UNIVERSIDAD DE MÁLAGA
Dpto. Lenguajes y CC. Computación
E.T.S.I. Telecomunicación

GESTIÓN DE MEMORIA DINÁMICA

Tema 3

Programación II

Tema 3: GESTIÓN DE MEMORIA DINÁMICA

INTRODUCCIÓN

- Variables *automáticas* (gestionadas por el compilador)
  - Se definen mediante tipo y NOMBRE
  - Creación automática cuando la ejecución entra en el *ámbito de vida*
  - Destrucción automática cuando la ejecución sale del *ámbito de vida*
  - Tiempo de vida asociado al ámbito de su declaración
  - Capacidad de almacenamiento *predeterminado* en *Tiempo de Compilación*
- Variables *dinámicas* (gestionadas por el programador)
  - Se definen con tipo pero sin nombre, son ANÓNIMAS
  - La gestión de la *Memoria Dinámica* permite al programa gestionar el *tiempo de vida* de las variables *dinámicas* y las necesidades de *almacenamiento* en *Tiempo de Ejecución*
  - Conlleva mayor complejidad en la programación, ya que es el propio programador (introduciendo código software para ello) el que debe gestionar el tiempo de vida y accesibilidad de cada variable dinámica
  - La memoria dinámica es un *recurso* a gestionar por el programador (especial atención a la pérdida de recursos)

INTRODUCCIÓN: ÁREAS DE MEMORIA

- Áreas de Memoria durante la ejecución de un programa:
  - **Memoria global** (*estática*): almacena constantes y datos globales, con un tiempo de vida que coincide con el tiempo de ejecución del programa.
  - **Memoria automática** (*stack–pila de ejecución*): almacena los parámetros y variables locales que se crean y destruyen durante la invocación a subprogramas. Gestionada automáticamente por el compilador y el flujo de ejecución del programa.
  - **Memoria dinámica** (*heap–montículo*): almacena datos cuyo tiempo de vida está gestionado dinámicamente por el programador, y cuyo acceso se realiza a través de punteros.

Tema 3: GESTIÓN DE MEMORIA DINÁMICA

EL TIPO PUNTERO (I)

- El *tipo puntero* es un **tipo simple** que permite a un determinado programa acceder a posiciones concretas de memoria. Representa una **dirección de memoria** (equivalente a un número).

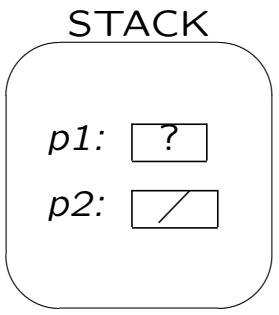
- Una variable de tipo puntero almacena una dirección de memoria, y apunta (o referencia) a una entidad (variable) de un determinado **tipo** alojada en un determinado **espacio** de la memoria.

- Para definir un tipo puntero, se debe especificar el **tipo** de la variable apuntada.

- La constante `NULL` indica una dirección de memoria **NULA**. Permite especificar que un puntero no apunta a nada (equivalente al número 0).

```
typedef int* PInt ;    // Tipo Puntero a int

struct Pers {          // Tipo Pers
    string nombre ;
    string telefono ;
    int edad ;
} ;
typedef Pers* PPers ; // Tipo Puntero a Pers
```

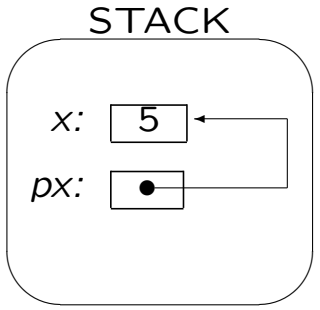
```
int main()
{
    PInt  p1 ;
    PPers p2 = NULL;

    int*  p3;
    Pers* p4 = NULL;
}
```



EL TIPO PUNTERO (II)

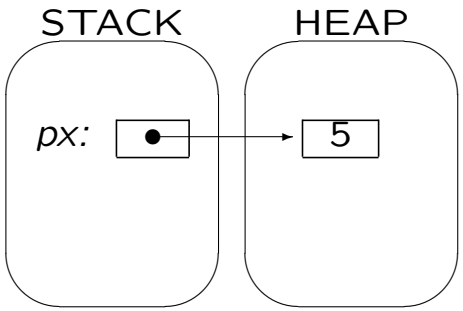
- El operador *"ampersand"* (`&`) permite obtener la dirección de memoria donde se almacena el contenido de una variable o parámetro.
  - No confundir con el especificador del paso por referencia.
  - Utilizado previamente para evitar la autoasignación al definir el *operador de asignación*.
  - Es una característica de **bajo nivel**, por lo que no será utilizada en este curso (excepto para evitar la autoasignación en el operador de asignación).
- El operador de *"desreferenciación"* (`*`) permite acceder al contenido almacenado en la variable apuntada por un puntero.
  - No confundir con el operador binario de multiplicación.
- En las figuras, el círculo representa la dirección de memoria almacenada por una variable de tipo puntero, y la flecha señala simbólicamente a la variable apuntada.

```
typedef int* PInt ;   // Tipo Puntero a int
int main()
{
    int  x = 3 ;
    PInt px = &x ;
    *px = 5 ;
    cout << x << " " << *px << endl ; // escribe: 5 5
}
```



EL TIPO PUNTERO (III)

- En este curso, utilizaremos variables de tipo puntero para acceder a variables almacenadas en la memoría dinámica (variables anónimas).
- La gestión de la memoria dinámica a través de punteros implica:
  - Gestión de la propia variable de tipo puntero.
    - Nótese que el tiempo de vida de las variables con NOMBRE es gestionado por el compilador.
  - Gestión de la variable dinámica apuntada por el puntero y su tiempo de vida
    - Reserva del espacio en memoria dinámica y construcción del objeto.
    - Acceso y manipulación de la información almacenada en memoria dinámica.
    - Destrucción del objeto y liberación del espacio en memoria dinámica.
- En las figuras, el círculo representa la dirección de memoria almacenada por una variable de tipo puntero, y la flecha señala simbólicamente a la variable dinámica anónima apuntada.




```
typedef int* PInt ;        // Tipo Puntero a int
int main()
{
    PInt px ;              // construcción automática de la variable PX
    px = new int ;         // crea una variable anónima de tipo int
    *px = 5 ;              // modifica el valor de la variable anónima
    cout << *px << endl ; // accede al valor de la variable anónima
    delete px ;            // destruye la variable anónima y libera su espacio de memoria
}                          // destrucción automática de la variable PX
```

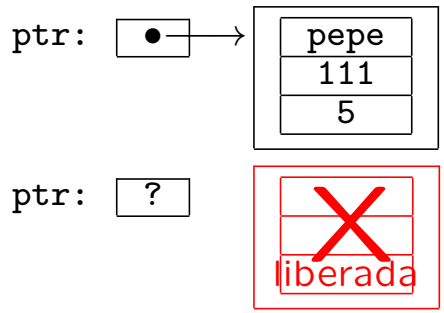
Tema 3: GESTIÓN DE MEMORIA DINÁMICA

GESTIÓN DE MEMORIA DINÁMICA (I)

