Tema 2

Memoria Dinámica

2.1 Datos estáticos y dinámicos

Datos estáticos: su tamaño y forma es constante durante la ejecución de un programa y por tanto se determinan en tiempo de compilación. El ejemplo típico son los arrays. Tienen el problema de que hay que dimensionar la estructura de antemano, lo que puede conllevar desperdicio o falta de memoria.

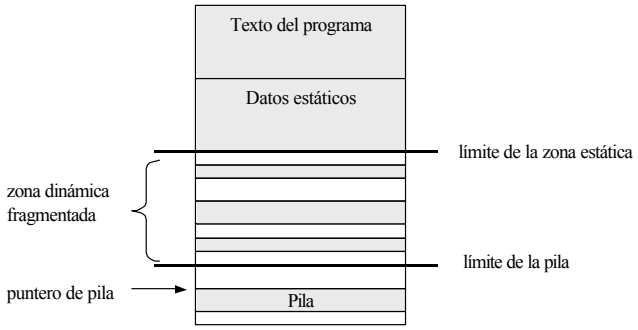
Datos dinámicos: su tamaño y forma es variable (o puede serlo) a lo largo de un programa, por lo que se crean y destruyen en tiempo de ejecución. Esto permite dimensionar la estructura de datos de una forma precisa: se va asignando memoria en tiempo de ejecución según se va necesitando.

Cuando el sistema operativo carga un programa para ejecutarlo y lo convierte en proceso, le asigna cuatro partes lógicas en memoria principal: texto, datos (estáticos), pila y una zona libre. Esta zona libre (o *heap*) es la que va a contener los datos dinámicos, la cual, a su vez, en cada instante de la ejecución tendrá partes asignadas a los mismos y partes libres que fragmentarán esta zona, siendo posible que se agote si no se liberan las partes utilizadas ya inservibles. (La pila también varía su tamaño dinámicamente, pero la gestiona el sistema operativo, no el programador):

Para trabajar con datos dinámicos necesitamos dos cosas:

1. Subprogramas predefinidos en el lenguaje que nos permitan gestionar la memoria de forma dinámica (asignación y liberación).

2. Algún tipo de dato con el que podamos acceder a esos datos dinámicos (ya que con los tipos vistos hasta ahora sólo podemos acceder a datos con un tamaño y forma ya determinados).



2.2 Tipo puntero

Las variables de tipo puntero son las que nos permiten referenciar datos dinámicos. Tenemos que diferenciar claramente entre:

1. la variable referencia o apuntadora, de tipo puntero;
2. la variable anónima referenciada o apuntada, de cualquier tipo, tipo que estará asociado siempre al puntero.

Físicamente, un puntero no es más que una dirección de memoria. En el siguiente ejemplo se muestra el contenido de la memoria con un puntero que apunta a la dirección $78AC_{(16}$, la cual contiene $6677_{(16}$:

Antes de definir los punteros, vamos a recordar cómo puede darse nombre a tipos de datos propios utilizando la palabra reservada `typedef`. El uso de `typedef` (definición de tipo) en C++ permite definir un nombre par un tipo de datos en C++. La declaración de `typedef` es similar a la declaración de una variable. La forma es: `typedef tipo nuevo-tipo;`.

Ejemplos:

```
typedef char LETRA;
LETRA caracter;
typedef enum luces_semaforo {Rojo, Amarillo, Verde} estado_luces;
estado_luces semaforo;
typedef int vector_de_20[20];
vector_de_20 mivector;
```

Introduciremos las declaraciones de tipo antes de las declaraciones de variables en el código.

Definiremos un tipo puntero con el carácter asterisco `(*)` y especificando siempre el tipo de la variable referenciada. Ejemplo:

```
typedef int *PtrInt;    // puntero a enteros
PtrInt p;               // puntero a enteros
```

O bien directamente:

```
int *p; // puntero a enteros
```

Cuando p esté apuntando a un entero de valor -13, gráficamente lo representaremos así:

Para acceder a la variable apuntada hay que hacerlo a través de la variable puntero, ya que aquélla no tiene nombre (por eso es anónima). La forma de denotarla es `*p`. En el ejemplo `*p = -13` (y `p` = dirección de memoria de la celda con el valor -13, dirección que no necesitamos tratar directamente).

