Metodología de la Programación

Tema 1. Punteros y memoria dinámica

Departamento de Ciencias de la Computación e I.A.

Curso 2016-17

¡Esta es una licencia de Cultura Libre!

Contenido del tema

Parte I: Punteros

Contenido del tema

Parte II: Gestión Dinámica de Memoria

Motivación

Limitación de los arrays de bajo nivel: tamaño fijo y dimensionamiento mediante una constante. No es posible:

- pedir al usuario el número de elementos a procesar
- dimensionar específicamente la colección para dicho tamaño

Solución actual: sobredimensionado para evitar problemas

Motivación

Consecuencia: uso excesivo e innecesario de memoria. Se trata de ver aquí como resolver este problema mediante:

- dimensionado en tiempo de ejecución
- necesario nuevo tipo de datos que permita almacenar direcciones de memoria: puntero

Objetivos

Objetivos:

- familiarizarse con el uso de punteros
- entender la relación entre punteros y arrays
- realizar operaciones con punteros
- usar punteros como argumentos a funciones
- poder definir punteros a funciones
- conocer la forma de gestionar la memoria de forma dinámica

Importante: comprensión de los ejemplos de código y realización de los ejercicios que se vayan indicando

Parte I

Punteros

Contenido del tema

Punteros: declaración e inicialización

Concepto clave: cada variable se asigna a una posición de memoria, caracterizada por una dirección

Puntero: dirección de una posición de memoria

Punteros: declaración e inicialización

Cada byte de memoria tiene una dirección única. En base a esto:

- toda las variables se ubican en una zona de memoria suficientemente grande como para almacenar los valores que debe alojar (char: 1 byte; short: 2 bytes; int: 4 bytes; float/long: 4 bytes; ....)

Punteros: declaración e inicialización

Imaginemos la siguiente declaración de variables

```
char letra;
short numero;
float temperatura;
```

¿Cómo se ubican en memoria?

Punteros: declaración e inicialización

En C++ el operador & (**operador de dirección**) permite obtener la dirección de memoria en que se ubica una variable

```
// Direccion de memoria de variable letra
&letra
```

Punteros: declaración e inicialización I

```
1  #include <iostream>
2
3  using namespace std;
4
5  int main(){
6     char letra;
7     short numero;
8     float temperatura;
9
10    // Se muestra la direccion de memoria de las variables
11    cout << "Dir. mem. letra: " << &letra << endl;
12    cout << "Dir. mem. numero: " << &numero << endl;
13    cout << "Dir. mem. temperatura: " << &temperatura << endl;
14 }
```

Punteros: declaración e inicialización I

La salida obtenida es:

```
Dir. mem. letra: 140730899979118
Dir. mem. numero: 140730899979116
Dir. mem. temperatura: 140730899979112
```

Punteros: declaración e inicialización I

Puntero: variable que almacena una dirección de memoria.

Importante:

- la declaración del puntero debe indicar el tipo de dato (el que contendrán las direcciones de memoria que podrá tomar como valor). En realidad habría que decir puntero a entero, puntero a char, puntero a objetos clase Punto, ...
- un puntero sólo almacena direcciones de memoria

Punteros: declaración e inicialización I

Ejemplo de declaración de puntero a entero

```
int * ptr;
```

El operador * se conoce como operador de indirección

Punteros: declaración e inicialización I

Ejemplo de uso de operadores relacionados **&** y *****:

```
1  #include <iostream>
2
3  using namespace std;
4
5  int main(){
6     int x=25;
7     int *ptr;
8
9     // Se hace que ptr apunte a x
10    ptr=&x;
11
12    // Ahora podemos acceder a la posicion de memoria donde
13    // se almacena x de varias formas
14    cout << "Valor de x (habitual): " << x << endl;
15    cout << "Valor de x (puntero): " << *ptr << endl;
16    cout << "Dir. mem. x (directa): " << &x << endl;
17    cout << "Dir. mem. x (puntero): " << ptr << endl;
18 }
```

Punteros: declaración e inicialización I

Salida:

```
Valor de x (habitual): 25
Valor de x (puntero): 25
Dir. mem. x (directa): 0x70fe34
Dir. mem. x (puntero): 0x70fe34
```

Punteros: declaración e inicialización I

Una vez que un puntero apunta a una variable, puede usarse también para cambiar el valor almacenado en ella, mediante el operador *****, que desreferencia al puntero. De esta forma

```
int x;
int *ptr=&x;
.....................................
// Dos formas de almacenar
// valor en la variable x
x=23;
*ptr=35;
```

Punteros: declaración e inicialización I

Ejemplo de cambio de valor de variable con puntero:

```
1  #include <iostream>
2
3  using namespace std;
4
5  int main(){
6     int x=25;
7     int *ptr;
8
9     // Se hace que ptr apunte a x
10    ptr=&x;
11
12    // Se cambia el valor almacenado en x
13    *ptr=100;
14
15    // Se muestra el valor de x de las dos formas posibles
16    cout << "Valor de x (habitual): " << x << endl;
17    cout << "Valor de x (puntero): " << *ptr << endl;
18 }
```

Punteros: declaración e inicialización I

Un puntero puede cambiar su valor (dirección de memoria a la que apunta):

```
1  #include <iostream>
2
3  using namespace std;
4
5  int main(){
6     int x=25, y=50, z=75;
7     int *ptr;
8
9     // Se muestran los valores de las tres variables
10    cout << "x: " << x << " y: " << y << " z: " << z << endl;
11
12    // Se usa el puntero para modificar los valores
13    // de las tres variables
14    ptr=&x;
15    *ptr=*ptr*2;
16    ptr=&y;
17    *ptr=*ptr*2;
18    ptr=&z;
19    *ptr=*ptr*2;
20
21    // Se muestran los valores de las tres variables
22    // tras el cambio
23    cout << "x: " << x << " y: " << y << " z: " << z << endl;
24 }
```

Punteros: declaración e inicialización I

Gráfico del funcionamiento del código anterior:

Punteros: declaración e inicialización I

Usos de * en C++:

- multiplicación
- definición de puntero
- operador de indirección

Punteros: declaración e inicialización I

A tener en cuenta:

- la inicialización se hace indicando una variable a la que apuntar: `ptr = &x;`
- sólo pueden asignarse direcciones de memoria para las que haya coincidencia de tipos

Punteros: declaración e inicialización I

A tener en cuenta:

- puede declararse un puntero en la misma sentencia en que se declara otra variable: `int x, *ptr;`
- el único valor que puede asignarse directamente es 0 (puntero nulo) (también puede asignarse la constante `NULL`, definida en `cstdlib`)

Contenido del tema

Relación entre punteros y arrays I

- El nombre de un array (es un alias) representa la dirección de memoria en la que se ubica. Por tanto, tiene relación con un puntero

¿Cómo comprobarlo?

