

ESTRUCTURAS DE DATOS

Gestión Dinámica de Memoria

Funciones New - Malloc - Realloc - delete - free

Ing Yamil Armando Cerquera Rojas - yacerque@gmail.com
Especialista en Sistemas Universidad Nacional
Docente Universidad Surcolombiana
Neiva - Huila

Contenido

<i>Introducción</i>	2
<i>El tipo de datos Puntero</i>	2
<i>DEFINICIÓN:</i>	3
<i>Declaración de variables puntero</i>	4
<i>Asignación errónea: "Cannot assign..."</i>	5
<i>Operaciones con Punteros</i>	6
<i>Operadores específicos de punteros</i>	7
<i>Diferencia entre punteros y variables:</i>	8
<i>Asignación de Punteros</i>	9
<i>Comparación de Punteros</i>	10
<i>Asignación Dinámica de Memoria</i>	10
<i>Funciones de reserva de memoria: new</i>	11
<i>Función de liberación de memoria dinámica: delete</i>	12
<i>Ejemplo de asignación dinámica de memoria</i>	13
<i>Punteros a Estructuras</i>	13
<i>Listas encadenadas o enlazadas</i>	15
<i>Operaciones básicas sobre listas enlazadas</i>	16
<i>Otras clases de listas enlazadas</i>	23
<i>Ejercicios</i>	27
<i>Otras funciones de reserva de memoria de forma dinámica.</i>	30
<i>Función de liberación de memoria dinámica: free() .</i>	33
<i>Creación Dinámica de Arrays y Aritmética de Punteros.</i>	34

Introducción

En este tema se estudiarán las posibilidades que ofrece el Lenguaje C a la hora de trabajar dinámicamente con la memoria dentro de los programas, esto es, reservar y liberar bloques de memoria al momento de ejecutar un programa.

Además en este tema se introducirá el concepto de tipo abstracto de dato y la forma de dividir un gran programa en otros más pequeños.

Los tipos de datos vistos hasta ahora, tanto los simples (*Predefinidos por el lenguaje*) como los estructurados (*Definidos por el programador*), sirven para describir datos o estructuras de datos cuyos tamaños y formas se conocen de antemano. Cuando se declara una variable se reserva la memoria suficiente para contener la información que debe almacenar. Esta memoria permanece asignada a la variable hasta que termine la ejecución del programa (fin de la función main). Sin embargo, hay programas cuyas estructuras de datos pueden variar de forma y tamaño durante la existencia del mismo (En modo ejecución).

Las variables de todos los tipos de datos vistos son variables *estáticas*, en el sentido de que se declaran en el programa, se designan por medio del identificador declarado (variable), y se reserva para ellas un espacio en memoria en tiempo de compilación. El contenido de la variable podrá cambiar durante la ejecución del programa, pero no el tamaño de memoria reservado para determinada variable.

En ocasiones el tamaño de los objetos no se conoce hasta el momento de la compilación. Por ejemplo, la longitud de una cadena de caracteres que introducirá el usuario no se conoce hasta el tiempo de ejecución. El tamaño de un arreglo puede depender de un parámetro cuyo valor se desconoce previo al momento de ejecución. Ciertas estructuras de datos como listas enlazadas, pilas y colas utilizan memoria dinámica.

C++, ofrece la posibilidad de crear o destruir variables en tiempo de ejecución del programa, a medida que van siendo necesitadas durante la ejecución del mismo. Puesto que estas variables no son declaradas en el programa, no tienen nombre y se denominan *variables anónimas*. Si un lenguaje permite la creación de variables anónimas, debe también proporcionar una forma de referirse a estas variables, de modo que se les pueda asignar valores. Del mismo modo, debe proporcionar una forma de acceder a estas. Para ello C++ proporciona el tipo *Puntero*.

El tipo de datos Puntero

El tipo puntero y las variables declaradas de tipo puntero se comportan de forma diferente a las variables que se han estudiado en temas anteriores. Hasta ahora cuando se declaraba una variable de un determinado tipo, dicha variable podía contener *directamente* un valor de dicho tipo. Con el tipo puntero esto no es así.