El tipo registro o estructura en C++: `struct`. Una declaración de estructura define una variable estructura e indica una secuencia de nombres de variables -denominadas miembros de la estructura o campos del registro- que pueden tener tipos diferentes. La forma básica es:

```
struct identificador_del_registro {
    tipo identificador_1;
    tipo identificador_2;
    .
    .
    .
```

```
    tipo identificador_3;
};
```

Ejemplo:

```
struct resistencias {
    char fabricante[20];    // Fabricante de la resistencia
    int cantidad;           // Numero de resistencias
    float precio_unitario;  // Precio de cada resistencia
};
```

El acceso a los miembros de la estructura se realiza mediante el operador punto `(.)`.

Ejemplo:

```
resistencias.cantidad=20;
gets(resistencias.fabricante);
if (resistencias.precio_unitario==50.0)
```

Podemos crear una plantilla de estructura para más tarde declarar variables de ese tipo. Ejemplo:

```
struct registro_resistencias {
    char fabricante[20];    // Fabricante de la resistencia
    int cantidad;           // Numero de resistencias
    float precio_unitario;  // Precio de cada resistencia
};
registro_resistencias resistencias;
```

Por último, podemos usar `typedef`:

```
typedef struct unaResistencia {
    char fabricante[20];    // Fabricante de la resistencia
    int cantidad;           // Numero de resistencias
    float precio_unitario;  // Precio de cada resistencia
}registro_resistencias;
registro_resistencias resistencias;
```

Ejemplo de punteros a estructuras:

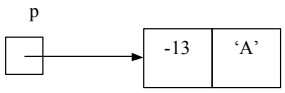
```
struct TipoRegistro {
    int num;
```

```
    char car;
};
typedef TipoRegistro *TipoPuntero;
TipoPuntero p;
```

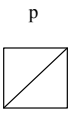
Así:

- `p` es la dirección de un registro con dos campos (tipo puntero)
- `*p` es un registro con dos campos (tipo registro)
- `(*p).num` es una variable simple (tipo entero)
- `p->num` es una variable simple (tipo entero)
- `(*p).car` es una variable simple (tipo carácter)
- `p->car` es una variable simple (tipo carácter)
- `& x` es la dirección de una variable x, siendo x, por ejemplo `int x;`.

Gráficamente:

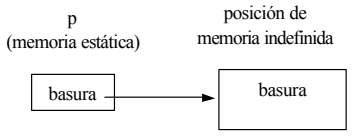


Si deseamos que una variable de tipo puntero en un momento determinado de la ejecución del programa no apunte a nada, le asignaremos la macro `NULL` (`p = NULL`) y gráficamente lo representaremos:



2.3 Gestión de la memoria dinámica

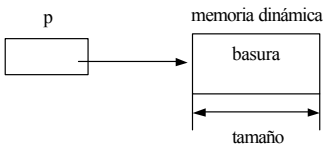
Cuando declaramos una variable de tipo puntero, por ejemplo `int *p;` estamos creando la variable `p`, y se le reservará memoria -estática- en tiempo de compilación; pero la variable referenciada o anónima no se crea. En este momento tenemos:



La variable anónima debemos crearla después mediante una llamada a un procedimiento de asignación de memoria -dinámica- predefinido. El operador `new` asigna un bloque de memoria que es el tamaño del tipo del dato apuntado por el puntero. El dato u objeto dato puede ser un `int`, un `float`, una estructura, un array o, en general, cualquier otro tipo de dato. El operador `new` devuelve un puntero, que es la dirección del bloque asignado de memoria. El formato del operador `new` es:

```
puntero = new nombreTipo (inicializado opcional);
```

Así: `p = new int;` donde `p` es una variable de tipo puntero a entero. En tiempo de ejecución, después de la llamada a este operador, tendremos ya la memoria (dinámica) reservada pero sin inicializar:



Para saber el tamaño necesario en bytes que ocupa una variable de un determinado tipo, dispondremos también de una función predefinida: `sizeof(Tipo)` que nos devuelve el número de bytes que necesita una variable del tipo de datos Tipo. Podemos usar este operador con datos más complejos:

```
struct TipoRegistro { int num; char car; };
typedef TipoRegistro *TipoPuntero;
TipoPuntero p = new TipoPuntero;
```

Tras usar la memoria asignada a un puntero, hay que liberarla para poder utilizarla con otros fines y no dejar datos inservibles en el mapa de la memoria dinámica. Se libera la memoria asignada a un puntero `p`, cuyo tipo ocupa `t` bytes, también dinámicamente, mediante el operador `delete`: `delete p;`.