Relación entre punteros y arrays I

Ejemplo sencillo: el nombre del array sirve para acceder a su primer valor

```
1  #include <iostream>
2
3  using namespace std;
4
5  int main(){
6    short numeros[5]={10,20,30,40,50};
7
8    // Primer valor del array en primera posicion
9    // de almacenamiento
10   cout << "Primer valor: " << *numeros << endl;
11 }
```

No se declaran punteros, pero se usa *: el nombre del array devuelve una dirección de memoria (como un puntero)

Relación entre punteros y arrays I

El almacenamiento de arrays en posiciones sucesivas permite realizar recorridos de forma sencilla usando punteros:

```
// Primera posicion
*numeros
// Siguiente
*(numeros+1)
..............
```

Relación entre punteros y arrays I

La declaración de un puntero implica indicar el tipo de valores almacenados en las posiciones de memoria a apuntar: el compilador debe saber qué hacer con dicha dirección (cuántos bytes lee a partir de la misma.....)

```
// Implica saltar tantos bytes como indique
// el tipo del puntero
*(numeros+1)
..............
```

Importante: uso del paréntesis

Relación entre punteros y arrays I

Ejemplo: manejo de arrays con punteros

```
1  #include <iostream>
2
3  using namespace std;
4
5  int main(){
6     const int TAMANIO=10;
7     int numeros[TAMANIO];
8
9     // Se gestiona el array mediante punteros
10    cout << "Introduzca valores: (" << TAMANIO << "): " << endl;
11
12    // Bucle de lectura
13    for(int i=0; i < TAMANIO; i++){
14       cin >> *(numeros+i);
15    }
16
17    // Ahora se muestran
18    for(int i=0; i < TAMANIO; i++){
19       cout << *(numeros+i) << " ";
20    }
21    cout << endl;
22 }
```

Relación entre punteros y arrays I

Importante:

- C++ no hace comprobaciones sobre el acceso con punteros....
- si los arrays son punteros, también podemos usar [] con punteros

Relación entre punteros y arrays I

Uso de corchetes con punteros

```
1  #include <iostream>
2
3  using namespace std;
4
5  int main(){
6     const int TAMANIO=5;
7     double monedas[TAMANIO]={0.05, 0.1, 0.25, 0.5, 1.0};
8     double *ptrDouble;
9
10    // Se asigna al puntero la direccion del array
11    ptrDouble=monedas;
12
13    // Se muestra el contenido del array usando []
14    for(int i=0; i < TAMANIO; i++){
15       cout << ptrDouble[i] << " ";
16    }
17    cout << endl;
18
19    // Se vuelve a mostrar usando unicamente punteros
20    for(int i=0; i < TAMANIO; i++){
21       cout << *(ptrDouble+i) << " ";
22    }
23    cout << endl;
24 }
```

Relación entre punteros y arrays I

A observar:

```
int x, array[50], *ptrInt;
.............................
ptrInt = &x;
// Con array la asignacion es directa
ptrInt = array;
// Es valido?
ptrInt = &array[1];
```

Diferencia entre array y ptrInt: array no puede apuntar a otra dirección (array es un puntero constante)

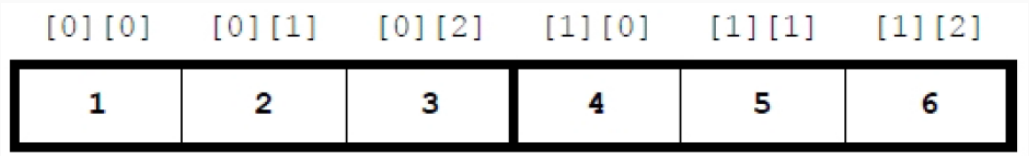
Relación entre punteros y arrays I

En memoria los datos se organizan linealmente:

- Los datos de cada fila se almacenan consecutivamente
- La fila i se almacena a continuación de la fila $i-1$

```
const int FIL = 2;
const int COL = 3;

int m[FIL][COL] = {{1,2,3},{4,5,6}};
```



Relación entre punteros y arrays I

```
int main(){
const int FIL = 2;
const int COL = 3;
....
int m[FIL][COL] = {{1,2,3},{4,5,6}};
....
//Puntero para acceder a cada casilla
int *p;
//primera casilla de m
p = &(m[0][0]);

for(int i=0; i<FIL*COL; i++){
cout << *(p+i) << " ";
}
}
```

Contenido del tema

Aritmética de punteros I

Con los punteros pueden realizarse algunas operaciones (ya hemos incrementado su valor para acceder a otra posición de memoria). Ejemplo:

```
1  #include <iostream>
2
3  using namespace std;
4
5  int main(){
6      const int TAMANIO=10;
7      int conjunto[TAMANIO]={5,10,15,20,25,30,35,40,45,50};
8      int *ptrInt;
9
10     // Se asigna al puntero la direccion del array
11     ptrInt=conjunto;
12
13     // Se muestra el contenido del array usando punteros
14     for(int i=0; i < TAMANIO; i++){
15         cout << *ptrInt << " ";
16
17         // Se avanza el puntero
18         ptrInt++;
19     }
20     cout << endl;
21
22
```

Aritmética de punteros II

```
23      // Se recorre ahora hacia atras: ptrInt ya ha quedado
24      // ubicado en la posicion siguiente al ultimo elemento
25      for(int i=0; i < TAMANIO; i++){
26          ptrInt--;
27          cout << *ptrInt << " ";
28      }
29      cout << endl;
30  }
```

Aritmética de punteros I

Los operadores ++ y - - se usan para incrementar y decrementar el valor de las variables tipo puntero. Son operadores de especial interés para recorrer arrays y matrices, donde los elementos se almacenan en posiciones sucesivas.

También tiene sentido usar comparadores relacionales para saber si una posición de memoria es mayor o menor que otra.

Operaciones relacionales con punteros I

Supongamos una variable array (de enteros): ¿cómo se evalúan las siguientes expresiones?

```
&array[1] > &array[0]
array < &array[4]
array == &array[0]
&array[2] != &array[3]
```

Operaciones relacionales con punteros I

Ejemplo de comparación de direcciones de memoria:

```
1  #include <iostream>
2
3  using namespace std;
4
5  int main(){
6      const int TAMANIO=10;
7      int datos[TAMANIO]={5,10,15,20,25,30,35,40,45,50};
8      int *ptrInt;
9
10     // Se asigna al puntero la direccion del array
11     ptrInt=datos;
12
13     // Bucle de recorrido de elementos
14     while(ptrInt < &datos[TAMANIO]){
15         // Se muestra el elemento
16         cout << *ptrInt << " ";
17
18         // Se avanza el puntero
19         ptrInt++;
20     }
21
22     // Se salta de linea
23     cout << endl;
24
```

Operaciones relacionales con punteros II

```
25      // Se recorre ahora hacia atras
26      while(ptrInt > datos){
27          // Se decrementa el puntero
28          ptrInt--;
29
30          // Se muestra el elemento
31          cout << *ptrInt << " ";
32      }
33      cout << endl;
34  }
```

Operaciones relacionales con punteros I

Suele ser habitual usar expresiones con comparaciones con el puntero nulo (para comprobar si contiene una dirección válida):

```
if(ptrInt != 0){
  cout << "Posicion correcta: " << *ptrInt << endl;
}
else{
  cout << "Posicion no valida " << endl;
}
```

Operaciones relacionales con punteros I

La condición también podría haberse escrito de la siguiente forma:

```
if(ptrInt){
  cout << "Posicion correcta: " << *ptrInt << endl;
}
else{
  cout << "Posicion no valida " << endl;
}
```

Contenido del tema

Punteros y funciones I

Las funciones pueden tener punteros como argumentos. Esto da acceso a la posición de almacenamiento del parámetro actual.

Muchas librerías de C usan paso de argumentos mediante punteros, especialmente en caso de programación de bajo nivel. Se consideran a continuación varios ejemplos de paso mediante punteros.

Punteros y funciones I

Ejemplo: paso de vector mediante punteros

```
1  #include <iostream>
2
3  using namespace std;
4
5
6  // Implementacion de las funciones
7
8
9  // Procedimiento para obtener valor mediante puntero
10 void obtenerNumeroPuntero(int *valor){
11     cout << "Introduzca valor entero: ";
12     cin >> *valor;
13 }
14
15
16 // Procedimiento para duplicar el valor mediante puntero
17 void doblarValorPuntero(int *valor){
18     *valor=*valor*2;
19 }
20
21
22
23
24
```

Punteros y funciones II

```
25 int main(){
26     int numero;
27 
28     // Se obtiene el valor de numero mediante la funcion
29     // con punteros
30     obtenerNumeroPuntero(&numero);
31 
32     // Se muestra el valor
33     cout << "Valor obtenido con metodo con punteros: " <<
34                                             numero << endl;
35 
36     // Se duplica su valor mediante metodo con punteros
37     doblarValorPuntero(&numero);
38 
39     // Se muestra el resultado
40     cout << "Valor duplicado con metodo con punteros: "
41                                             << numero << endl;
42 
43 }
```

Punteros y funciones I

También podemos usar punteros como argumento formal y arrays como parámetro actual.