- Creación de variable anónima: operador `new` seguido del *tipo de la variable apuntada*
  - Aloja y reserva la zona de memoria dinámica necesaria
  - Inicializa el contenido de la variable invocando al constructor especificado
  - Devuelve la dirección de memoria donde se aloja la variable dinámica
- Destrucción de variable anónima: operador `delete` seguido por la variable de tipo puntero que apunta a la variable dinámica que se desea destruir
  - Destruye el contenido de la variable dinámica invocando al destructor
  - Desaloja y libera la memoria dinámica reservada apuntada por el puntero
  - La variable puntero queda con valor inespecificado
  - Si la variable puntero tiene un valor NULL, entonces `delete` no hace nada
- Atención a la **pérdida de recursos** de memoria dinámica

```
int main()
{
    PPers ptr ;
    ptr = new Pers ;
    // manipulación ...

    delete ptr ;
}
```

```
int main()
{
    PPers ptr ;
    ptr = new Pers("pepe", "111", 5) ;
    // manipulación ...

    delete ptr ;
}
```

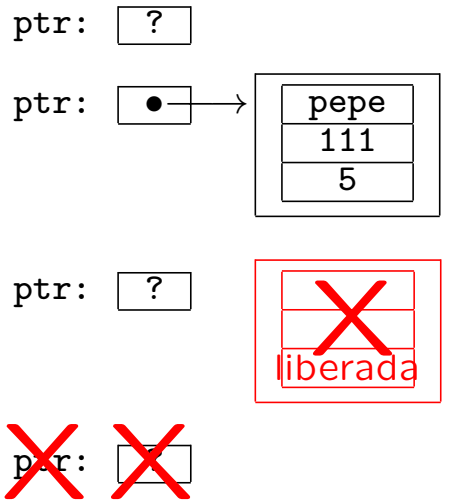


GESTIÓN DE MEMORIA DINÁMICA (II)

- La operación `delete ptr;` **destruye la variable dinámica** (anónima) apuntada por la variable `ptr`.

- La variable `ptr` es una variable con nombre, y por lo tanto su tiempo de vida está gestionado por el compilador.
  - La variable `ptr` se **construye automáticamente** cuando el flujo de ejecución alcanza el **comienzo** del ámbito en el que dicha variable es definida.
  - La variable `ptr` se **destruye automáticamente** cuando el flujo de ejecución alcanza el **final** del ámbito en el que dicha variable es definida.

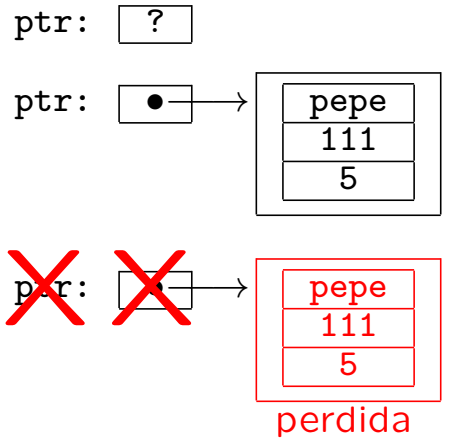
```
int main()
{
    PPers ptr ;      // creación automática de la variable PTR

    ptr = new Pers ; // creación de la variable dinámica (anónima)

    // manipulación ...

    delete ptr ;    // destrucción de la variable dinámica (anónima)



}                   // destrucción automática de la variable PTR
```



GESTIÓN DE MEMORIA DINÁMICA (III)

- Hay que prestar atención a la **pérdida de recursos** de memoria dinámica cuando se destruye una variable de tipo puntero, ya que podría estar apuntando a una variable dinámica anónima válida (no destruida).

```
int main()
{
    PPers ptr ;      // creación automática de la variable PTR

    ptr = new Pers ; // creación de la variable dinámica (anónima)

    // manipulación ...

}                    // destrucción automática de la variable PTR
                     // pérdida de la variable dinámica (anónima)
```

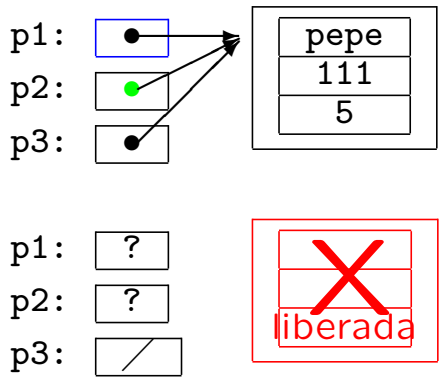


Tema 3: GESTIÓN DE MEMORIA DINÁMICA

OPERACIONES CON PUNTEROS: ASIGNACIÓN (I)

- La constante `NULL` se puede asignar a cualquier variable de tipo puntero
- El resultado de `new` se puede asignar a una variable puntero al tipo alojado
- Las variables de tipo puntero al mismo tipo se pueden asignar entre ellas.
  - Se asigna (copia) el valor del puntero (un número que representa una dirección en memoria donde está almacenada la variable anónima). Por ejemplo la asignación `p1 = p2;`
    - El valor de `p2` (representado con un circulo de color verde) se copia a la variable `p1` (representada con una caja de color azul).
  - Ambos punteros apuntarán a la misma variable dinámica **compartida**
  - Atención a la liberación de la variable dinámica compartida

```
int main()
{
    PPers p1, p2, p3 ;

    p1 = NULL ;
    p2 = new Pers("pepe", "111", 5) ;
    p1 = p2 ;
    p3 = p2 ;
    p3 = NULL ;
    delete p1 ;
}
```

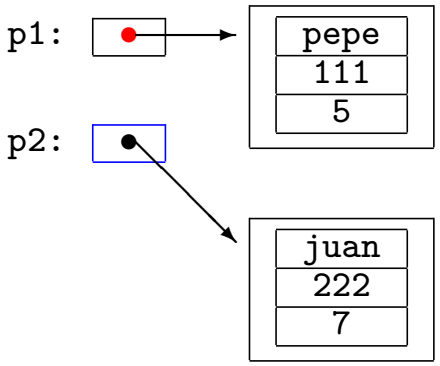


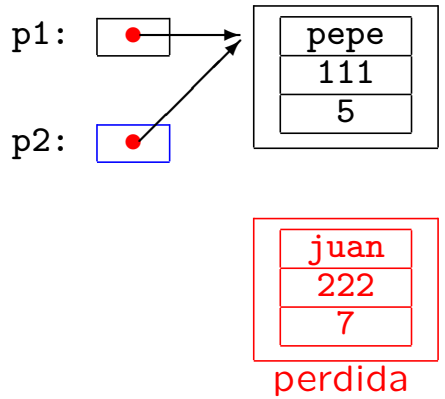
OPERACIONES CON PUNTEROS: ASIGNACIÓN (II)

- El valor almacenado en el puntero anterior a la asignación se pierde al asignar (copiar) un nuevo valor.