2.4 Operaciones con punteros

Al definir una variable de tipo puntero, implícitamente podemos realizar las siguientes operaciones con ella:

1. Acceso a la variable anónima
2. Asignación
3. Comparación
4. Paso como parámetros

Utilizaremos el siguiente ejemplo para ver estas operaciones:

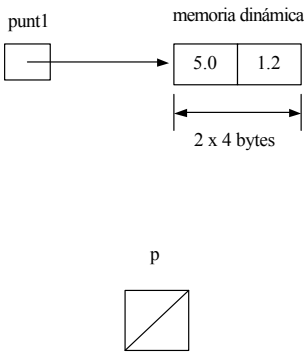
```
struct PartesComplejas {
    float ParteReal;
    float ParteCompleja;
};
typedef PartesComplejas *PunteroAComplejo;

int main()
{
    PunteroAComplejo punt1, punt2;

    punt1 = new PartesComplejas;
    punt2 = new PartesComplejas;
    ...
```

El acceso a la variable anónima se consigue de la forma ya vista con el operador `*`, y la asignación se realiza de la misma forma que en los tipos simples. En el ejemplo, suponiendo que punt1 ya tenga memoria reservada, podemos asignar valores a los campos de la variable anónima de la siguiente forma:

```
    punt1->PartReal = 5.0;
    punt1->PartImag = 1.2;
```

Tras las asignaciones tendremos (suponiendo que un real ocupara 4 bytes):

A una variable puntero también se le puede asignar la macro `NULL` (sea cual sea el tipo de la variable anónima): `punt = NULL;`, tras lo cual tendremos:



Se puede asignar el valor de una variable puntero a otra siempre que las variables a las que apuntan sean del mismo tipo: `punt2 = punt1;`

La comparación de dos punteros, mediante los operadores != y ==, permite determinar si apuntan a la misma variable referenciada:

```
if (punt1 == punt2) // expresion logica
```

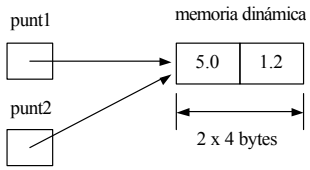
o si no apuntan a nada:

```
if (punt1 != NULL) // expresion logica
```

El paso de punteros como parámetros permite el paso de variables por referencia a funciones. En la función `llamame()` del siguiente ejemplo el parámetro `p` es un puntero a un número entero; en la función `main()` del ejemplo se declara un entero `x` y se pasa su dirección de memoria a la función `llamame()` usando, para ello, el operador `&`. El funcionamiento de la llamada sería equivalente al uso de un parámetro por referencia en la función `llamame()` mediante el operador `&` y la llamada desde la función `main()` del modo `llamame(x)`. Lo que ocurre, realmente, es un paso por dirección en la llamada a la función `llamame()`. Aunque puede usarse en C++ el paso de parámetros por dirección (con parámetros formales que son punteros) no resulta nada recomendable, aconsejándose el uso del paso de parámetros por refencia (mediante el operador `&` en los parámetros formales) como se vio en los temas anteriores.

Ejemplo:

```
#include <iostream>

using namespace std;

void llamame(int *p);
```




```
int main()
{
    int x = 0;

    cout << "El valor de x es " << x << endl;
    llamame(&x);
    cout << "El nuevo valor de x es " << x << endl;

    return 0;
}

void llamame(int *p)
{
    *p=5;
}
```

2.5 Listas enlazadas

Lo que realmente hace de los punteros una herramienta potente es la circunstancia de que pueden apuntar a variables que a su vez contienen punteros. Los punteros así creados sí son variables dinámicas, a diferencia de los declarados explícitamente, los cuales son estáticos porque pueden crearse en tiempo de compilación.

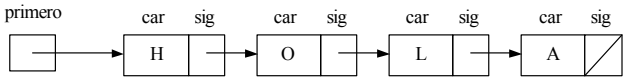
Cuando en cada variable anónima o nodo tenemos un solo puntero que apunta al siguiente nodo tenemos una lista enlazada.