Punteros y funciones I

Ejemplo: argumento formal tipo puntero y argumento actual tipo array

```
1  #include <iostream>
2
3  using namespace std;
4
5  // Declaracion anticipada de funciones
6  void leerNotas(double *, int);
7  double calcularValorMedia(double *, int);
8
9  // Metodo main para prueba
10 int main(){
11     const int TAMANIO=10;
12     double notas[TAMANIO];
13
14     // Se leen los valores
15     leerNotas(notas, TAMANIO);
16
17     // Se calcula la media
18     double media=calcularValorMedia(notas, TAMANIO);
19
20     // Se muestra el valor de la media
21     cout << "Nota media: " << media << endl;
22 }
23
24
```

Punteros y funciones II

```
25 // Implementacion de las funciones
26
27 // Proc. para leer los valores de las notas
28 void leerNotas(double *notas, int tam){
29     for(int i=0; i< tam; i++){
30         cout << "Introduzca valor entero: ";
31         cin >> notas[i]; // *(notas+i);
32     }
33 }
34
35 // Funcion que calcula la media
36 double calcularValorMedia(double *notas, int tam){
37     double media=0;
38
39     // Se calcula el valor
40     for(int i=0; i< tam; i++){
41         media=media+(*notas);
42         notas++;
43     }
44
45     // Se calcula la division
46     media=media/tam;
47
48     // Se devuelve el valor de la media
49     return media;
50 }
```

Punteros y funciones I

Otra alternativa de recibir un vector en una función:

```
1  #include <iostream>
2
3  using namespace std;
4
5  // Declaracion anticipada de funciones
6  void leerNotas(double [], int);
7  double calcularValorMedia(double [], int);
8
9  // Metodo main para prueba
10 int main(){
11     const int TAMANIO=5;
12     double notas[TAMANIO];
13
14     // Se leen los valores
15     leerNotas(notas, TAMANIO);
16
17     // Se calcula la media
18     double media=calcularValorMedia(notas, TAMANIO);
19
20     // Se muestra el valor de la media
21     cout << "Nota media: " << media << endl;
22 }
23
24
```

Punteros y funciones II

```
25 // Implementacion de las funciones
26
27 //Proc. para leer los valores de las notas
28 void leerNotas(double notas[], int tam){
29     for(int i=0; i< tam; i++){
30         cout << "Introduzca valor entero: ";
31         cin >> notas[i];
32     }
33 }
34
35 // Funcion que calcula la media
36 double calcularValorMedia(double notas[], int tam){
37     double media=0;
38
39     // Se calcula el valor
40     for(int i=0; i< tam; i++){
41         media=media+(*notas);
42         notas++;
43     }
44
45     // Se calcula la division
46     media=media/tam;
47
48     // Se devuelve el valor de la media
49     return media;
50 }
```

Punteros y funciones I

Paso de matrices a funciones:

Cuando se pasa una matriz a una función, se escribe el nombre de la matriz. Estamos proporcionando la dirección de memoria del primer elemento de la matriz.

La función que recibe una matriz puede hacerlo de dos maneras:

Punteros y funciones I

a) Notación clásica (con corchetes):

```
const int FIL=10;
const int COL=20;

int matriz[FIL][COL];
...

function(matriz,FIL,COL);
...

void funcion(int m[][COL], int FIL, int COL){
...
}
```

Punteros y funciones II

- Necesario indicar el número de columnas, para que el compilador sepa cómo calcular la posición de un elemento.
- El número de filas es irrelevante.

Punteros y funciones I

Ejemplo de paso de matriz en una función:

```
1  #include <iostream>
2
3  using namespace std;
4
5  // Declaracion de constantes
6  const int FILS = 10;
7  const int COLS = 10;
8
9
10 int Maximo (int m[][COLS], int filas, int cols)    {
11
12     int maximo = -1;
13
14     for (int f=0; f<filas; f++)
15         for (int c=0; c<cols; c++)
16             if (m[f][c] > maximo)
17                 maximo = m[f][c];
18
19     return (maximo);
20 }
21
22
23 // Metodo main para prueba
24
```

Punteros y funciones II

```
25 int main(){
26
27 int valores [FILS][COLS] = {{33, 22, 11, 44, 55},
28                            { 2, 55, 66, 35, 6},
29                            {23, 99, 4, 34, 12}};
30 int filas_utilizadas = 3;
31 int cols_utilizadas = 5;
32
33 cout << Maximo (valores, filas_utilizadas, cols_utilizadas);
34
35 }
```

Punteros y funciones I

b) Con un puntero a la primera fila: uso no frecuente

Se trata de que la función gestione la matriz como si fuera un vector de vectores.

```
void funcion(int (*m)[COL], int filas, int columnas){
...
}
```

- Necesario indicar el número de columnas reservadas [*COL*], que componen cada fila de la matriz.
- El número de columnas realmente ocupadas (*columnas*) puede ser menor que el de las reservadas.
- El número de filas ocupadas (*filas*) no es obligatorio (necesario para evitar acceder a posiciones no válidas).

Punteros y funciones I

La función máximo quedaría:

```
1  #include <iostream>
2
3  using namespace std;
4
5  // Declaracion de constantes
6  const int FILS = 10;
7  const int COLS = 10;
8
9
10 int Maximo (int (* m) [COLS], int filas, int cols){
11     int maximo = -1;
12
13     for (int f=0; f<filas; f++) {
14         for (int c=0; c<cols; c++)
15             if ((* m)[c] > maximo) maximo = (* m)[c];
16
17         m++; // "m" referencia a la siguiente fila
18     }
19     return (maximo);
20 }
```

Contenido del tema

Punteros a constantes y punteros constantes I

En un puntero se asocian dos datos:

- dirección de memoria que almacena
- valor almacenado en dicha posición

Podemos usar **const** para preservar alguno de ellos (o ambos):

- puntero a valor constante: no podrá modificarse el valor
- puntero constante: no podrá modificarse la dirección de memoria (no podrá apuntar a otra dirección)
- todo constante

Punteros a constantes y punteros constantes I

Si un dato es constante, el puntero asociado debe respetar esta condición. Ejemplo: array constante y puntero asociado indicando que los valores almacenados son constantes

```
const int SIZE=10;
const double tasas[SIZE]={18.0, 21.0, 24.5, 27.3};
```

Si deseamos pasar este array a una función mediante punteros, habrá que declararlo constante:

```
void mostrarTasas(const double *valoresTasas, int util){
.....................................
}
```

Punteros a constantes y punteros constantes I

El tipo de valoresTasas es puntero a const double:

```
// tipo del contenido        puntero
const double                 * valoresTasas
```

Si no se incluye const en la declaración de la función mostrarTasas se obtiene un error de compilación.

Punteros a constantes y punteros constantes I

Un método que espera recibir como argumento un puntero a constante también puede usarse con parámetros actuales del mismo tipo, aunque sin const:

```
void mostrarValores(const int*, int);
.....................................
const int TAM=10;
const int array1[TAM]={1,2,3,4,5,6};
int array2[TAM]={2,4,6,8,10};
mostrarValores(array1,6);
mostrarValores(array2,5);
```

Ya usado antes.... (métodos de impresión, etc)

Punteros a constantes y punteros constantes I

Como hemos indicado antes, al manejar punteros hay dos aspectos a considerar:

- contenido de la variable puntero (dirección de memoria)
- contenido de la posición de memoria

Estos dos elementos pueden estar afectados por const. En el ejemplo anterior

```
// tipo del contenido        puntero
const double                 * valoresTasas
```

lo constante es el valor.