Los punteros proporcionan la mayor parte de la potencia al C y C++, y marcan la principal diferencia con otros lenguajes de programación.

Una buena comprensión y un buen dominio de los punteros pondrán en sus manos una herramienta de gran potencia. Un conocimiento mediocre o incompleto te impedirá desarrollar programas eficaces.

Por eso se le dedicará especial atención y mucho espacio a los punteros. Es muy importante comprender bien cómo funcionan y cómo se usan.

Para entender qué es un puntero se da un repaso primero cómo se almacenan los datos en un ordenador.

La memoria de un ordenador está compuesta por unidades básicas llamadas bits. Cada bit sólo puede tomar dos valores, normalmente denominados alto y bajo, ó 1 y 0 (Estados lógicos). Pero trabajar con bits no es práctico, y por eso se agrupan.

Cada grupo de 8 bits forma un byte u octeto. En realidad el microprocesador, y por lo tanto el programa, sólo puede manejar directamente bytes o grupos de dos o cuatro bytes. Para acceder a los bits hay que acceder antes a los bytes. Y aquí se llega al asunto, cada byte tiene una dirección, llamada normalmente dirección de memoria (compuesta por un segmento y un desplazamiento).

La unidad de información básica es la palabra, dependiendo del tipo de microprocesador una palabra puede estar compuesta por dos, cuatro, ocho o dieciséis bytes. Se hablará en estos casos de plataformas de 16, 32, 64 ó 128 bits. Se habla indistintamente de direcciones de memoria, aunque las palabras sean de distinta longitud. Cada dirección de memoria contiene siempre un byte (Una dirección de memoria corresponde únicamente a un byte). Lo que sucederá cuando las palabras sean de 32 bits es que se accede a posiciones de memoria que serán múltiplos de 4.

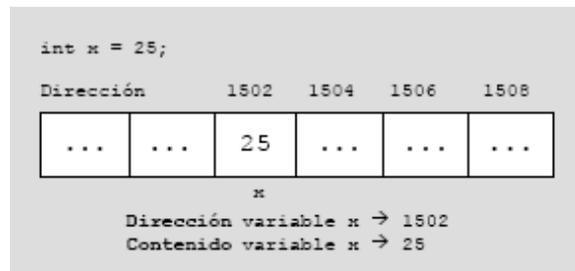
Todo esto sucede en el interior de la máquina, e interesa más bien poco. Se puede saber qué tipo de plataforma se usa averiguando el tamaño del tipo int, y para ello hay que usar el operador "sizeof()", por ejemplo:

```
cout << "Plataforma de " << 8*sizeof(int) << " bits";
```

DEFINICIÓN:

Un puntero es un tipo especial de variable que contiene, ni más ni menos que, una dirección de memoria. Por supuesto, a partir de esa dirección de memoria puede haber cualquier tipo de objeto (o dato): un char, un int, un float, un array, una estructura, una función u otro puntero. El programador será el responsables de decidir ese contenido.

En síntesis un puntero es una variable cuyo valor es la dirección de memoria de otra variable. Esto quiere decir que un puntero se refiere *indirectamente* a un valor. Por tanto, no hay que confundir una dirección de memoria con el contenido de esa dirección de memoria:



Se hace una distinción entre la variable **referencia** (*puntero*) y la variable **referenciada** por un puntero (*anónima* o *apuntada*).

Variable Referencia (Puntero): Es una variable estática, es decir se crea en tiempo de compilación.

Variable Referenciada (Anónima): Es una variable dinámica creada en tiempo de ejecución, que únicamente puede ser accedida a través de un puntero.

Una variable puntero no puede apuntar a cualquier variable anónima; debe apuntar a variables anónimas de un determinado tipo. El tipo de la variable anónima vendrá determinado por el tipo de la variable que la apunta. Este tipo debe ser incluido en la especificación del tipo puntero.

Declaración de variables puntero

Un puntero, como cualquier variable u objeto, además de ser **declarado** (para comenzar a existir) necesita ser **inicializado** (darle un valor de modo controlado), lo cual se realiza mediante el operador de asignación ('='). Desde que el puntero es declarado almacena un valor, el problema es que se trata de un valor aleatorio, intentar operar con un puntero sin haberlo inicializado es una frecuente causa de problemas.