- Hay que prestar atención a la **pérdida de recursos** que pudiera producir si se realiza de forma inadecuada. Por ejemplo:
  - La primera figura es el resultado de la ejecución del siguiente programa hasta la línea 6 inclusive.
  - La segunda figura es el resultado de la ejecución del siguiente programa hasta la línea 7 inclusive.
  - Podemos apreciar como, después de la asignación de la línea 7, se **pierde** la memoria de la variable dinámica anteriormente apuntada por `p2`.

```
1 int main()
2 {
3   PPers p1, p2 ;
4
5   p1 = new Pers ;
6   p2 = new Pers ;
7   p2 = p1 ;
8   delete p1 ;
9 }
```





OPERACIONES CON PUNTEROS: DESREFERENCIACIÓN (I)

- Desreferenciación: acceso a la variable dinámica apuntada
- Operador (*): `*ptr` es la variable dinámica apuntada por `ptr`
- Operador (->): `ptr->nombre` accede al miembro de la variable apuntada por `ptr`
- Operador (->): `ptr->metodo(...)` invoca al método del objeto apuntado por `ptr`
- Desreferenciar un puntero con valor NULL produce un **error** en tiempo de ejecución (aborta la ejecución)
- Desreferenciar un puntero con valor inespecificado produce un comportamiento **anómalo** en tiempo de ejecución

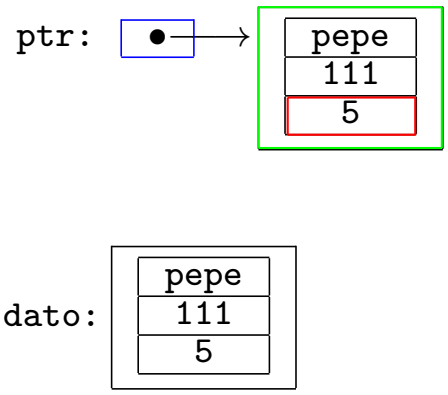
```
int main()
{
    PPers ptr = new Pers("pepe", "111", 5) ;

    Pers p = *ptr ;

    *ptr = p ;

    delete ptr ;
}
```

```
int main()
{
    PPers ptr = new Pers ;

    ptr->nombre   = "pepe" ;
    ptr->telefono = "111" ;
    ptr->edad     = 5 ;

    delete ptr ;
}
```

OPERACIONES CON PUNTEROS: DESREFERENCIACIÓN (II)

- Cuando se **asigna** un valor a una variable, el nuevo valor se almacena en su zona de memoria (**caja**) correspondiente.
- Cuando se **toma** el valor de una variable, se toma el valor almacenado en la zona de memoria (**caja**) correspondiente.
- Si la variable `ptr` apunta a una variable dinámica de tipo `Pers`, entonces:
  - La variable `ptr` contiene la dirección de memoria (un número) donde se encuentra almacenada la variable dinámica. En la figura está representada por una **caja** en color azul.
  - La operación `*ptr` se refiere a la variable dinámica (anónima) completa apuntada por `ptr`. En la figura está representada por una **caja** en color verde.
  - La operación `ptr->edad` se refiere al campo `edad` de la variable dinámica (anónima) apuntada por `ptr`. En la figura está representada por una **caja** en color rojo.

```
int main()
{
    PPers ptr ;
    ptr = new Pers ;

    ptr->nombre   = "pepe" ;
    ptr->telefono = "111" ;
    ptr->edad     = 5 ;

    Pers dato = *ptr ;

    delete ptr ;
}
```



OPERACIONES: COMPARACIÓN Y PASO DE PARÁMETROS

- Los punteros se pueden comparar por igualdad (==) y desigualdad (!=)
  - Comparación de un puntero con la constante `NULL`
  - Comparación entre punteros que apuntan al mismo tipo
- Los punteros se pueden pasar como parámetros *por valor* y por *referencia*
  - Los punteros son **tipos simples**
  - Las funciones pueden devolver punteros
  - Atención a modificaciones de la variable apuntada en el paso por valor

```
int main()
{
    PPers p1, p2 ;
    // ...
    if (p1 != NULL) {
        // ...
    }
    if (p1 == p2) {
        // ...
    }
}
```

```
void crear(PPers& p)
{
    p = new Pers ;
    p->nombre = "pepe" ;
    p->telefono = "111" ;
    p->edad = 5 ;
}
void modificar(PPers p)
{
    // DESACONSEJADO MODIFICAR
    p->nombre = "juan" ;
}
```

```
PPers crear() ;
{
    PPers p = new Pers ;
    p->nombre = "pepe" ;
    p->telefono = "111" ;
    p->edad = 5 ;
    return p;
}
```

Tema 3: GESTIÓN DE MEMORIA DINÁMICA

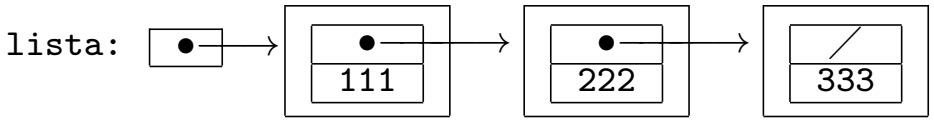
LISTAS ENLAZADAS LINEALES (I)

- Permiten almacenar en memoria dinámica secuencias de un número inespecificado de elementos homogéneos

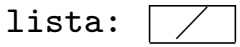
- Estructuras enlazadas: un campo de enlace del tipo de la variable dinámica es de tipo puntero al mismo tipo de la variable dinámica

```
struct Nodo ;          // Declaración adelantada del tipo incompleto Nodo
typedef Nodo* PNodo ; // Definición de tipo Puntero a tipo incompleto Nodo
struct Nodo {          // Definición del tipo Nodo
    PNodo sig ;        // Enlace a la siguiente estructura dinámica
    int dato ;         // Dato almacenado en la lista (cualquier tipo)
} ;
```

- Una lista es una variable de tipo puntero que apunta a una variable dinámica, donde un campo de enlace de la variable dinámica apunta a otra variable dinámica que a su vez apunta a otra, así hasta que en la última, el campo de enlace no apunta a nada (`NULL`)



- En una lista **vacía** (sin elementos), la variable de tipo puntero que apunta al primer elemento de la lista toma el valor `NULL`



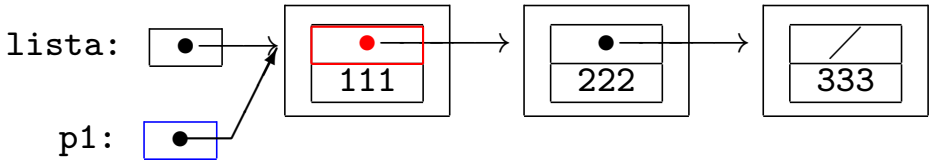
LISTAS ENLAZADAS LINEALES (II)

- En el trabajo con listas enlazadas, es importante tener claro los conceptos de asignación y de desreferenciación.