Punteros a constantes y punteros constantes I

En el ejemplo siguiente lo constante es el puntero:

```
// tipo del contenido     puntero
double                  * const valoresTasas
```

Punteros a constantes y punteros constantes I

Y si todo es constante:

```
// tipo del contenido      puntero
const double              * const tasas
```

Contenido del tema

Punteros a punteros I

También podemos tener lo siguiente:

```
int **ptr;
```

¿Qué es ptr? Una variable que almacena una dirección de memoria donde se almacena (a su vez) otra dirección de memoria.

Punteros a punteros I

Imaginemos la siguiente situación:

```
int x;
int *ptr=&x;
int **ptrptr=&ptr;
//Como cambiar el valor de x?
*ptr=23;
*(*ptrptr)=34;
```

¿Qué es ptrptr? Una variable que almacena una dirección de memoria donde se almacena (a su vez) otra dirección de memoria.

Contenido del tema

Struct y punteros I

Un puntero puede contener la dirección de un dato de tipo struct.

Los operadores **&** y ***** funcionan igual que para otro tipo de dato.

```
struct Persona{
int edad;
double estatura;
};

Persona pepe;
Persona *ptr;
pepe.edad=27;
pepe.estatura=1.89;
ptr = &pepe;
cout << (*ptr).edad << endl; //Imprime 27
```

Struct y punteros I

La asignación entre punteros funciona igual cuando trabajamos con punteros a *struct*.

```
struct Persona{
int edad;
double estatura;
};
Persona pepe;
Persona *ptr, *ptr2;
pepe.edad=27;
pepe.estatura=1.89;
ptr = &pepe;
ptr2 = ptr;
cout << (*ptr).edad << endl;  //Imprime 27
cout << (*ptr2).edad << endl; //Imprime 27
```

Struct y punteros I

La expresión *(*puntero).campo* se simplifica con el operador $\rightarrow$, quedando más claro *puntero* $\rightarrow$ *campo*.

```
struct Persona{
int edad;
double estatura;
};
Persona pepe;
Persona *ptr, *ptr2;
pepe.edad=27;
pepe.estatura=1.89;
ptr = &pepe;
ptr2 = ptr;
cout << ptr -> edad << endl;  //Imprime 27
cout << ptr2 -> edad << endl; //Imprime 27
```

Struct y punteros I

Un *struct* puede tener campos de tipo puntero.

La asignación entre dos *struct* se hace campo a campo. Si un campo es un puntero, se realiza una asignación entre punteros (los dos punteros apuntan al mismo dato).

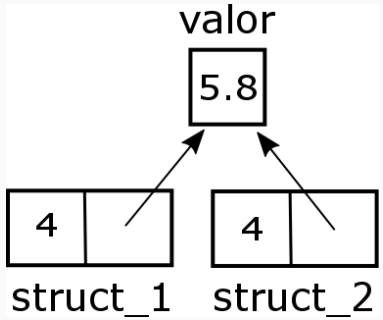
```
double valor = 5.8;

struct TipoStruct{
int dato;
double *puntero;
};
TipoStruct struct_1, struct_2;

struct_1.dato = 4;
struct_1.puntero = &valor;
```

Struct y punteros II

```
struct_2 = struct_1;
cout << *(struct_1.puntero); //imprime 5.8
cout << *(struct_2.puntero); //imprime 5.8
```

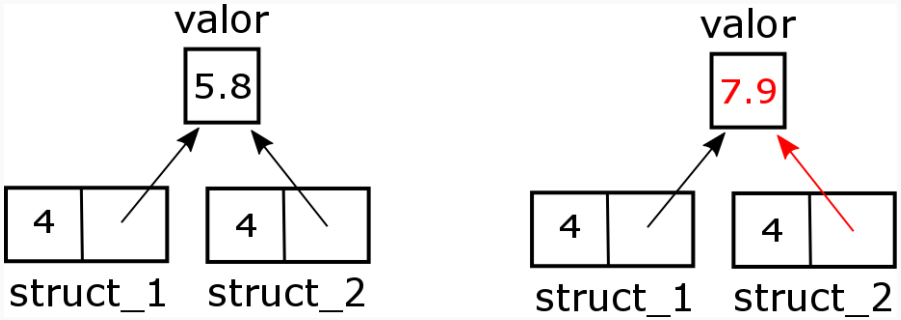


Struct y punteros I

Si escribimos:

```
*(struct_2.puntero) = 7.9;
```

modificamos el valor de la variable *valor*.



Esta instrucción provoca un efecto colateral en *struct_1*, el objeto referenciado por su puntero se ha modificado.

Contenido del tema

Cadenas y punteros I

Una cadena clásica de caracteres (estilo C) no es más que un vector de caracteres que contiene un carácter especial, \0 (caracter nulo, código ASCII 0) que indica el fin de la cadena.

- Toda cadena clásica es un vector de caracteres
- Un vector de caracteres no tiene porqué ser una cadena clásica. Para ello debe contener el carácter \0, que se consigue:
  - Inicializando la cadena de manera adecuada
  - Con funciones que añaden el delimitador de fin de cadena
    1. Automáticamente, utilizando funciones de lectura o gestión de cadenas (biblioteca cstring)
    2. Manualmente, utilizando funciones propias

Inicialización de cadenas de caracteres I

a) Notación de corchetes

```
char saludo1[5] = {'H','o','l','a','\0'};
char saludo1[10] = {'H','o','l','a','\0'};
char saludo1[] = {'H','o','l','a','\0'};
```

- Puede indicarse el tamaño exacto
- Puede reservarse más de lo necesario
- Puede omitirse el tamaño (se reserva lo necesario)

En estos casos es posible modificar la cadena posteriormente.

Inicialización de cadenas de caracteres I

b) Notación de punteros

Literales de cadena (su tipo es const char *).

- Cualquier función que tenga un parámetro formal de ese tipo puede recibir un literal de cadena de caracteres y procesar esa cadena mediante un puntero.
- Evidentemente no se podrá modificar su contenido.

Esta inicialización

```
const char *saludo = "Hola";
```

- Copia la dirección de memoria de la constante literal cuyo valor es *Hola* en el puntero *saludo*.
- Constante literal tiene 5 caracteres (incluido el delimitador de fin de cadena).
- **No es posible modificar la cadena**.

```
saludo[1] = 'a'; //error de compilacion
```

Funciones de lectura/escritura I

El operador « está sobrecargado para cadenas de caracteres clásicas.

```
#include <iostream>
using namespace std;

int main(){

char cadena1[5] = {'H','o','l','a','\0'};
char cadena2[10] = {'H','o','l','a','\0'};
char cadena3[] = {'H','o','l','a','\0'};

cout << cadena1 << " " << cadena2 << " " << cadena3 << endl;

const char * cadena4 = "Hello";
cout << cadena4 << endl;
}
```

Funciones de lectura/escritura I

Para lectura de cadenas: función getline(char *p, int n)

- Lee como mucho $n - 1$ caracteres de la entrada estándar.
- Caracteres leidos se copian por orden a partir de la dirección de memoria guardada en p.
- Reservar suficiente memoria.

```
1  #include <iostream>
2
3  using namespace std;
4
5  int main(){
6
7      const int TAM = 80;
8      char cadena[TAM];
9
10     cout << "Nombre: ";
11     cin.getline (cadena, TAM);
12     cout << "Su nombre: "<< cadena << endl;
13
14 }
```