Ejemplo: cadenas de caracteres de longitud variable.

```
struct TpNodo{
    char car;           // Caracter
    TpNodo *sig;   // Al siguiente nodo
};
```

```
TpNodo *primero;
```

Con estos tipos de datos podemos crear estructuras que no malgastan la memoria, y que sí malgastaría un array de longitud fija:



Un algoritmo que crea una lista enlazada similar sería:

```
#include <iostream>
#include <cstdlib>

using namespace std;

struct TpNodo {
    int dato;
    TpNodo *sig;
};

typedef TpNodo *LISTA;

void mostrar_lista(const LISTA ptr);
void insertar(LISTA &ptr, const int elemento);

int main() {
    LISTA n1 = NULL;
    int elemento;

    do
    {
        cout << endl << "Introduzca elemento: ";
        cin >> elemento;
        if(elemento != 0)
            insertar(n1, elemento);
    } while(elemento != 0);
```

```
    cout << endl << "La nueva lista enlazada es: ";
    mostrar_lista(n1);

    return 0;
}

void mostrar_lista(const LISTA ptr) {
    while(ptr != NULL)
    {
        cout << ptr->dato << " ";
        ptr = ptr->sig;
    }
    cout << endl;
}

void insertar(LISTA &ptr, const int elemento) // Al final de la lista {
    LISTA p1, p2;

    p1 = ptr;
    if (p1 == NULL) // Lista vacia
    {
        p1 = new TipoNodo;
        p1->dato = elemento;
        p1->sig = NULL;
        ptr = p1;
    }
    else
    {
        while(p1->sig != NULL)
            p1 = p1->sig;
        p2 = new TipoNodo;
        p2->dato = elemento;
        p2->sig = NULL;
        p1->sig = p2;
    }
```

```
}
```

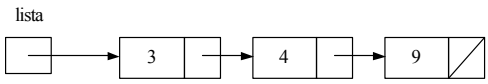
2.5.1 Operaciones básicas sobre listas enlazadas

1. Inserción de un nodo al principio
2. Insertar un nodo en una lista enlazada ordenada
3. Eliminar el primer nodo
4. Eliminar un nodo determinado

Vamos a basarnos en las declaraciones de una lista enlazada de enteros:

```
struct TipoNodo{
    int dato;           // Elemento entero
    TipoNodo *sig;   // Al siguiente nodo
};

typedef TipoNodo *TipoLista;

TipoLista lista; // Cabeza de la lista
TipoLista nodo;  // Nuevo nodo a insertar
TipoLista ptr;   // Puntero auxiliar
```

Y partiremos del estado inicial de la lista:



Inserción de un nodo al principio

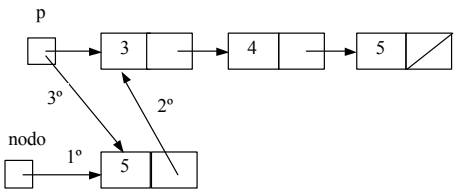
Los pasos a dar son los siguientes:

1. Reservar memoria para el nuevo nodo:
   `nodo = new TipoNodo;`
2. Asignar valor nuevo al nuevo nodo:
   `nodo->dato = 5; // por ejemplo`

3. Enlazar la lista al siguiente del nuevo nodo:
   `nodo->sig = lista;`

4. Actualizar la cabeza de la lista: `lista = nodo;`

Gráficamente (la línea discontinua refleja el enlace anterior a la operación, y también se especifica el orden de asignación de valores a los punteros):



Inserción de un nodo en una lista ordenada (ascendentemente)

Los pasos a seguir son los siguientes:

1. Reservar memoria para el nuevo nodo:
   `nodo = new TipoNodo;`

2. Asignar valor nuevo al nuevo nodo:
   `nodo->dato = 5; // por ejemplo`

3. Si la lista está vacía o el dato nuevo es menor que el de la cabeza, el nodo se inserta al principio y se actualiza la cabeza lista para que apunte al nuevo nodo.

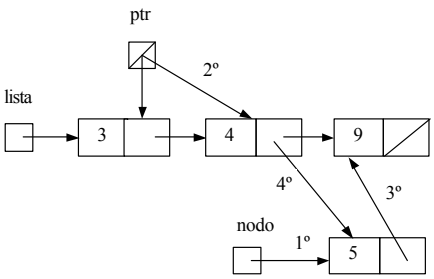
4. Si la lista no está vacía y el dato nuevo es mayor que el que hay en la cabeza, buscamos la posición de inserción usando un bucle del tipo:

```
while ((ptr->sig != NULL) &&
       (nodo->dato > (ptr->sig)->dato))
    ptr = ptr->sig;
 /* ptr queda apuntando al predecesor */
```

   y enlazamos el nodo:

```
nodo->sig = ptr->sig;   // con el sucesor
ptr->sig = nodo;        // con el predecesor
```

Gráficamente (observar el orden de asignación de punteros -el segundo paso son las sucesivas asignaciones al puntero `ptr`):



