Metodología de la Programación

Tema 2. Punteros y memoria dinámica

Departamento de Ciencias de la Computación e I.A.

Curso 2015-16

¡Esta es una licencia de Cultura Libre!

Índice

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

Índice

Objetivos

Objetivos:

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

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

Índice

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: " << long (&letra) << endl;
12    cout << "Dir. mem. numero: " << long (&numero) << endl;
13    cout << "Dir. mem. temperatura: " << long (&temperatura) << endl;
14 }
```

Punteros: declaración e inicialización I

La salida obtenida es:

```
Dir. mem. letra: 140730899979119
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): " << long (&x) << endl;
17    cout << "Dir. mem. x (puntero): " << long (ptr) << endl;
18 }
```

Punteros: declaración e inicialización I

Salida:

```
Valor de x (habitual): 25
Valor de x (puntero): 25
Dir. mem. x (directa): 140734650180948
Dir. mem. x (puntero): 140734650180948
```

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

```
// 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`)

Índice

Relación entre punteros y arrays I

Ya comentado algo en tema anterior:

- el nombre de un array representa (es un alias) de la dirección de memoria en que se ubica. Por tanto, es 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[]={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 es en realidad 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

Ejemplo adicional:

Índice

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.

Aritmética de 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]
```

Aritmética de 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
```

Aritmética de 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  }
```

Aritmética de 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;
}
```

Aritmética de 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;
}
```

Índice

Punteros y funciones I

Las funciones pueden tener punteros como argumentos. Esto da acceso a la posición de almacenamiento del parámetro actual (como ocurría en el paso de arrays y en el paso por referencia).

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 por referencia y mediante punteros.

Punteros y funciones I

Ejemplo: paso por referencia y mediante punteros

```
1  #include <iostream>
2
3  using namespace std;
4
5  // Declaracion anticipada de funciones
6  void obtenerNumeroPuntero(int *);
7  void doblarValorPuntero(int *);
8  void obtenerNumeroReferencia(int &);
9  void doblarValorReferencia(int &);
10
11 int main(){
12     int numero;
13
14     // Se obtiene el valor de numero mediante la funcion
15     // con punteros
16     obtenerNumeroPuntero(&numero);
17
18     // Se muestra el valor
19     cout << "Valor obtenido con metodo con punteros: " <<
20                                             numero << endl;
21
22     // Se duplica su valor mediante metodo con punteros
23     doblarValorPuntero(&numero);
24
```

Punteros y funciones II

```
25      // Se muestra el resultado
26      cout << "Valor duplicado con metodo con punteros: "
27                                              << numero << endl;
28
29      // Se obtiene el valor de numero mediante referencia
30      obtenerNumeroReferencia(numero);
31
32      // Se muestra el valor
33      cout << "Valor obtenido con metodo con referencia: "
34                                              << numero << endl;
35
36      // Se duplica el valor mediante referencia
37      doblarValorReferencia(numero);
38
39      // Se muestra el resultado final
40      cout << "Valor duplicado con metodo con referencia: "
41                                              << numero << endl;
42  }
43
44
45
46
47
48
49
50
51
```

Punteros y funciones III

```
52 // Implementacion de las funciones
53
54 // Funcion para obtener valor mediante puntero
55 void obtenerNumeroPuntero(int *valor){
56     cout << "Introduzca valor entero: ";
57     cin >> *valor;
58 }
59
60 // Funcion para duplicar el valor mediante puntero
61 void doblarValorPuntero(int *valor){
62     *valor=*valor*2;
63 }
64
65 // Funcion para obtener valor mediante referencia
66 void obtenerNumeroReferencia(int &valor){
67     cout << "Introduzca valor entero: ";
68     cin >> valor;
69 }
70
71 // Funcion para duplicar el valor mediante referencia
72 void doblarValorReferencia(int &valor){
73     valor=valor*2;
74 }
```

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 // Funcion 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 }
```

Índice

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
```

Índice

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.

Índice

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

Lo normal será reservar bloques de memoria mayores, para un array:

```
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

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 que 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.

```
// Se libera espacio de una variable
delete ptr;

// Se libera el espacio del array
delete [] ptrArray;
.....................................
```

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

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  }
```

Índice

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  // Declaracion de la funcion
7  int *generarNumerosAleatorios(int numeros){
8      int *array=0;
9
10     // Se controla el valor de numeros
11     if (numeros > 0) {
12         // Se reserva espacio de memoria
13         array=new int[numeros];
14
15         // Se genera semilla con fecha actual
16         srand(time(0));
17
18         // Bucle de generacion
19         for(int i=0; i < numeros; i++){
20             array[i]=rand();
21         }
22     }
23
24
```

Devolución de punteros en funciones II

```
25      // Se devuelve el array
26      return array;
27  }
28
29  // Metodo main para prueba
30  int main(int argc, char *argv[]){
31      // Variable para guardar numero de valores a generar
32      int muestras=atoi(argv[1]);
33      cout << "Muestras: " << muestras << endl;
34      // Puntero a enteros
35      int *numerosAleatorios;
36
37      // Se genera array de numeros aleatorios
38      numerosAleatorios=generarNumerosAleatorios(muestras);
39
40      // Si pudo hacerse la reserva
41      if (numerosAleatorios){
42          // Se muestran los valores generados
43          for(int i=0; i < muestras; i++){
44              cout << numerosAleatorios[i] << " ";
45          }
46          cout << endl;
47
48          // Se libera la memoria reservada
49          delete [] numerosAleatorios;
50      }
51  }
```

Índice

Punteros a objectos y estructuras I

Los punteros también pueden usarse para apuntar a objetos y estructuras. La forma de trabajo es idéntica en ambos casos. Se expone esta funcionalidad usando una clase sencilla:

```
class Rectangulo{
  public:
    int ancho;
    int alto;
    ................
    int calcularArea(){
       ..................
    }
}
```

Punteros a objectos y estructuras I

La forma de declarar un puntero a un objeto de esta clase sería:

```
Rectangulo *ptr;
// Puede asignarse a un objeto
Rectangulo rectangulo1;
ptr=&rectangulo1;
```

Punteros a objectos y estructuras I

Acceso a los datos miembro y métodos:

```
(*ptr).ancho=34;
(*ptr).alto=10;
(*ptr).calcularArea();
// Equivalente a usar operador ->
ptr->calcularArea();
```

Punteros a objectos y estructuras I

Otras posibles operaciones:

- creación de objetos en el montón usando new:

```
ptr=new Rectangulo();
// Luego habra que liberar espacio
delete ptr;
```

- paso de objetos a funciones mediante punteros
- creación de objetos en funciones y devolución de punteros a ellos

Índice

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 una única función 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  // Funciones para comprobar ordenacion ascendente
6  bool comprobarOrdenAscendente(int a, int b){
7     return(a < b);
8  }
9
10 // Funciones para comprobar ordenacion descendente
11 bool comprobarOrdenDescendente(int a, int b){
12    return(a > b);
13 }
14
15 // Funcion 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 // Funcion 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  }
```

Índice

Punteros a funciones I

Cuestiones clave:

- punteros como tipo básico para gestión dinámica de memoria
- uso de operadores new y delete
- uso y funcionamiento de punteros a objetos y/o estructuras autoreferenciadas