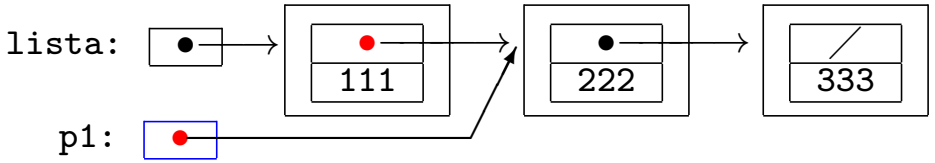
- Si la variable `p1` apunta a una variable dinámica de tipo `Nodo`, entonces:

  - La variable `p1` contiene la dirección de memoria (un número) donde se encuentra almacenada la variable dinámica. En la figura está representada por una **caja** en color azul.

  - La operación `p1->sig` se refiere al campo `sig` de la variable dinámica (anónima) apuntada por `p1`. En la figura está representada por una **caja** en color rojo.

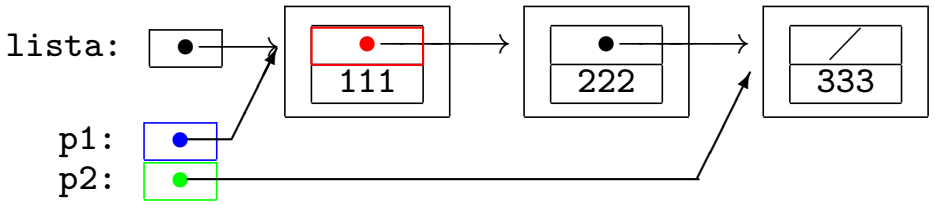


  - La sentencia `p1 = p1->sig`, almacena en la variable `p1` (caja azul) el valor de `p1->sig` (circulo rojo), por lo que la variable `p1` pasa a apuntar al siguiente nodo.

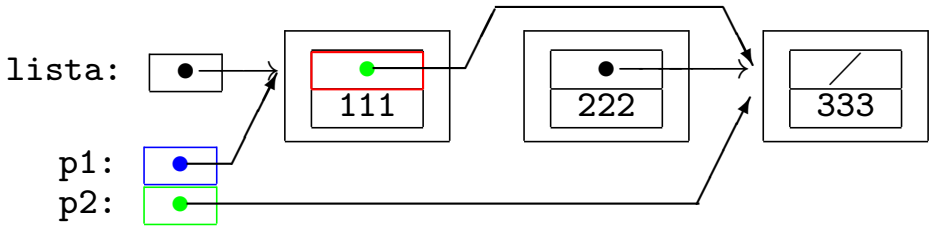


LISTAS ENLAZADAS LINEALES (III)

- En el trabajo con listas enlazadas, es importante tener claro los conceptos de asignación y de desreferenciación.
- Si las variables `p1` y `p2` apuntan a variables dinámicas de tipo `Nodo`, entonces:
  - La variable `p1` contiene la dirección de memoria (un número) donde se encuentra almacenada la variable dinámica. En la figura está representada por una **caja** en color azul.
  - La operación `p1->sig` se refiere al campo `sig` de la variable dinámica (anónima) apuntada por `p1`. En la figura está representada por una **caja** en color rojo.
  - La variable `p2` contiene la dirección de memoria (un número) donde se encuentra almacenada la variable dinámica. En la figura está representada por una **caja** en color verde.

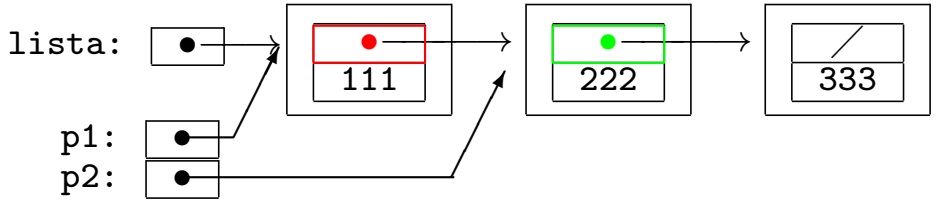


  - La sentencia `p1->sig = p2`, almacena en la variable `p1->sig` (caja roja) el valor de `p2` (circulo verde), por lo que la variable `p1->sig` pasa a apuntar al tercer nodo.

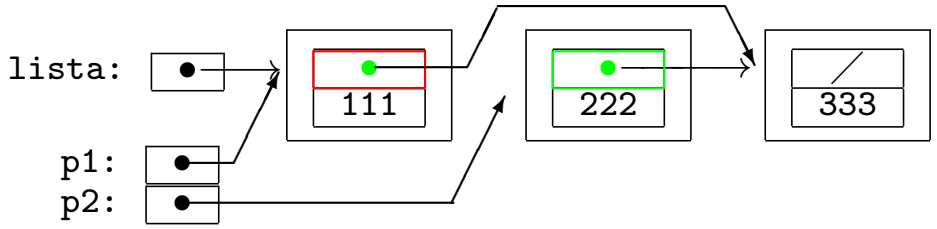


LISTAS ENLAZADAS LINEALES (IV)

- En el trabajo con listas enlazadas, es importante tener claro los conceptos de asignación y de desreferenciación.
- Si las variables `p1` y `p2` apuntan a variables dinámicas de tipo `Nodo`, entonces:
  - Las variables `p1` y `p2` contienen las direcciones de memoria (unos números) donde se encuentran almacenadas las variables dinámicas apuntadas.
  - La operación `p1->sig` se refiere al campo `sig` de la variable dinámica (anónima) apuntada por `p1`. En la figura está representada por una **caja** en color rojo.
  - La operación `p2->sig` se refiere al campo `sig` de la variable dinámica (anónima) apuntada por `p2`. En la figura está representada por una **caja** en color verde.



  - La sentencia `p1->sig = p2->sig`, almacena en la variable `p1->sig` (caja roja) el valor de `p2->sig` (circulo verde), por lo que la variable `p1->sig` pasa a apuntar al tercer nodo.



LISTAS ENLAZADAS LINEALES. OPERACIONES BÁSICAS (I)

- `void inicializar(PNodo& lista)`: inicializa `lista` a una lista vacía.
- `void mostrar(PNodo lista)`: muestra en pantalla el contenido de `lista`.
- `PNodo buscar(PNodo lista, int dt)`: devuelve un puntero al primer nodo que contiene un elemento igual a `dt`. Si no se encuentra, entonces devuelve `NULL`.
- `void destruir(PNodo& lista)`: destruye y libera todos los elementos de `lista`, quedando vacía.
- `void insertar_principio(PNodo& lista, int dt)`: inserta un elemento al principio de `lista`.
- `void insertar_final(PNodo& lista, int dt)`: inserta un elemento al final de `lista`.
- `void insertar_ord(PNodo& lista, int dt)`: inserta un elemento de forma ordenada en `lista`, que debe estar ordenada.
- `void eliminar_primero(PNodo& lista)`: elimina, si existe, el primer elemento de `lista`.

- `void eliminar_ultimo(PNodo& lista)`: elimina, si existe, el último elemento de `lista`.

- `void eliminar_elem(PNodo& lista, int dt, bool& ok)`: elimina, si existe, el primer elemento de `lista` igual a `dt`.