Eliminar el primer nodo

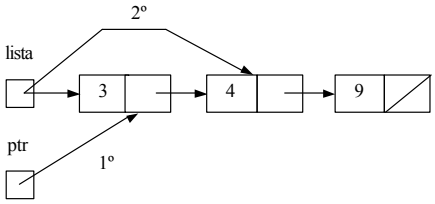
Los pasos son (suponemos lista no vacía):

1. Guardar el puntero a la cabeza actual:
   `ptr = lista;`

2. Avanzar una posición la cabeza actual:
   `lista = lista->sig;`

3. Liberar la memoria del primer nodo:
   `delete ptr;`

Gráficamente:



Eliminar un nodo determinado

Vamos a eliminar el nodo que contenga un valor determinado. Los pasos son los siguientes (suponemos lista no vacía):

Caso a) El dato coincide con el de la cabeza: Se elimina como en la operación anterior.

Caso b) El dato no coincide con el de la cabeza:

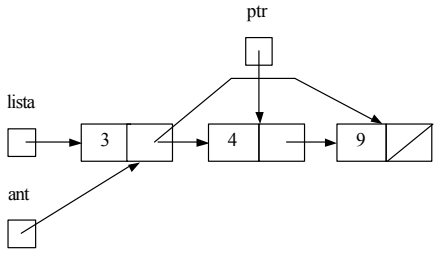
1. Se busca el predecesor y se almacena en una variable de tipo puntero, por ejemplo, en `ant`. En otra variable, `ptr`, se almacena el nodo actual:

```
ant = lista;
ptr = lista->sig;
while ((ptr != NULL) &&
         (ptr->dato != valor)){
   ant = ptr;
   ptr = ptr->sig;
}
```

2. Se comprueba si se ha encontrado, en cuyo caso se enlaza el predecesor con el sucesor del actual:

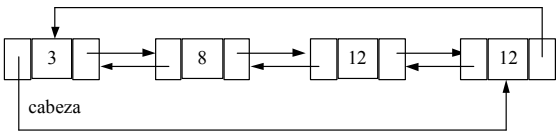
```
if (ptr != NULL){ // encontrado
   ant->sig = ptr->sig;
   delete ptr;
   // ptr = NULL; //Opcional. No seria necesario
}
```

Gráficamente quedaría (eliminando el nodo de valor 4):

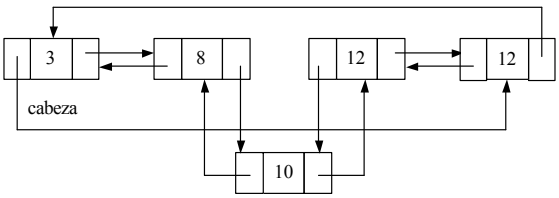


2.6 Listas doblemente enlazadas

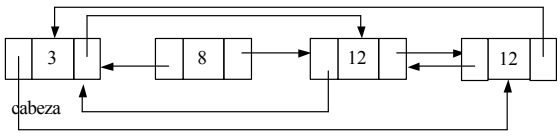
Hasta ahora, el recorrido de la lista se realizaba en sentido directo (hacia delante). En algunos casos puede realizarse en sentido inverso (hacia atrás). Sin embargo, existen numerosas aplicaciones en las que es conveniente poder acceder a los elementos o nodos de una lista en cualquier orden. En este caso se recomienda el uso de una lista doblemente enlazada. En tal lista, cada elemento contiene dos punteros, aparte del valor almacenado en el elemento. Un puntero apunta al siguiente elemento de la lista y el otro puntero apunta al elemento anterior. Ejemplo gráfico (lista doblemente enlazada circular):



Existe una operación de insertar y eliminar (borrar) en cada dirección. En caso de eliminar un nodo de una lista doblemente enlazada es preciso cambiar dos punteros. Por ejemplo, para insertar un nodo a la derecha del nodo actual tenemos que asignar cuatro nuevos punteros, como puede comprobarse gráficamente:



Gráficamente, borrar significaría:



Una lista doblemente enlazada con valores de tipo int necesita dos punteros y el valor del campo datos:

```
struct TipoNodo{
    int dato;
    TipoNodo *siguiente;
```

```
    TipoNodo *anterior;
};
```

Ejercicios

1. Implementa las siguientes operaciones adicionales con listas enlazadas:

   1.1 `void imprimir_elementos(const TipoLista lista);`
   `/* Imprime por pantalla todos los elementos de una lista */`

   1.2 `void copiar_lista(const TipoLista listaOrg, TipoLista &listaDes);`
   `/* Copia la lista listaOrg en la lista listaDes */`