Funciones de gestión de cadenas I

La biblioteca cstring ofrece funciones para realizar tareas comunes:

- strcpy: copia una cadena en otra
- strlen: determina la longitud, sin incluir la marca final en el conteo
- strcat: concatenación de cadenas
- strcmp: comparación de cadenas en orden lexicográfico

Funciones de gestión de cadenas I

```
#include <iostream>
#include <cstring>
using namespace std;

void MuestraDatosCadena (const char * msg, const char * c){
cout << msg << ": " << c << strlen(c) << " caracteres." << endl;
}

int main(){

char saludo[20] = {'H','o','l','a','\0'};
// Hay 15 casillas no usadas en "saludo"
const char * nombre = "Rosa";
const char * espacio = " ";
char cadena[100]; // No inicializada (no contiene '\0')
```

Funciones de gestión de cadenas II

```
strcpy (cadena, saludo); // cadena contiene "Hola" (copia el '\0')
strcat (cadena, ","); // "Hola,"
strcat (cadena, espacio); // "Hola, "
strcat (cadena, nombre); // "Hola, Rosa"

MuestraDatosCadena ("Saludo", saludo);
MuestraDatosCadena ("Nombre", nombre);
MuestraDatosCadena ("Cadena compuesta", cadena);
}
```

Contenido del tema

Vectores de punteros I

Un vector:

- Almacena una colección de datos en posiciones consecutivas de memoria
- Se accede a cada elemento mediante un índice, utilizando el operador []

Los datos de la colección, ¿pueden ser punteros?

Vectores de punteros I

Declaración: indicar el tipo de datos de sus componentes y el tamaño del vector.

Ejemplo:

```
const int TAM=5;
int *vectorPtr[TAM];
```

Vectores de punteros I

Ejemplo:

```
int *vectorPtr[4];
int a=5, b=7, c=3, d=2;

vectorPtr[0] = &a;
vectorPtr[1] = &b;
vectorPtr[2] = &c;
vectorPtr[3] = &d;

for(int i=0; i<4; i++){
   cout << *vectorPtr[i] << " ";
}
cout << endl;
```

Vectores de punteros I

Ejemplo: vector de punteros a enteros

```
1  #include <iostream>
2  using namespace std;
3
4  void ordenacionPorSeleccion(const int * v[], int util){
5     int pos_min;
6     const int *aux;
7
8     for (int i=0; i<util-1; i++){
9        pos_min=i;
10       for (int j=i+1; j<util; j++)
11          if (*v[j] < *v[pos_min])
12             pos_min=j;
13
14       aux = v[i];
15       v[i] = v[pos_min];
16       v[pos_min] = aux;
17    }
18 }
19
20 int main(){
21    const int TAM=100;
22    const int* vectorPunts[TAM];
23    const int vectorInts[TAM]={5,7,3,2};
24    int util=4;
```

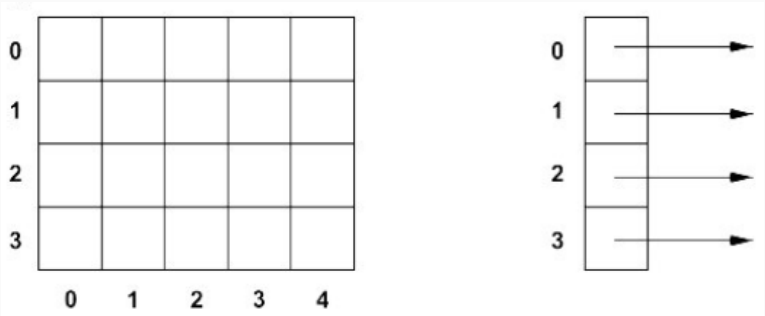
Vectores de punteros II

```
25
26      for(int i=0; i< util; i++){
27          vectorPunts[i] = &vectorInts[i];
28      }
29
30      cout<<"Array antes de ordenar (impreso con vectorPunts):"<<endl;
31      for(int i=0; i< util; i++){
32          cout << *vectorPunts[i] << " ";
33      }
34      cout << endl;
35
36      ordenacionPorSeleccion(vectorPunts,util);
37
38      cout<<"Array despues de ordenar (impreso con vectorPunts):"<<endl;
39      for(int i=0; i< util; i++){
40          cout << *vectorPunts[i] << " ";
41      }
42      cout << endl;
43
44      cout<<"Array despues de ordenar (impreso con vectorInts):"<<endl;
45      for(int i=0; i< util; i++){
46          cout << vectorInts[i] << " ";
47      }
48      cout << endl;
49  }
```

Vectores de punteros I

Vector de punteros y matriz bidimensional

```
int matriz[4][5];
int *vector[4];
```



Las expresiones: matriz[3][4]; vector[3][4];

Son expresiones sintácticamente correctas.

OJO!! vector[3][4], esta zona de memoria no está reservada (no se ha inicializado).

Vectores de punteros I

Es necesario inicializar las zonas de memoria referenciadas por vector[i]:

- Tiempo de compilación: cada puntero del vector referencia a una zona de memoria reservada previamente, por ejemplo, otro vector o una fila de una matriz.
- Tiempo de ejecucición: memoria dinámica (más adelante).

Vectores de punteros I

Vector de punteros a char:

```
char * meses[]={"enero","febrero","marzo"};
```

- Cada componente se inicializa con la dirección de memoria del literal especificado.
- Es posible modificar los punteros, pero no es posible modificar el contenido referenciado.

Con una matriz de char:

```
char meses[][8]={"enero","febrero","marzo"};
```

El compilador necesita conocer el número de columnas (indicada por cadena más larga +1, del \0).

Vectores de punteros I

- Ventaja de usar vectores de punteros frente a matrices: cada línea puede tener un número diferente de columnas. Optimización de memoria.
- Un vector de punteros se comporta como una matriz, ya que se puede acceder a sus componentes mediante índices.

Contenido del tema

Punteros a funciones I

Importante: el nombre de una función es como una variable, que permite recordar de forma sencilla la dirección de memoria en que se almacena su código.

Por tanto podemos tener un puntero a dicho espacio de memoria: puntero a función.

Punteros a funciones I

Ejemplo de uso:

```
// Supongamos la existencia de los metodos
bool comprobarOrdenAscendente(int, int)
bool comprobarOrdenDescendente(int, int)
```

Haciendo uso de ellos pretendemos desarrollar un método genérico de ordenación, que pueda hacer tanto ordenación ascendente como descendente. El método se basa en trabajar con alguna de las dos funciones vistas antes, según se necesite.

Punteros a funciones I

Lo más general es implementar un único método general:

```
void ordenar(int [], const int, bool (*)(int, int));
```

donde los argumentos se refieren a :

- array a ordenar
- elementos del array a ordenar
- puntero a función

Punteros a funciones I

El código completo es el siguiente:

```
1  #include <iostream>
2
3  using namespace std;
4
5  // Funcion para comprobar ordenacion ascendente
6  bool comprobarOrdenAscendente(int a, int b){
7     return(a < b);
8  }
9
10 // Funcion para comprobar ordenacion descendente
11 bool comprobarOrdenDescendente(int a, int b){
12    return(a > b);
13 }
14
15 // Procedimiento para intercambiar los valores de dos
16 // variables pasadas mediante punteros
17 void cambiar(int *const ptr1, int * const ptr2){
18   int aux=*ptr1;
19   *ptr1=*ptr2;
20   *ptr2=aux;
21 }
22
23
24
```