- `PNodo situar(PNodo lista, int pos)`: devuelve un puntero al nodo que se encuentra en la posición indicada por `pos`. La posición 0 indica el primer nodo. Si no existe, entonces devuelve NULL.

- `void insertar_pos(PNodo& lista, int pos, int dt)`: inserta en `lista` un elemento en la posición indicada por `pos`.

- `void eliminar_pos(PNodo& lista, int pos)`: elimina de `lista`, si existe, el elemento de la posición indicada por `pos`.

- `PNodo duplicar(PNodo lista)`: devuelve un puntero a una *nueva lista* resultado de duplicar en memoria dinámica la lista recibida como parámetro.

- `void eliminar_mayor(PNodo& lista)`: elimina, si existe, el mayor elemento de `lista`.

- `void purgar(PNodo& lista, int dt)`: elimina de 'lista' todos los elementos que sean iguales a 'dt'.

- `PNodo leer()`: devuelve una lista con los números leídos de teclado (en el mismo orden que son introducidos) hasta que lea el numero 0 (que no es introducido).

LISTAS ENLAZADAS LINEALES. OPERACIONES BÁSICAS (II)

```
void inicializar(PNodo& lista)
{
    lista = NULL ;
}
void mostrar(PNodo lista)
{
    PNodo ptr = lista ;
    while (ptr != NULL) {
        cout << ptr->dato << endl ;
        ptr = ptr->sig ;
    }
}
PNodo buscar(PNodo lista, int dt)
{
    PNodo ptr = lista ;
    while ((ptr != NULL)&&(ptr->dato != dt)) {
        ptr = ptr->sig ;
    }
    return ptr ;
}
void destruir(PNodo& lista)
{
    while (lista != NULL) {
        PNodo ptr = lista ;
        lista = lista->sig ;
        delete ptr ;
    }
}
```

```
PNodo leer_inversa()
{
    PNodo lista ;
    int dt ;
    inicializar(lista) ; // inicializa la lista vacía
    cin >> dt ;
    while (dt != 0) {
        PNodo ptr = new Nodo ;
        ptr->dato = dt ;
        ptr->sig = lista ;
        lista = ptr ;
        cin >> dt ;
    }
    return lista ;
}
int main()
{
    PNodo lista ;
    lista = leer_inversa() ;
    mostrar(lista) ;
    PNodo ptr = buscar(lista, 5) ;
    if (ptr != NULL) {
        cout << ptr->dato << endl ;
    }
    destruir(lista) ;
}
```

LISTAS ENLAZADAS LINEALES. OPERACIONES BÁSICAS (III)

```
void insertar_principio(PNodo& lista, int dt)
{
    PNodo ptr = new Nodo ;
    ptr->dato = dt ;
    ptr->sig = lista ;
    lista = ptr ;
}

void insertar_final(PNodo& lista, int dt)
{
    PNodo ptr = new Nodo ;
    ptr->dato = dt ;
    ptr->sig = NULL ;
    if (lista == NULL) {
        lista = ptr ;
    } else {
        PNodo act = lista ;
        while (act->sig != NULL) {
            act = act->sig ;
        }
        act->sig = ptr ;
    }
}
```

```
void insertar_ord(PNodo& lista, int dt)
{
    PNodo ptr = new Nodo ;
    ptr->dato = dt ;
    if ((lista == NULL)||(dt < lista->dato)) {
        ptr->sig = lista ;
        lista = ptr ;
    } else {
        PNodo ant = lista ;
        PNodo act = ant->sig ;
        while ((act!=NULL)&&(act->dato<=dt)) {
            ant = act ;
            act = act->sig ;
        }
        ptr->sig = ant->sig ;
        ant->sig = ptr ;
    }
}
```

LISTAS ENLAZADAS LINEALES. OPERACIONES BÁSICAS (IV)

```
void eliminar_primero(PNodo& lista)
{
    if (lista != NULL) {
        PNodo ptr = lista ;
        lista = lista->sig ;
        delete ptr ;
    }
}

void eliminar_ultimo(PNodo& lista)
{
    if (lista != NULL) {
        if (lista->sig == NULL) {
            delete lista ;
            lista = NULL ;
        } else {
            PNodo ant = lista ;
            PNodo act = ant->sig ;
            while (act->sig != NULL) {
                ant = act ;
                act = act->sig ;
            }
            delete act ;
            ant->sig = NULL ;
        }
    }
}
```

```
void eliminar_elem(PNodo& lista, int dt, bool& ok)
{
    ok = false;
    if (lista != NULL) {
        if (lista->dato == dt) {
            PNodo ptr = lista ;
            lista = lista->sig ;
            delete ptr ;
            ok = true;
        } else {
            PNodo ant = lista ;
            PNodo act = ant->sig ;
            while ((act != NULL)&&(act->dato != dt)) {
                ant = act ;
                act = act->sig ;
            }
            if (act != NULL) {
                ant->sig = act->sig ;
                delete act ;
                ok = true;
            }
        }
    }
}
```

LISTAS ENLAZADAS LINEALES. OPERACIONES BÁSICAS (V)

```
PNodo situar(PNodo lista, int pos)
{
    int i = 0;
    PNodo ptr = lista;
    while ((ptr != NULL)&&(i < pos)) {
        ptr = ptr->sig;
        ++i;
    }
    return ptr;
}
void insertar_pos(PNodo& lista, int pos, int dt)
{
    if (pos < 1) {
        PNodo ptr = new Nodo ;
        ptr->dato = dt ;
        ptr->sig = lista ;
        lista = ptr ;
    } else {
        PNodo ant = situar(lista, pos - 1);
        if (ant != NULL) {
            PNodo ptr = new Nodo ;
            ptr->dato = dt ;
            ptr->sig = ant->sig ;
            ant->sig = ptr ;
        }
    }
}
```

```
void eliminar_pos(PNodo& lista, int pos)
{
    if (lista != NULL) {
        if (pos < 1) {
            PNodo ptr = lista ;
            lista = lista->sig ;
            delete ptr ;
        } else {
            PNodo ant = situar(lista, pos - 1) ;
            if ((ant != NULL)&&(ant->sig != NULL)) {
                PNodo act = ant->sig ;
                ant->sig = act->sig ;
                delete act ;
            }
        }
    }
}
```

LISTAS ENLAZADAS LINEALES. OPERACIONES BÁSICAS (VI)

```
void insertar_ultimo(PNodo& lista, PNodo& ult,
                     int dt)
{
    PNodo p = new Nodo ;
    p->dato = dt ;
    p->sig = NULL ;
    if (lista == NULL) {
        lista = p ;
    } else {
        ult->sig = p ;
    }
    ult = p ;
}
PNodo leer()
{
    PNodo lista ;
    PNodo ult = NULL ;
    int dt ;
    inicializar(lista) ; // inic la lista vacía
    cin >> dt ;
    while (dt != 0) {
        insertar_ultimo(lista, ult, dt) ;
        cin >> dt ;
    }
    return lista;
}
```

```
PNodo duplicar(PNodo lista)
{
    PNodo nueva = NULL;
    if (lista != NULL) {
        nueva = new Nodo ;
        nueva->dato = lista->dato ;
        PNodo u = nueva ;
        PNodo p = lista->sig ;
        while (p != NULL) {
            u->sig = new Nodo ;
            u->sig->dato = p->dato ;
            u = u->sig ;
            p = p->sig ;
        }
        u->sig = NULL ;
    }
    return nueva;
}
```