Los punteros se declaran precediendo el identificador con el operador de indirección, (*), que se leerá como "puntero a".

La forma general de declarar un tipo de datos puntero es la siguiente:

```
typedef <tipo> *<identificador>;
```

Donde *tipo* es el tipo base del puntero, que puede ser cualquier tipo válido e *identificador* es el nombre del tipo de datos. Ejemplos de declaración de tipos de datos punteros son los siguientes:

Ejemplos:

```
int *entero;
char *carácter;
struct stPunto *punto;
typedef int *PunAInt; // Tipo puntero a enteros (int)
typedef char *PunACar; // Tipo puntero a caracteres (char)
```

Asignación errónea: "Cannot assign..."

En primer lugar veremos un caso simple de asignación errónea para comprender el mensaje enviado por el compilador.

```
void main()
{ int a;
  int* b;
  b = a; //Error
}
```

El programa no compila, y recibimos el mensaje de error:

```
"Cannot assign 'int' to 'int near*' en function main();
```

Los punteros siempre apuntan a un objeto de un tipo determinado, en el ejemplo anterior, "entero" siempre apuntará a un objeto de tipo "int".

Tras definir los tipos de datos, es posible definir una variable de estos tipos del modo siguiente:

```
PunAInt contador; //contador es una variable de tipo PunAInt
PunACar carácter; // carácter es una variable de tipo PunACar
```

La variable contador no contendrá un valor entero (tipo int), sino la dirección de memoria donde esté almacenado un valor de tipo int. El valor almacenado en la variable anónima de tipo int apuntada por contador será accesible a través de este puntero.

La forma:

```
<tipo>* <identificador>;
```

con el (*) junto al tipo, en lugar de junto al identificador de variable, también está permitida.

Vea algunos matices. Retomando el ejemplo anterior:

```
int *entero; equivale a:
int* entero;
```

Debe tener muy claro que "entero" es una variable del tipo "puntero a int", que **"*entero" NO es una variable de tipo "int"**.

Un puntero, puede apuntar a cualquier tipo de dato definido por el usuario, no sólo a tipos simples. Es muy importante tener en cuenta que primero debe declararse el tipo de datos al que apunta (un array, registro, ...) y después el tipo de datos puntero.

Ejemplo:

```
struct Complejo
{ float parte_real;
  float parte_imag;
};
```

```
typedef Complejo *TPComplejo; //Tipo puntero a un valor de tipo Complejo
```

Después definir el tipo TPComplejo, una variable de este tipo se definiría de la siguiente forma:

```
TPComplejo ptr1; //ptr1 es una variable de tipo TPComplejo
```

ptr1, es un puntero, por tanto contendrá una dirección de memoria; y en esa posición de memoria habrá una variable de tipo Complejo.

Por otro lado, también es posible definir variables de tipos puntero sin asociarles un nombre de tipo (declaración anónima):

```
<tipo> *<ident>;
```

Donde *tipo* es el tipo base del puntero, y puede ser cualquier tipo válido. Algunos ejemplos de declaración de variables puntero son los siguientes:

```
int *contador; // puntero a una variable de tipo entero (int)
char *character; // puntero a una variable de tipo caracter (char)
Complejo *ptr1; // ptr1 es un puntero a una variable de tipo Complejo
```

Siempre que sea posible evite las declaraciones anónimas, y asocie un nombre de tipo a las variables que declare.

NOTA: Al declarar un puntero hay que asegurarse de que su tipo base sea compatible con el tipo de objeto al que se quiera que apunte.

NOTA: Aunque no es una imposición de C, nosotros siempre declararemos el tipo de datos puntero **inmediatamente** después del tipo de datos al que apunta.

Operaciones con Punteros

Las operaciones que se pueden llevar a cabo con punteros son:

- 1) Operadores específicos de punteros
- 2) Asignaciones de punteros
- 3) Comparación de punteros

Operadores específicos de punteros

Al trabajar con punteros se emplean dos operadores específicos:

Operador de dirección: & Es un operador monario (sólo requiere un operando) que devuelve la dirección de memoria del operando.