Punteros a funciones II

```
25 // Procedimiento general de ordenacion
26 void ordenar(int array[], const int util,
27               bool (*comparacion)(int,int)){
28   int indice;
29
30   // Bucle de recorrido de valores
31   for(int i=0; i < util-1; i++){
32     indice=i;
33     for(int j=i+1; j < util; j++){
34       if (!(*comparacion)(array[indice],array[j])){
35          indice=j;
36       }
37     }
38
39     // Se intercambian los valores
40     cambiar(&array[indice], &array[i]);
41   }
42 }
43
44
45 // Funcion main para probar
46 int main(){
47    int datos[]={1,2,3,5,1,7,9,2,1,3,10,11,4,8,6};
48
49    // Se hace la ordenacion ascendente
50    ordenar(datos, 15, comprobarOrdenAscendente);
51
```

Punteros a funciones III

```
52      // Se muestra el resultado
53      for(int i=0; i < 15; i++){
54          cout << " " << datos[i];
55      }
56      cout << endl;
57
58      // Se hace la ordenacion descendente
59      ordenar(datos, 15, comprobarOrdenDescendente);
60
61      // Se muestra el resultado
62      for(int i=0; i < 15; i++){
63          cout << " " << datos[i];
64      }
65      cout << endl;
66  }
```

Parte II

Gestión Dinámica de Memoria

Contenido del tema

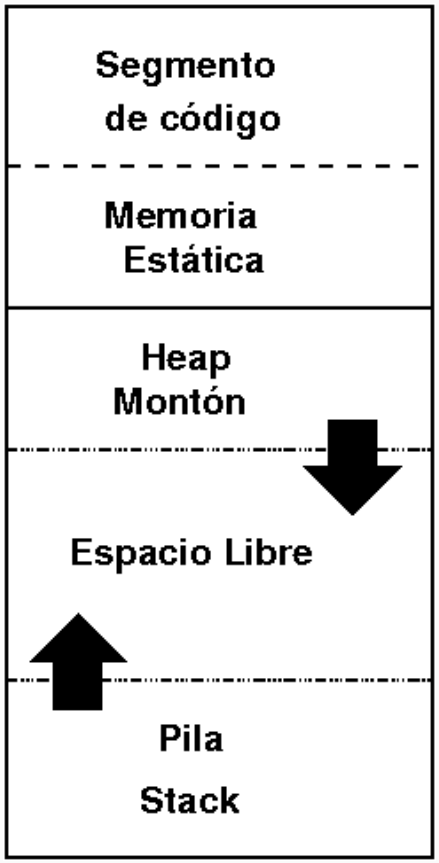
Estructura de la memoria I

Gracias a la gestión de memoria del Sistema Operativo, los programas tienen una visión más simplificada del uso de la memoria, la cual ofrece una serie de componentes bien definidos.

Segmento de código
Es la parte de la memoria asociada a un programa que contiene las instrucciones ejecutables del mismo.

Memoria estática

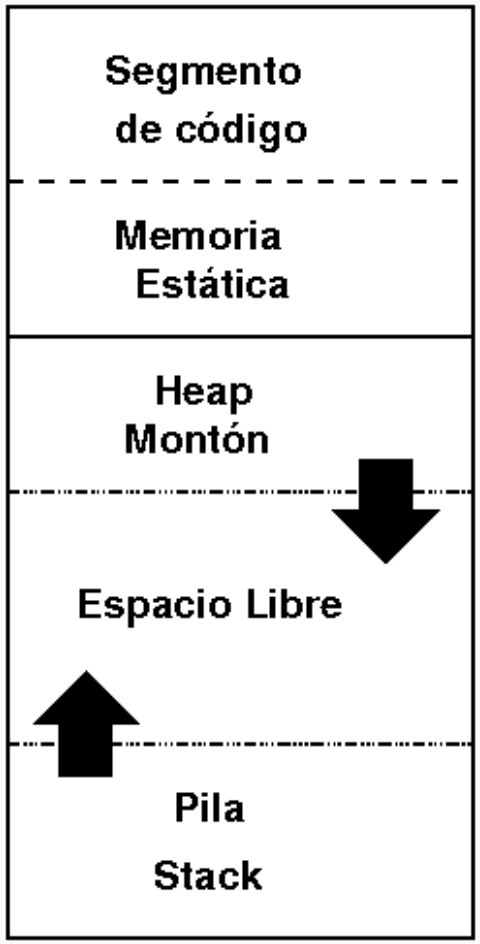
- Reserva antes de la ejecución del programa
- Permanece fija
- No requiere gestión durante la ejecución
- El sistema operativo se encarga de la reserva, recuperación y reutilización.
- Variables globales y `static`.



Estructura de la memoria I

La pila (Stack)

- Es una zona de memoria que gestiona las llamadas a funciones durante la ejecución de un programa.
- Cada vez que se realiza una llamada a una función en el programa, se crea un **entorno de programa**, que se libera cuando acaba su ejecución.
- La reserva y liberación de la memoria la realiza el S.O. de forma automática durante la ejecución del programa.
- Las variables locales no son variables estáticas. Son un tipo especial de variables dinámicas, conocidas como **variables automáticas**.

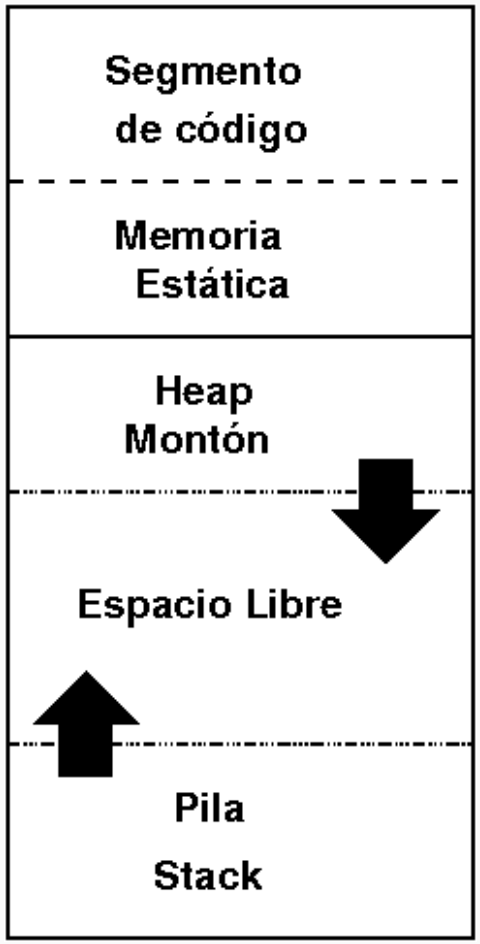


Estructura de la memoria I

El montón (Heap)

- Es una zona de memoria donde se reservan y se liberan "trozos" durante la ejecución de los programas según sus propias necesidades.
- Esta memoria surge de la necesidad de los programas de "crear nuevas variables" en tiempo de ejecución con el fin de optimizar el almacenamiento de datos.

El montón y la pila comparten la memoría dinámica.



Contenido del tema

Gestión dinámica de memoria I

Es posible crear y destruir variables de forma explícita durante la ejecución de un programa. Esto permitirá crear (y destruir) arrays a medida, justo con el espacio de memoria necesario. Así evitaremos el problema del sobredimensionado.

Esta técnica se conoce como gestión dinámica de memoria y precisa del uso de punteros.