LISTAS ENLAZADAS LINEALES. OPERACIONES BÁSICAS (VII)

```
void eliminar_mayor(PNodo& lista)
{
    if (lista != NULL) {
        PNodo ant_may = NULL ;
        PNodo ptr_may = lista ;
        PNodo ant = lista ;
        PNodo ptr = lista->sig ;
        while (ptr != NULL) {
            if (ptr->dato > ptr_may->dato) {
                ant_may = ant ;
                ptr_may = ptr ;
            }
            ant = ptr ;
            ptr = ptr->sig ;
        }
        if (ptr_may == lista) {
            lista = lista->sig;
        } else {
            ant_may->sig = ptr_may->sig;
        }
        delete ptr_may;
    }
}
```

```
void purgar(PNodo& lista, int dt)
{
    while ((lista!=NULL)&&(dt == lista->dato)) {
        PNodo ptr = lista;
        lista = lista->sig;
        delete ptr;
    }
    if (lista != NULL) {
        PNodo ant = lista;
        PNodo act = lista->sig;
        while (act != NULL) {
            if (dt == act->dato) {
                ant->sig = act->sig;
                delete act;
            } else {
                ant = act;
            }
            act = ant->sig;
        }
    }
}
```

Tema 3: GESTIÓN DE MEMORIA DINÁMICA

ABSTRACCIÓN EN LA GESTIÓN DE MEMORIA DINÁMICA (I)

- La gestión de memoria dinámica da lugar a código complejo, propenso a errores de programación y pérdida de recursos de memoria
- Mezclar sentencias de gestión de memoria con sentencias aplicadas al dominio de problema a resolver da lugar a código no legible y propenso a errores
- Es adecuado aplicar niveles de abstracción que aislen la gestión de memoria dinámica del resto del código
- Los tipos abstractos de datos permiten aplicar la abstracción a estructuras de datos basadas en la gestión de memoria dinámica
- La gestión de recursos (memoria dinámica) se realiza adecuadamente gracias a los constructores y destructores (liberan recursos asociados a un objeto)

```
class Lista {
public:
    ~Lista() { destruir(...) ; }
    Lista() : sz(0), lista(NULL) { }
    void insertar(int pos, const Tipo& d) { ... }
    void eliminar(int pos) { ... }
    // ...
} ;
```

- Hay que duplicar la memoria dinámica al copiar/asignar objetos
- Hay que liberar la memoria dinámica antes de asignar nueva o de destruir el objeto

ABSTRACCIÓN EN LA GESTIÓN DE MEMORIA DINÁMICA (II)

- El *Destructor* debe liberar la memoria dinámica antes de destruir el objeto
- El *Ctor-Defecto* debe inicializar adecuadamente los atributos de tipo puntero
- El *Ctor-Copia* debe duplicar la memoria dinámica del objeto a copiar
- El *Op-Asignación* debe liberar la memoria dinámica actual y duplicar la memoria dinámica del objeto a asignar
- Los *métodos* de la clase permiten manipular la estructura de datos, proporcionando abstracción sobre su complejidad y representación interna
- El acceso restringido a la representación interna impide una manipulación externa propensa a errores

```
class Lista {
public:
    ~Lista() { destruir(lista) ; }
    Lista() : sz(0), lista(NULL) { }
    Lista(const Lista& o)
        : sz(o.sz), lista(duplicar(o.lista)) { }
    Lista& operator = (const Lista& o)
    {
        if (this != &o) {
            destruir(lista) ;
            sz = o.sz ;
            lista = duplicar(o.lista) ;
        }
        return *this ;
    }
private:
    struct Nodo ;
    typedef Nodo* PNodo ;
    struct Nodo {
        PNodo sig ;
        Tipo dato ;
    } ;
    //-- Atributos privados --
    int sz ;
    PNodo lista ;
} ;
```

Tema 3: GESTIÓN DE MEMORIA DINÁMICA

EJEMPLO: TAD LISTA DE NÚMEROS ENTEROS

- Diseñe un TAD lista de números enteros posicional que contiene una secuencia de elementos homogéneos (de números enteros) accesibles por su posición, y defina los siguientes métodos:
  - Destructor: destruye y libera los recursos asociados a la lista
  - Constructor por defecto: construye una lista vacía
  - Constructor de copia: copia el contenido de otra lista
  - Operador de asignación: asigna el contenido de otra lista
  - Llena: método constante que devuelve si el número de elementos almacenados en la lista alcanzó su máxima capacidad (en este caso nunca se alcanza la máxima capacidad)
  - Size: método constante que devuelve el número de elementos almacenados en la lista
  - Clear: elimina los elementos almacenados, dejando la lista vacía
  - Insertar: inserta el elemento recibido como parámetro en la posición indicada como parámetro. *PRECOND:* `( ! llena() && 0 <= pos && pos <= size() )`
  - Eliminar: elimina el elemento que se encuentra en la posición indicada como parámetro. *PRECOND:* `( 0 <= pos && pos < size() )`
  - Acceder: método constante que devuelve el elemento almacenado en la posición indicada como parámetro. *PRECOND:* `( 0 <= pos && pos < size() )`
  - Modificar: asigna el elemento recibido como parámetro al elemento que se encuentra en la posición indicada como parámetro. *PRECOND:* `( 0 <= pos && pos < size() )`

TAD LISTA ENTEROS (I) [listaint.hpp]

```
#ifndef listaint_hpp_
#define listaint_hpp_
namespace umalcc {
    class ListaInt {
    public:
        //-- Métodos Públicos --------
        ~ListaInt() ;
        ListaInt() ;
        ListaInt(const ListaInt& o) ;
        ListaInt& operator=(const ListaInt& o) ;
        bool llena() const ;
        int size() const ;
        void clear() ;
        void insertar(int pos, int d) ;
        void eliminar(int pos) ;
        int acceder(int pos) const ;
        void modificar(int pos, int d) ;
    private:
        //-- Tipos Privados ----------
        struct Nodo ;
        typedef Nodo* PNodo ;
        struct Nodo {
            PNodo sig ;
            int dato ;
        } ;
        //-- Métodos Privados --------
        void destruir(PNodo& lst) const ;
        PNodo duplicar(PNodo lst) const ;
        PNodo situar(PNodo lst, int pos) const ;
        void insertar_pos(PNodo& lst, int pos,
                          int d) const ;
        void eliminar_pos(PNodo& lst, int pos) const ;
        //-- Atributos privados ------
        int sz ;
        PNodo lista ;
        //----------------------------
    } ; // fin de class ListaInt
} // fin de namespace umalcc
#endif
```

