueron declaradas. Por ejemplo:

```
int main() {
    int x = 5; // nace x
    if (x != 10) {
        int y; // nace y
        //...
    } // muere la variable y
    //...
    return 0;
} // muere la variable x
```

Sin embargo, con el manejo de la memoria dinámica a través de los punteros podemos tener persistencia de datos. Esto significa que las variables pueden trascender el bloque en donde fueron definidas. Por ejemplo, una función que pide memoria dinámica y devuelve un puntero:

```
int* f() {
    int* p; // nace p
    p = new int; // nace una variable anónima, su dirección la guarda p
    //...
    return p;
} // la variable p muere pero la anónima sigue viviendo
```

En estos casos hay que tener mucho cuidado ya que es fácil olvidarse de liberar la memoria porque la función que la pide no se encarga de hacerlo. Debería encargarse de su liberación la función que llama a *f* o alguna otra.

Otra variante de hacer extender el ámbito de vida de una variable es a través de un puntero pasado por referencia. El siguiente código es equivalente al anterior:

```
void f(int* &p) {
    p = new int;      // nace una variable anónima, su dirección la guarda p
    //...
}
```

## Ejercicios