Gestión dinámica de memoria I

La reserva de espacio de memoria se realiza mediante el operador new. Este operador devuelve la dirección de memoria donde se hizo la reserva:

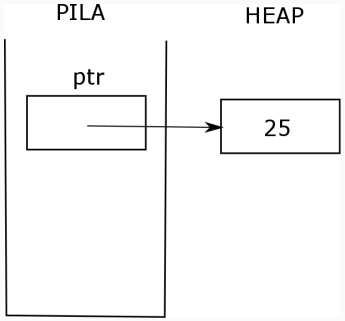
```
int *ptrInt;

// Se reserva espacio para un entero:
// SOLO COMO EJEMPLO...
// Se reservan 4 bytes y new devuelve
// la direccion de comienzo del bloque
// de 4 bytes
ptrInt = new int;
```

Gestión dinámica de memoria I

Una vez reservada la memoria es posible almacenar valores en ella:

```
*ptrInt=25;
cout << "Almacenado valor: " << *ptrInt;
cout << "Introduzca nuevo valor: ";
cin >> *ptrInt;
.....................................
```



Gestión dinámica de memoria I

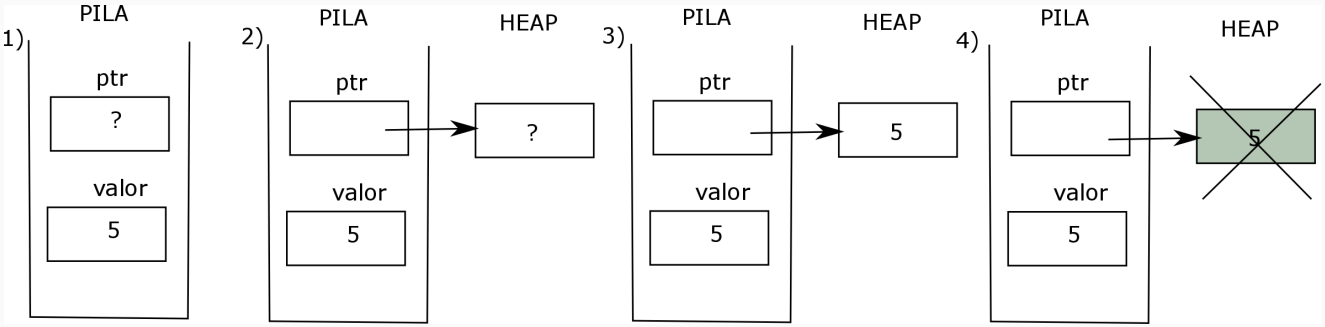
El trozo de memoria reservado pertenece a una zona especial llamada montón (heap) (que tiene un tamaño limitado y podría agotarse y no completarse la operación).

Si no tiene éxito la operación se genera excepción de tipo bad_alloc, que implica la finalización del programa.

Gestión dinámica de memoria I

Gestión dinámica: funcionalidad muy potente, pero debe ser usada de forma cuidadosa: el espacio reservado debe ser liberado de forma explícita (en caso de no hacerlo se irá perdiendo memoria....). El operador encargado de realizar la liberación de memoria es delete.

```
1) int *ptr, valor = 5;
2) ptr = new int;
3) *ptr = valor;
.......
4) delete ptr;  // Se libera espacio de una variable
```



Gestión dinámica de memoria I

Problemas del uso de memoria:

- intento de uso de memoria liberada y puesta de nuevo a disposición del montón
- no liberación de la memoria (pérdida de memoria), por lo que no puede accederse de nuevo a ella, ni reaprovecharla

Contenido del tema

Devolución de punteros en funciones I

Al hacer uso de memoria dinámica es posible reservar espacio de memoria en el cuerpo de una función o método y devolver un puntero a este espacio. Fuera de la función podrá seguir usándose.

Problema: debemos ser conscientes que en algún momento habrá que liberar ese espacio.

Devolución de punteros en funciones I

Pensemos en la validez (o no) del siguiente fragmento de código:

```
char *obtenerNombre(){
  char nombre[80];
  cout << "Introduzca nombre: ";
  cin.getline(nombre,80);
  return nombre;
}
```

Devolución de punteros en funciones I

A observar:

- nombre es una variable local
- al finalizar la función se libera el espacio de todas las variables locales (residentes en la pila)

Así que el código anterior no es correcto.

Devolución de punteros en funciones I

Sin embargo, el espacio de memoria reservado con new reside en el montón y no se destruye al finalizar la función.

```
char *obtenerNombre(){
  char *nombre=new char[80];
  cout << "Introduzca nombre: ";
  cin.getline(nombre,80);
  return nombre;
}
```

Problema: recordar posteriormente que hemos de liberar....

Devolución de punteros en funciones I

Situaciones válidas:

- devolución de puntero a un dato pasado como argumento
- puntero a posición de memoria reservada en función con new

Devolución de punteros en funciones I

Ejemplo de devolución de puntero a array creado en función:

```
1  #include <iostream>
2  #include <cstdlib>
3  #include <ctime>
4  using namespace std;
5
6  int *generarNumerosAleatorios(int numeros){
7      int *array=0;
8
9      // Se controla el valor de numeros
10     if (numeros > 0) {
11         // Se reserva espacio de memoria
12         array=new int[numeros];
13
14         // Se genera semilla con fecha actual
15         srand(time(0));
16
17         // Bucle de generacion
18         for(int i=0; i < numeros; i++){
19             array[i]=rand();
20         }
21     }
22     return array;
23 }
24
```

Devolución de punteros en funciones II

```
25 // Metodo main para prueba
26 int main(int argc, char *argv[]){
27     // Variable para guardar numero de valores a generar
28     int muestras=atoi(argv[1]);
29     cout << "Muestras: " << muestras << endl;
30     // Puntero a enteros
31     int *numerosAleatorios;
32 
33     // Se genera array de numeros aleatorios
34     numerosAleatorios=generarNumerosAleatorios(muestras);
35 
36     // Si pudo hacerse la reserva
37     if (numerosAleatorios){
38         // Se muestran los valores generados
39         for(int i=0; i < muestras; i++){
40             cout << numerosAleatorios[i] << " ";
41         }
42         cout << endl;
43 
44         // Se libera la memoria reservada
45         delete [] numerosAleatorios;
46     }
47 }
```

Contenido del tema

Vectores dinámicos I

- Hasta ahora sólo podíamos crear un array conociendo *a priori* el número máximo de elementos que podría llegar a tener
  P.e. int vector[20];
- Esa memoria está ocupada durante la ejecución del módulo en el que se realiza la declaración.
- Para reservar la memoria estrictamente necesaria: Operador new []

```
new <tipo_dato> [tam];
```

- Reserva en el Heap una zona de memoria para almacenar *tam* datos de tipo *tipo_dato*
- Devuelve la dirección de memoria del bloque de memoria reservado

Vectores dinámicos II

```
int tam;
cout << "Introduzca numero de valores a usar: "
cin >> tam;

// Se reserva a medida
int *ptrArray=new int[tam];

// Uso normal del array a traves de ptrArray
ptrArray[0]=3;
.....................................
```

Gestión dinámica de memoria I

La liberación se realiza con el operador delete []

```
delete [] ptrArray;
```

Libera (lo marca como disponible) la zona de memoria reservada con new.

Con la utilización de esta forma de reserva dinámica podemos crear vectores que tengan justo el tamaño necesario. Podemos, además, crearlo justo en el momento en el que lo necesitamos y destruirlo cuando deje de ser útil.

No puede aumentar ni disminuir el número de casillas reservadas para un vector dinámico.