TAD LISTA ENTEROS (II) [listaint.cpp]

```
#include "listaint.hpp"
#include <cstddef>
#include <cassert>
using namespace std;
namespace umalcc {
    ListaInt::~ListaInt()
    {
        destruir(lista) ;
    }
    ListaInt::ListaInt()
        : sz(0), lista(NULL) { }

    ListaInt::ListaInt(const ListaInt& o)
        : sz(o.sz), lista(duplicar(o.lista)) { }

    ListaInt& ListaInt::operator=(const
                                  ListaInt& o)
    {
        if (this != &o) {
            destruir(lista) ;
            sz = o.sz ;
            lista = duplicar(o.lista) ;
        }
        return *this ;
    }
```

```
    bool ListaInt::llena() const
    {
        return false;
    }
    void ListaInt::clear()
    {
        destruir(lista) ;
        sz = 0 ;
    }
    int ListaInt::size() const
    {
        return sz ;
    }
    void ListaInt::insertar(int pos, int d)
    {
        assert(! llena() && 0 <= pos && pos <= size()) ;
        insertar_pos(lista, pos, d) ;
        ++sz ;
    }
    void ListaInt::eliminar(int pos)
    {
        assert(0 <= pos && pos < size()) ;
        eliminar_pos(lista, pos) ;
        --sz ;
    }
```

TAD LISTA ENTEROS (III) [listaint.cpp] (Nótese la definición de tipo recuadrada)

```
int ListaInt::acceder(int pos) const
{
    assert(0 <= pos && pos < size()) ;
    PNodo ptr = situar(lista, pos) ;
    assert(ptr != NULL) ;
    return ptr->dato ;
}

void ListaInt::modificar(int pos, int d)
{
    assert(0 <= pos && pos < size()) ;
    PNodo ptr = situar(lista, pos) ;
    assert(ptr != NULL) ;
    ptr->dato = d ;
}

ListaInt::PNodo ListaInt::situar(PNodo lst,
                                 int pos)
                                 const
{
    int i = 0;
    PNodo ptr = lst;
    while ((ptr != NULL)&&(i < pos)) {
        ptr = ptr->sig;
        ++i;
    }
    return ptr;
}
```

```
void ListaInt::destruir(PNodo& lst) const
{
    while (lst != NULL) {
        PNodo ptr = lst ;
        lst = lst->sig ;
        delete ptr ;
    }
}
ListaInt::PNodo ListaInt::duplicar(PNodo lst) const
{
    PNodo nueva = NULL;
    if (lst != NULL) {
        nueva = new Nodo ;
        nueva->dato = lst->dato ;
        PNodo u = nueva ;
        PNodo p = lst->sig ;
        while (p != NULL) {
            u->sig = new Nodo ;
            u->sig->dato = p->dato ;
            u = u->sig ;
            p = p->sig ;
        }
        u->sig = NULL ;
    }
    return nueva;
}
```

TAD LISTA ENTEROS (IV) [listaint.cpp]

```
void ListaInt::insertar_pos(PNodo& lst,
                            int pos,
                            int d) const
{
    if (pos < 1) {
        PNodo ptr = new Nodo ;
        ptr->dato = d ;
        ptr->sig = lst ;
        lst = ptr ;
    } else {
        PNodo ant = situar(lst, pos - 1);
        if (ant != NULL) {
            PNodo ptr = new Nodo ;
            ptr->dato = d ;
            ptr->sig = ant->sig ;
            ant->sig = ptr ;
        }
    }
}
```

```
void ListaInt::eliminar_pos(PNodo& lst,
                            int pos) const
{
    if (lst != NULL) {
        if (pos < 1) {
            PNodo ptr = lst ;
            lst = lst->sig ;
            delete ptr ;
        } else {
            PNodo ant = situar(lst, pos - 1) ;
            if ((ant != NULL)&&(ant->sig != NULL)) {
                PNodo act = ant->sig ;
                ant->sig = act->sig ;
                delete act ;
            }
        }
    }
}
} // fin de namespace umalcc
```

Tema 3: GESTIÓN DE MEMORIA DINÁMICA

EJEMPLO: TAD DATO

- Diseñe un TAD `Dato` que contiene un valor de tipo entero, y defina los siguientes métodos:
  - Destructor: destruye y libera los recursos asociados al dato
  - Constructor por defecto: construye un dato con valor cero
  - Constructor específico: construye un dato con valor recibido como parámetro
  - Constructor de copia: copia el contenido de otro dato
  - Operador de asignación: asigna el contenido de otro dato
  - Acceder: método constante que devuelve el valor contenido en el dato
  - Modificar: cambia el valor contenido en el dato por el valor recibido como parámetro

TAD DATO [dato.hpp y dato.cpp]

```
// fichero: dato.hpp
#ifndef dato_hpp_
#define dato_hpp_
namespace umalcc {
    class Dato {
    public:
        // ~Dato();
        // Dato(const Dato& o);
        // Dato& operator=(const Dato& o);
        Dato();
        Dato(int v);
        int acceder() const;
        void modificar(int v);
    private:
        int valor;
    };
}
#endif
```

```
// fichero: dato.cpp
#include "dato.hpp"
namespace umalcc {
    // Dato::~Dato() {}
    // Dato::Dato(const Dato& o) : valor(o.valor) {}
    // Dato& Dato::operator=(const Dato& o)
    // { if (this != &o) { valor = o.valor; }
    //   return *this;
    // }
    Dato::Dato()
        : valor(0) {}
    Dato::Dato(int v)
        : valor(v) {}
    int Dato::acceder() const
    {
        return valor;
    }
    void Dato::modificar(int v)
    {
        valor = v;
    }
}
```

EJEMPLO: TAD BOLSA de DATOS

- Diseñe un TAD `BolsaDato` que contiene una colección de elementos homogéneos (de tipo `Dato`), y defina los siguientes métodos:
  - Destructor: destruye y libera los recursos asociados a la bolsa
  - Constructor por defecto: construye una bolsa vacía
  - Constructor de copia: copia el contenido de otra bolsa
  - Operador de asignación: asigna el contenido de otra bolsa
  - Size: método constante que devuelve el número de elementos almacenados en la bolsa
  - Clear: elimina los elementos almacenados, dejando la bolsa vacía
  - Añadir: añade el dato recibido como parámetro a la bolsa
  - Eliminar: elimina un elemento de la bolsa que sea igual al valor del dato recibido como parámetro
  - Pertenece: método constante que devuelve `true` si el dato recibido como parámetro se encuentra en la bolsa, y `false` en cualquier otro caso
  - Cambiar: cambia el valor de todos los elementos de la bolsa que son iguales al primer dato recibido como parámetro por el valor del segundo dato recibido como parámetro
  - Extraer: elimina de la bolsa todos los elementos iguales al valor del dato recibido como parámetro, y devuelve la cuenta del número de elementos eliminados

