

Hoja de ejercicios del Tema 9

1. ¿Qué mostraría el siguiente código?

```
int x = 5, y = 12, z;  
int *p1, *p2, *p3;  
p1 = &x;  
p2 = &y;  
z = *p1 * *p2;  
p3 = &z;  
(*p3)++;  
p1 = p3;  
cout << *p1 << " " << *p2 << " " << *p3;
```

2. ¿Qué problema hay en el siguiente código?

```
int dato = 5;  
int *p1, p2;  
p1 = &dato;  
p2 = p1;  
cout << *p2;
```

3. ¿Qué problema hay en el siguiente código?

```
double d = 5.4, e = 1.2, f = 0.9;  
double *p1, *p2, *p3;  
p1 = &d;  
(*p1) = (*p1) + 3;  
p2 = &e;  
(*p3) = (*p1) + (*p2);  
cout << *p1 << " " << *p2 << " " << *p3;
```

4. ¿Cómo declararías un puntero constante p para apuntar a una constante entera? (Repasa la diapositiva 31 de la presentación del tema.)

5. Dado el siguiente tipo:

```
typedef struct {  
    string nombre;  
    double sueldo;  
    int edad;  
} tRegistro;
```

Y el siguiente subprograma:

```
void func(tRegistro &reg, double &irpf, int &edad) {  
    const double TIPO = 0.18;  
    reg.edad++;  
    irpf = reg.sueldo * TIPO;  
    edad = reg.edad;  
}
```

Reescribe el subprograma para que implemente el paso de parámetros por variable con punteros, en lugar de las referencias que usa ahora (modifica la cabecera y la implementación convenientemente).

6. Como podemos tener punteros que apunten a cualquier tipo de datos, también podemos tener punteros que apunten a punteros:

```
int x = 5;  
int *p = &x; // Puntero a entero  
int **pp = &p; // Puntero a puntero a entero
```

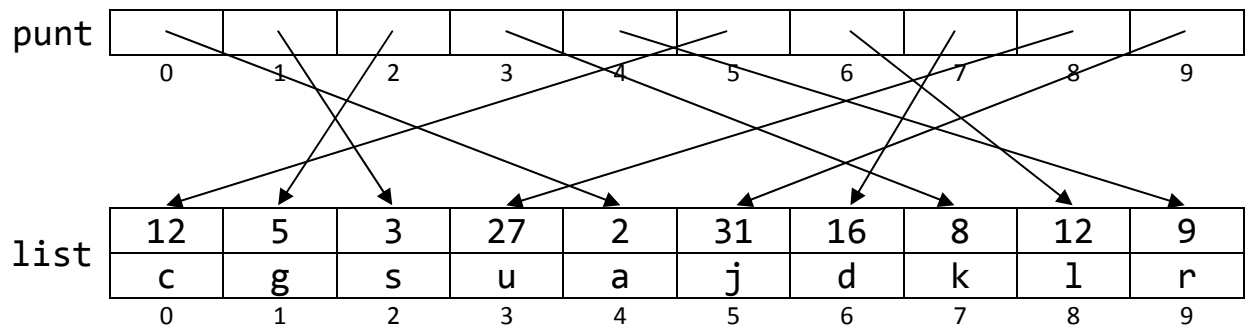
Para acceder a x a través de p escribimos *p. Para acceder a x a través de pp escribimos **pp, o *(*pp). Con *pp accedemos a p, el otro puntero.