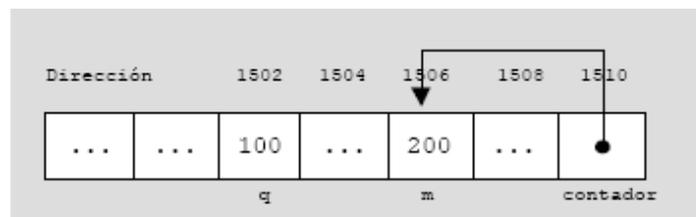
Por ejemplo,

```
typedef int *PunAInt;
int q,m;                // q y m son variables de tipo entero (int)
PunAInt contador;      // contador es un puntero a un entero (int)
q = 100;
m = 200;
contador = &m; // contador contiene la dirección de m
```

La línea de código `contador = &m`, coloca en `contador` la dirección de memoria de la variable `m` (es decir el valor 1506). No tiene nada que ver con el valor de la variable `m`.

Esta instrucción de asignación significa “*contador recibe la dirección de m*”.

Para entender mejor cómo funciona el operador de dirección `&`, la siguiente figura muestra cómo se almacenan en memoria las variables `q`, `m` y `contador`.



Por supuesto, los tipos tienen que ser "compatibles", no se puede almacenar la dirección de una variable de tipo "char" en un puntero de tipo "int".

Por ejemplo:

```
int A;
int *pA;
pA = &A;
```

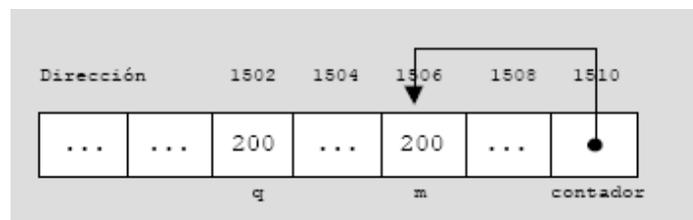
Según este ejemplo, `pA` es un puntero a `int` que apunta a la dirección donde se almacena el valor del entero identificado con `A`.

Operador de contenido o indirección: * Este operador es el complementario de &. Es un operador monario que devuelve el valor de la variable ubicada en la dirección que se especifica (contenido de la variable anónima apuntada por el puntero).

Continuando con el ejemplo anterior, si contador contiene la dirección de memoria de la variable m (posición 1506), *contador devolverá el valor 200 (el valor de la variable anónima apuntada por contador). Entonces al ejecutar la siguiente asignación:

```
q = *contador;
```

La línea de código anterior colocará el valor 200 en la variable q.



NOTA: No se debe confundir el operador * en la declaración del puntero (int *contador;) con el operador * en las instrucciones (*contador = 200;)

Diferencia entre punteros y variables:

Declarar un puntero no creará un objeto. Por ejemplo: int *entero; no crea un objeto de tipo "int" en memoria, sólo crea una variable que puede contener una dirección de memoria. Se puede decir que existe físicamente la variable "entero", y también que esta variable puede contener la dirección de un objeto de tipo "int". Observe el siguiente ejemplo:

```
int A,B;
int *entero;
...
B = 213;           /* B vale 213 */

entero = &A;      /* entero apunta a la dirección de la variable A */
*entero = 103;    /* equivale a la línea A = 103; */
B = *entero;      /* equivale a B = A; */
...
```

En este ejemplo se nota que "entero" puede apuntar a cualquier variable de tipo "int", y que puede hacer referencia al contenido de dichas variables usando el operador de indirección (*).

Como todas las variables, los punteros también contienen "basura" cuando son declaradas. Es costumbre dar valores iniciales nulos a los punteros que no apuntan a ningún sitio concreto:

```
entero = NULL;  
caracter = NULL;
```

NULL es una constante, que está definida como cero en varios ficheros de cabecera, como "cstdio" o "iostream", y normalmente vale 0L.

Asignación de Punteros

Después de declarar un puntero, pero antes de asignarle un valor, el puntero contiene un valor desconocido. Si se intenta utilizar el puntero antes de darle valor, se producirá un fallo en el programa.