TAD BOLSA de DATOS (I) [bolsa_dato.hpp]

```
#ifndef bolsa_dato_hpp_
#define bolsa_dato_hpp_
#include "dato.hpp"
namespace umalcc {
    class BolsaDato {
    public:
        ~BolsaDato();
        BolsaDato();
        BolsaDato(const BolsaDato& o);
        BolsaDato& operator=(const BolsaDato& o);
        int size() const;
        void clear();
        void anyadir(const Dato& d);
        void eliminar(const Dato& d);
        bool pertenece(const Dato& d) const;
        void cambiar(const Dato& d1,
                     const Dato& d2);
        int extraer(const Dato& d);
    private:
        //-- Tipos --
        struct Nodo;
        typedef Nodo* PNodo;
        struct Nodo {
            PNodo sig;
            Dato d;
        };
        //-- Métodos Privados --
        void destruir(PNodo& l) const;
        PNodo duplicar(PNodo l) const;
        void insertar_principio(PNodo& l,
                                const Dato& d) const;
        void eliminar_elm(PNodo& l, const Dato& d,
                          bool& ok) const;
        PNodo buscar_elm(PNodo l, const Dato& d) const;
        void reemplazar(PNodo& l, const Dato& d1,
                        const Dato& d2) const;
        int purgar(PNodo& l, const Dato& d) const;
        //-- Atributos --
        int sz;
        PNodo lista;
    };
}
#endif
```

TAD BOLSA de DATOS (II) [bolsa_dato.cpp]

```
#include "bolsa_dato.hpp"
#include <cstddef>
namespace umalcc {
    BolsaDato::~BolsaDato()
    {
        destruir(lista);
    }
    BolsaDato::BolsaDato()
        : sz(0), lista(NULL) {}
    BolsaDato::BolsaDato(const BolsaDato& o)
        : sz(o.sz), lista(duplicar(o.lista)) {}
    BolsaDato&
      BolsaDato::operator=(const BolsaDato& o)
    {
        if (this != &o) {
            destruir(lista);
            sz = o.sz;
            lista = duplicar(o.lista);
        }
        return *this;
    }
    void BolsaDato::clear()
    {
        destruir(lista);
        sz = 0;
    }
    int BolsaDato::size() const
    {
        return sz;
    }
    void BolsaDato::anyadir(const Dato& d)
    {
        insertar_principio(lista, d);
        ++sz;
    }
    void BolsaDato::eliminar(const Dato& d)
    {
        bool ok;
        eliminar_elm(lista, d, ok);
        if (ok) {
            --sz;
        }
    }
    bool BolsaDato::pertenece(const Dato& d) const
    {
        PNodo p = buscar_elm(lista, d);
        return p != NULL;
    }
    void BolsaDato::cambiar(const Dato& d1,
                            const Dato& d2)
    {
        reemplazar(lista, d1, d2);
    }
    int BolsaDato::extraer(const Dato& d)
    {
        int cnt = purgar(lista, d);
        sz -= cnt;
        return cnt;
    }
```

TAD BOLSA de DATOS (III) [bolsa_dato.cpp]

```
void BolsaDato::destruir(PNodo& l) const
{
    while (l != NULL) {
        PNodo p = l;
        l = l->sig;
        delete p;
    }
}
BolsaDato::PNodo
  BolsaDato::duplicar(PNodo l) const
{
    PNodo r = NULL;
    if (l != NULL) {
        r = new Nodo;
        r->d = l->d;
        PNodo u = r;
        PNodo p = l->sig;
        while (p != NULL) {
            u->sig = new Nodo;
            u->sig->d = p->d;
            u = u->sig;
            p = p->sig;
        }
        u->sig = NULL;
    }
    return r;
}
```

```
void BolsaDato::eliminar_elm(PNodo& l, const Dato& d,
                             bool& ok) const
{
    ok = false;
    if (l != NULL) {
        if (d.acceder() == l->d.acceder()) {
            PNodo p = l;
            l = l->sig;
            delete p;
            ok = true;
        } else {
            PNodo ant = l;
            PNodo act = l->sig;
            while ((act != NULL)&&
                   (d.acceder()!=act->d.acceder())){
                ant = act;
                act = act->sig;
            }
            if (act != NULL) {
                ant->sig = act->sig;
                delete act;
                ok = true;
            }
        }
    }
}
```

TAD BOLSA de DATOS (IV) [bolsa_dato.cpp]

```
void BolsaDato::insertar_principio(PNodo& l, const Dato& d) const
{
    PNodo p = new Nodo;
    p->d = d;
    p->sig = l;
    l = p;
}
BolsaDato::PNodo BolsaDato::buscar_elm(PNodo l, const Dato& d) const
{
    PNodo p = l;
    while ((p != NULL)&&(d.acceder() != p->d.acceder())) {
        p = p->sig;
    }
    return p;
}
void BolsaDato::reemplazar(PNodo& l, const Dato& d1, const Dato& d2) const
{
    PNodo p = l;
    while (p != NULL) {
        if (d1.acceder() == p->d.acceder()) {
            p->d.modificar(d2.acceder()); // equivalente a: p->d = d2;
        }
        p = p->sig;
    }
}
```

TAD BOLSA de DATOS (V) [bolsa_dato.cpp]

```
    int BolsaDato::purgar(PNodo& l, const Dato& d) const
    {
        int cnt = 0;
        while ((l != NULL)&&(d.acceder() == l->d.acceder())) {
            PNodo ptr = l;
            l = l->sig;
            delete ptr;
            ++cnt;
        }
        if (l != NULL) {
            PNodo ant = l;
            PNodo act = l->sig;
            while (act != NULL) {
                if (d.acceder() == act->d.acceder()) {
                    ant->sig = act->sig;
                    delete act;
                    ++cnt;
                } else {
                    ant = act;
                }
                act = ant->sig;
            }
        }
        return cnt;
    }
}
```

TAD BOLSA de DATOS (VI) [main.cpp]

```
#include <iostream>
#include "bolsa_dato.hpp"
using namespace std;
using namespace umalcc;
void comprobar_1(const BolsaDato& b)
{
    if (b.size() != 5) {
        cout << "Error en tamaño" << endl;
    }
    for (int i = 1; i <= 4; ++i) {
        if (! b.pertenece(Dato(i))) {
            cout << "Error: elemento " << i
                 << endl;
        }
    }
}
void comprobar_2(const BolsaDato& b)
{
    if (b.size() != 3) {
        cout << "Error en tamaño" << endl;
    }
    for (int i = 4; i <= 6; ++i) {
        if (! b.pertenece(Dato(i))) {
            cout << "Error: elemento " << i
                 << endl;
        }
    }
}
```

```
int main()
{
    BolsaDato b1;
    for (int i = 1; i <= 4; ++i) {
        b1.anyadir(Dato(i));
    }
    b1.anyadir(Dato(2));

    BolsaDato b2 = b1;
    comprobar_1(b1);
    comprobar_1(b2);
    b2.cambiar(Dato(2), Dato(5));
    b2.cambiar(Dato(3), Dato(6));
    b2.eliminar(Dato(5));
    b2.eliminar(Dato(1));
    comprobar_2(b2);

    b2 = b1;
    comprobar_1(b2);
    cout << "Fin de BolsaDato" << endl;

    // Falta prueba de extraer
}
```

Tema 3: GESTIÓN DE MEMORIA DINÁMICA