Indica qué es lo que muestra el siguiente código:

```
int x = 5, y = 8;  
int *px = &x, *py = &y, *p;  
int **ppx = &px, **ppy = &py, **pp;  
p = px;  
px = py;  
py = p;  
pp = ppx;  
ppx = ppy;  
ppy = pp;  
cout << **ppx << " " << **ppy;
```

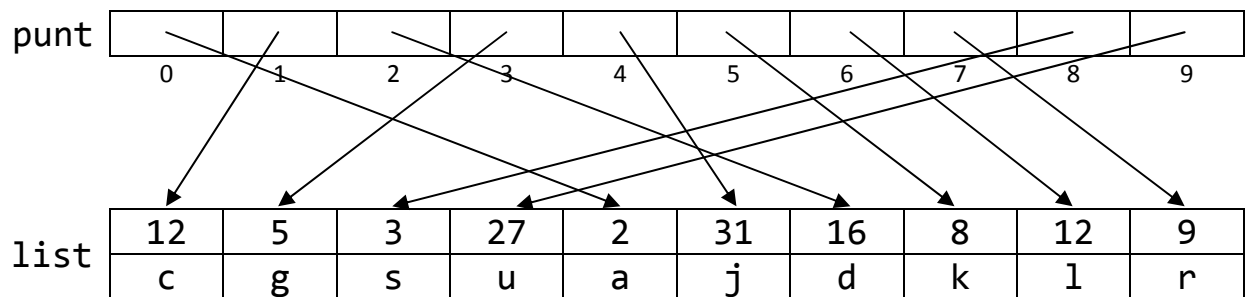
Dibuja en una hoja los distintos datos y cómo van apuntando los punteros a los otros datos a medida que se ejecutan las instrucciones.

7. Dada una lista basada en array, podemos tener otro array paralelo de punteros a los elementos de la lista para realizar ordenaciones por distintas claves sin tener que modificar el orden relativo de los elementos en la lista original:



El array de punteros indica el orden en el que hay que mostrar los registros para que estén ordenados, en este caso, por el campo `int`.

Si ahora queremos ver los registros en orden por el campo `char`:



Dadas las siguientes declaraciones:

```
typedef struct {
    int num;
    char car;
} tRegistro;
const int N = 10;
typedef struct {
    tRegistro elementos[N];
    int cont;
} tLista;
typedef const tRegistro *tPtr;
typedef struct {
    tPtr punt[N];
    int cont;
} tListaPtr; // Lista de punteros
```

Implementa dos subprogramas que devuelvan sendas listas de punteros apuntando a los registros de la lista de registros en orden de, respectivamente, `num` y `car`:

```
void porNum(const tLista &lista, tListaPtr &ord);
void porCar(const tLista &lista, tListaPtr &ord);
```

Es importante que pases a los subprogramas la lista de registros por referencia (constante, para no ser modificada), pues si se trata de un parámetro por valor, los punteros acabarían apuntando a registros de la copia local de la lista (parámetro), que se destruye al terminar la ejecución del subprograma (!).

Crea una función que muestre los registros de la lista en el orden que indique su parámetro de lista de punteros:

```
void mostrar(const tListaPtr &ord);
```

Para terminar, implementa un programa principal que use esos subprogramas para mostrar la lista ordenada por num y por car.

8. Las funciones, por supuesto, también pueden devolver punteros, tanto por medio de parámetros, como por medio de la instrucción return.

Modifica el código del Ejercicio 7, añadiendo una función que, dada la lista de punteros y un número, devuelva por parámetro un puntero al registro con ese número, o NULL si no existe ningún registro con ese número:

```
void buscaPar(const tListaPtr &ord, int num, tPtr &punt);
```

Luego añade otra función que, dada la lista de punteros y un número, devuelva por medio de return un puntero al registro con ese número, o NULL si no existe ningún registro con ese número:

```
tPtr buscaRet(const tListaPtr &ord, int num);
```

Prueba las funciones en el programa principal.

9. Modifica el código del Ejercicio 7 para que la función mostrar() reciba un puntero a tListaPtr, en lugar de un tListaPtr.

Adapta el programa principal para que genere dos listas de punteros: una con el orden por num y otra con el orden por car. Se mostrará la lista de registros ordenada a través de ambas listas de punteros.

10. Los siguientes fragmentos de código emplean memoria dinámica, pero su funcionamiento no es evidente. Indica, para cada fragmento, cuál es el resultado de la ejecución y si el código tiene algún problema o defecto. Indica también en qué zona de la memoria se guarda cada uno de los datos.