Un puntero que no apunte a ninguna posición válida de la memoria ha de tener asignado el *valor nulo*, que es un *cero*. Se usa el valor nulo porque C++ garantiza que no existe ningún objeto en la dirección nula. Así, cualquier puntero que sea nulo indica que no apunta a nada y no debe ser usado.

Una forma de dar a un puntero el valor nulo es asignándole el valor cero. Por ejemplo, lo siguiente inicializa el puntero p a nulo:

```
char *p = 0;
```

Además, la mayoría de los archivos de cabecera de C++, como <stdio>, definen NULL como el valor de puntero nulo. Por eso también es posible ver la inicialización de un puntero a nulo usando la constante NULL (aunque realmente con esto le estamos asignando un 0).

```
char *p = NULL;
```

NOTA: El uso del valor nulo es simplemente un convenio que siguen los programadores C++.

No es una regla impuesta por el lenguaje C++. Tener mucho cuidado porque la siguiente secuencia no provocará ningún error de compilación, pero hará que el programa falle:

```
int *p = 0;  
*p = 10; // Incorrecto - Asignación en la posición 0
```

Es posible asignar el valor de una variable puntero a otra variable puntero, siempre que ambas sean del mismo tipo.

```
int *ptr1, *ptr2;  
ptr2 = ptr1;
```

La variable ptr2, apuntará a donde apunte ptr1. A la variable ptr2 se le asigna la dirección de memoria de ptr1. Es muy importante tener en cuenta que si ptr2, anteriormente estaba apuntando a una variable anónima (tenía un valor distinto del valor nulo), ésta será inaccesible, puesto que la forma de acceder a ella era a través del puntero que la apuntaba ptr2, y este ahora apunta a otra variable anónima (la apuntada por ptr1).

Comparación de Punteros

Es posible comparar dos variables de tipo puntero en una *expresión relacional*, usando operaciones relacionales de **igualdad** (==), **desigualdad** (!=) y **comparación** (<, >, <=, >=).

Dos variables puntero son iguales, sólo si ambas apuntan a la misma variable anónima (misma dirección de memoria) o si ambas están inicializados al valor nulo (valor 0). Los punteros a los que se apliquen estas operaciones relacionales han de ser del mismo tipo. Sin embargo, es posible comparar punteros de cualquier tipo (igualdad o desigualdad) con el valor nulo.

```
if (ptr1 < ptr2)
{ cout << "p apunta a una dirección de memoria menor que q" << endl;
}
```

Asignación Dinámica de Memoria

Hasta este momento solamente se han realizado asignaciones estáticas del programa, y más concretamente estas asignaciones estáticas no eran otras que las declaraciones de variables en el programa. Cuando se declara una variable se reserva la memoria suficiente para contener la información que debe almacenar. Esta memoria permanece asignada a la variable hasta que termine la ejecución del programa (fin de la función main). Realmente las variables locales de las funciones se crean cuando éstas son llamadas pero no se tiene control sobre esa memoria, el compilador genera el código para esta operación automáticamente.

En este sentido las variables locales están asociadas a asignaciones de memoria dinámicas, puesto que se crean y destruyen durante la ejecución del programa

Permite crear variables y arreglos de manera dinámica. Dinámica quiere decir que el espacio de memoria es asignado durante la ejecución o corrida del programa.

En ocasiones el tamaño de los objetos no se sabe hasta el momento de la ejecución. Por ejemplo, la longitud de una cadena de caracteres que introducirá el usuario no se sabe hasta el momento en que se ejecute el programa. El tamaño de un arreglo puede depender de un parámetro cuyo valor se desconoce previo al momento de ejecución. Ciertas estructuras de datos como listas enlazadas, pilas y colas utilizan memoria dinámica.

Cuando se habla de *asignación dinámica* de memoria, se refiere al hecho de crear variables anónimas, es decir reservar espacio en memoria para estas variables en tiempo de ejecución, y también a liberar el espacio reservado para una variable anónima en tiempo de ejecución, en caso de que dicha variable ya no sea necesaria.

La zona de la memoria donde se reservan espacios para asignarlos a variables dinámicas se denomina HEAP o montón. Puesto que esta zona tiene un tamaño limitado, puede llegar a agotarse si únicamente asignamos memoria a variables anónimas y no liberamos memoria cuando ya no sea necesaria, de ahí la necesidad de un mecanismo para liberar memoria.