Gestión dinámica de memoria I

Ejemplo de uso de gestión de memoria:

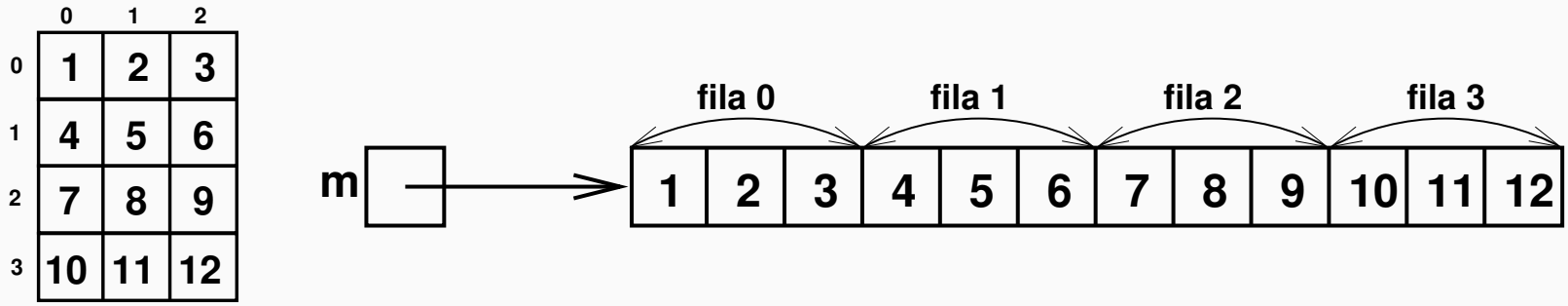
```
1  #include <iostream>
2
3  using namespace std;
4
5  // Ejemplo de gestion dinamica de memoria
6
7  // Metodo main para prueba
8  int main(){
9      double *ventas;
10     double total=0;
11     double media;
12     int numeroDias;
13
14     // Se pregunta al usuario el numero de ventas
15     // a manejar
16     cout << "Numero de ventas: ";
17     cin >> numeroDias;
18
19     // Se crea array para ese tamaño
20     ventas=new double[numeroDias];
21
22
23
24
```

Gestión dinámica de memoria II

```
25      // Se preguntan los datos al usuario
26      for (int i=0; i < numeroDias; i++) {
27          cout << "Ventas para dia (" << i << ") : ";
28          cin >> *(ventas+i);
29      }
30
31      // Se calcula la media
32      for(int i=0; i < numeroDias; i++){
33          total+=*(ventas+i);
34      }
35
36      // Ahora se divide por el numero de dias
37      media=total/numeroDias;
38
39      // Se muestra el resultado
40      cout << "Ventas medias: " << media << endl;
41
42      // Se libera el espacio
43      delete [] ventas;
44      ventas=0;
45  }
```

Contenido del tema

Matriz 2D usando un array 1D I



- **Creación de la matriz:**

```
int *matriz;
int fil, col;
matriz = new int [fil*col];
```

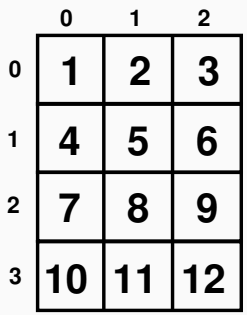
- **Acceso al elemento {f,c}:**

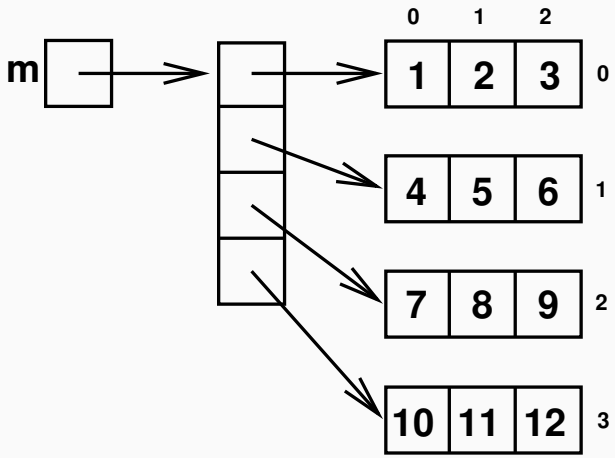
```
int valor;
valor = matriz[f*col+c]
```

- **Liberación de la matriz:**

```
delete [] matriz;
```

Matriz 2D usando un array 1D de punteros a arrays 1D I





- **Creación de la matriz:**

```
int **matriz;
int fil, col;
matriz = new int *[fil];
for(int i=0; i < fil; i++){
matriz[i] = new int[col];
}
```

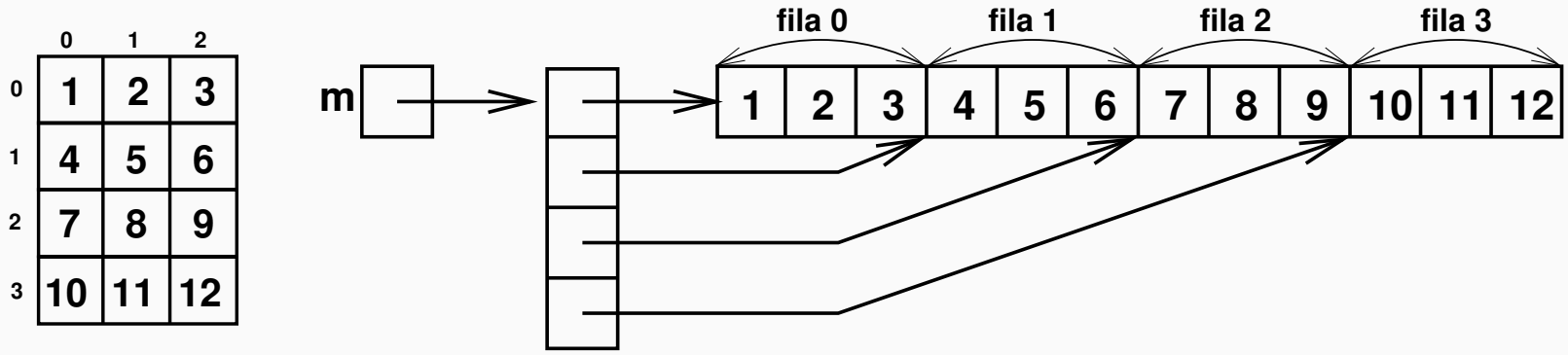
- **Acceso al elemento {f,c}:**

```
int valor;
valor = matriz[f][c];
```

- **Liberación de la matriz:**

```
for(int i=0; i < fil; i++){
delete [] matriz[i];
}
delete [] matriz;
```

Matriz 2D usando un array 1D de punteros a un único array I



- **Creación de la matriz:**

```
int **matriz;
int fil, col;
matriz = new int *[fil];
matriz[0] = new int[fil*col];
for(int i=1; i < fil; i++){
matriz[i] = matriz[i-1]+col;
}
```

- **Acceso al elemento {f,c}:**

```
int valor;
valor = matriz[f][c];
```

- **Liberación de la matriz:**

```
delete [] matriz[0];
delete [] matriz;
```

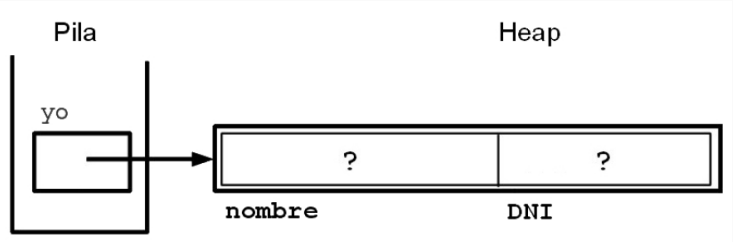
Contenido del tema

Objetos dinámicos compuestos I

Datos heterogéneos compuestos struct que son referenciados por punteros y a su vez contienen punteros. Se alojan en el Heap (montón).

En el caso de los struct, la instrucción new reserva memoria para cada uno de los campos de la estructura.

```
struct TipoPersona{
string nombre;
string DNI;
}
TipoPersona *yo;
yo = new TipoPersona;
```



Objetos dinámicos compuestos I

Para asignar valores a los campos de la estructura y mostrar su contenido:

```
//Lectura
cout << "Nombre";
getline(cin,yo->nombre);
cout << "DNI";
getline(cin,yo->dni);
.........................
//Mostrar valores
cout << "Datos de la estructura:" << endl;
cout << "Nombre: " << yo->nombre << end;
cout << "DNI: " << yo->dni << endl;

//Liberamos memoria
delete yo;
```