(a)

```
int* p;  
p = new int;  
*p = 100;  
cout << *p;  
p = new int;  
*p = 32;  
cout << *p;
```

(b)

```
int* p,q;  
p = new int;  
q = p;  
*p = 42;  
cout << *q;  
delete q;  
cout << *p;
```

(c)

```
int n=12;  
int* p,q;  
int** pp,qq;  
pp = new int*;  
*pp = new int;  
qq = pp;  
q = *qq;  
p = q;  
**pp = 42;  
*p = n;  
cout << **qq;  
delete p;  
delete pp;
```

- 11.** A partir del Ejercicio 9 del Tema 6 (lista de empleados de una empresa) queremos reimplementar el programa de forma modular y de paso reimplementar la lista como un array de datos dinámicos.
- 12.** Igual que el ejercicio anterior, pero esta vez reimplementando la lista como un array dinámico.
- 13.** Queremos preparar un programa para el mantenimiento del censo de una población. El programa almacenará la siguiente información de cada uno de los habitantes censados:
- ✓ Nombre (cadena de estilo C de hasta 120 caracteres)
 - ✓ Apellido 1 (cadena de estilo C de hasta 120 caracteres)
 - ✓ Apellido 2 (cadena de estilo C de hasta 120 caracteres)
 - ✓ Sexo (hombre, mujer)
 - ✓ Estado civil (soltero, casado, pareja de hecho, divorciado, viudo)
 - ✓ Dirección (cadena de estilo C de hasta 500 caracteres)
 - ✓ Fecha de nacimiento (estructura con día, mes y año)
 - ✓ DNI (sin letra, entero sin signo)
 - ✓ Código postal (entero corto sin signo)

Define las estructuras de datos necesarias para almacenar una lista de hasta N habitantes utilizando los siguientes enfoques:

- ✓ Array estático
- ✓ Array de datos dinámicos
- ✓ Array dinámico

Para cada caso, calcula, experimentalmente, el tamaño máximo de las listas que te permite el compilador. Tendrás que crear todos los datos dinámicos posibles en cada caso.

- 14.** Los arrays dinámicos pueden ser tan grandes como la capacidad del montón lo permita, pero esto no significa que debamos reservar porciones enormes de memoria si apenas necesitamos unas pocas posiciones. Una posible solución es reservar una cantidad limitada de memoria y aumentar el tamaño del array dinámico si vemos que necesitamos más espacio (crear otro más grande, copiar los elementos y eliminar el anterior; hay que mantener la capacidad actual).
- a) Modifica el ejemplo de arrays dinámicos del temario para que el tamaño inicial del array sea $N=10$.
 - b) Escribe una versión modificada de la función insertar que, en caso de que no quede capacidad para el nuevo elemento, incremente el tamaño del array dinámico en otras N unidades.

- c) Escribe una versión mejorada de la función eliminar que, al final de la eliminación, compruebe si quedan más de $2 \cdot N$ posiciones desocupadas. En caso afirmativo, deberá reducir el tamaño del array en N unidades.

15. Como ya sabemos, una pila es una estructura de datos LIFO (*Last-In, First-Out*). Las dos operaciones básicas para trabajar con pilas son *push* (añadir un elemento a la pila) y *pop* (quitar el elemento de la cima de la pila).

- La operación *push* recibe una lista y un registro, y devuelve la lista modificada con el registro puesto al final.
- La operación *pop* recibe una lista y devuelve el elemento de la cima (el último), además de quitar el elemento de la lista.

Implementa las operaciones *push* y *pop* empleando como lista un array de datos dinámicos con cadenas de caracteres.

16. Dada una lista de enteros desordenada, queremos ordenarla. Escribe los respectivos subprogramas de ordenación empleando cualquiera de los algoritmos vistos en clase para:

- a) Una lista implementada como array de objetos dinámicos.
- b) Una lista implementada como array dinámico.