   1.3 `int longitud_lista(const TipoLista lista);`
   `/* Devuelve la longitud de la lista */`

   1.4 `void eliminar_ultimo(TipoLista &lista);`
   `/* Elimina el último elemento de la lista */`

   1.5 `bool en_lista(const TipoLista lista, const int dato);`
   `/* Devuelve true si el dato está en la lista */`

   1.6 `void insertar_tras(TipoLista &lista, const int dato, const int dato_ant);`
   `/* Inserta "dato"tras "dato_ant */`

   1.7 `void purgar_dup(TipoLista &lista);`
   `/* Elimina elementos duplicados en una lista */`

   1.8 `void ordenar_lista(TipoLista &lista);`
   `/* Ordena la lista de menor a mayor */`

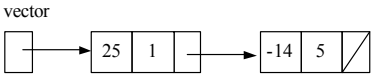
   Suponer la siguiente declaración de tipos:

```
struct TipoNodo{
    int dato;
    TipoNodo *sig;
};
typedef TipoNodo *TipoLista;
```

2. Realizar todas las operaciones del ejercicio 1 con listas doblemente enlazadas. Supondremos la siguiente declaración de tipos:

```
struct TipoNodo{
    int dato;
    TipoNodo *sig;
    TipoNodo *ant;
};
typedef TipoNodo *TipoListaDob;
```

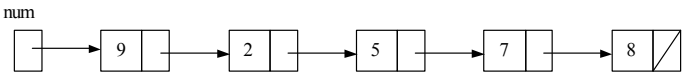
3. Hay muchas aplicaciones en las que se debe almacenar en la memoria un vector de grandes dimensiones. Si la mayoría de los elementos del vector son ceros, éste puede representarse más eficientemente utilizando una lista enlazada con punteros, en la que cada nodo es un registro con tres campos: el dato en esa posición, si es distinta de cero, el índice de esa posición y un puntero al siguiente nodo. Por ejemplo:



Esto indica que hay sólo dos elementos distintos de cero en el vector, es decir, éste es: (25, 0, 0, 0, -14, 0, 0) si consideramos los vectores con siete posiciones.

Escribe un programa que lea dos vectores por teclado, los introduzca en listas enlazadas y calcule su producto escalar. Diseña también las funciones para insertar y consultar el valor en una determinada posición del vector, teniendo en cuenta que si introducimos un elemento en una posición previamente existente se debe eliminar el valor anterior y sustituirlo por el nuevo.

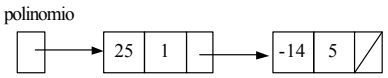
4. Una forma de almacenar un número natural de valor mayor que el permitido en una computadora es introducir cada dígito en un nodo de una lista enlazada. Por ejemplo, la siguiente lista representa al número 92578:



Escribe una función que tome como parámetro un puntero a una lista enlazada y devuelva el número correspondiente en una variable de tipo unsigned int. Diseña también una función que lea por teclado una sucesión de dígitos (caracteres) y los introduzca como dígitos (naturales) en una lista enlazada, y otras dos funciones

que realicen, respectivamente, la suma y el producto de números representados de esta forma.

5. Un polinomio en x de un grado arbitrario se puede representar mediante una lista enlazada con punteros, donde cada nodo contiene el coeficiente y el exponente de un término del polinomio, más un puntero al siguiente nodo. Por ejemplo el polinomio: $25x - 14x^5$ se puede representar como:



   5.1 Escribe una función que evalúe un polinomio P en un x = valor.
   `int evaluar (const TipoPolinomio p, const int valor);`

   5.2 Escribe una función que devuelva el coeficiente del término de grado `i` de un polinomio `P`.
   `int obtener (const TipoPolinomio p, const int i);`

   5.3 Escribe una función que sume 2 polinomios `P1` y `P2`:
   `TipoPolinomio sumar(const TipoPolinomio p1, const TipoPolinomio p2);`

   5.4 Escribe una función que realice la derivada de un polinomio P:
   `TipoPolinomio derivada (const TipoPolinomio p);`

6. Supongamos el tipo Conjunto definido mediante una lista enlazada según la siguiente cabecera:

```
struct TipoNodo{
    int dato;
    TipoNodo *sig;
};
typedef TipoNodo *Conjunto;
```

   Diseñar las siguientes operaciones para la intersección y unión de conjuntos:

```
Conjunto interseccion (const Conjunto c1, const Conjunto c2);
Conjunto union(const Conjunto c1, const Conjunto c2);
```