El núcleo del sistema de asignación dinámica de C++ está compuesto por la función `new` para la asignación de memoria, y la función `delete` para la liberación de memoria. Estas funciones trabajan juntas usando la región de memoria libre. Esto es, cada vez que se hace una petición de memoria con `new`, se asigna una parte de la memoria libre restante. Cada vez que se libera memoria con `delete`, se devuelve la memoria al sistema.

El subsistema de asignación dinámica de C++ se usa para dar soporte a una gran variedad de estructuras de programación, tales como *listas enlazadas* que se verán más adelante en este tema. Otra importante aplicación de la asignación dinámica es la *asignación dinámica de arrays*.

Funciones de reserva de memoria: *new*

La función `new` es simple de usar y será la función que **nosotros siempre utilizaremos**. El prototipo de la función `new` es:

```
void *new <nombre_tipo>;
```

Aquí, `nombre_tipo` es el tipo para el cual se quiere reservar memoria. La función `new` reservará la cantidad apropiada de memoria para almacenar un dato de dicho tipo.

La función `new` devuelve un puntero de tipo `void *`, lo que significa que se puede asignar a cualquier tipo de puntero. Convierte automáticamente el puntero devuelto al tipo adecuado, por lo que el programador no tiene que preocuparse de hacer la conversión de forma explícita.

```
char *p;  
p = new char; // reserva espacio para un carácter
```

La función `new` también se encarga de averiguar cual es el tamaño en bytes del tipo de datos para el cual se desea reservar memoria (recordemos que la cardinalidad de un tipo nos permite saber la memoria necesaria para su almacenamiento). Observar en el ejemplo anterior que se indica el que se quiere reservar memoria para un dato de tipo `char` y no cual es el tamaño en bytes que se quiere reservar.

Tras una llamada con éxito, `new` devuelve un puntero al primer byte de la región de memoria dispuesta del montón. **Si no hay suficiente memoria libre** para satisfacer la petición, se produce un fallo de asignación y `new` devuelve un **NULL**.

NOTA: Si se define un tipo puntero con `typedef`, cuando se reserva memoria para una variable de ese tipo se indica el tipo de datos de la variable anónima (a la que apunta el puntero) y no el nombre del tipo puntero.

```
typedef char *TPChar;
TPChar p;
p = new char; //Correcto
p = new TPChar; //INCORRECTO
```

NOTA: Como el montón no es infinito, siempre que se reserve memoria con `malloc()` o `new` debe comprobarse, antes de usar el puntero, el valor devuelto para asegurarse que no es *nulo*.

```
char *p;
p = new char;
if (p == NULL)
{
// No se ha podido reservar la memoria deseada
// Tratar error
}
```

Función de liberación de memoria dinámica: *delete*

La función `delete` es la complementaria de `new`. Una vez que la memoria ha sido liberada, puede ser reutilizada en una posterior llamada a `new`.

El prototipo de la función `delete` es:

```
void delete <variable_puntero>;
```

Aquí, `p` es un puntero a memoria que ha sido previamente asignado con `new`. Por ejemplo:

```
char *p;
p = new char; // reserva espacio para un carácter
delete p; //libera la memoria previamente reservada
```

NOTA: Es muy importante no llamar **NUNCA** a `delete` con un argumento no válido; se dañará el sistema de asignación.

NOTA: Las función `delete` p **NO** asignan el valor nulo al puntero `p` después de liberar la memoria a la que apuntaba.

Ejemplo de asignación dinámica de memoria

Para ver mediante un ejemplo, como se asigna/libera memoria dinámica, recordemos la estructura Complejo definida anteriormente.

Para crear una variable anónima de tipo Complejo hacemos una llamada a new que creará una variable anónima de tipo Complejo apuntada por el puntero ptr1, donde ptr1 es un puntero del tipo definido anteriormente TPCComplejo (Tipo puntero al tipo registro Complejo)

```
TPComplejo ptr1;  
ptr1 = new Complejo;
```

Si queremos liberar el espacio reservado antes, haremos una llamada a delete que liberará el espacio de memoria reservado para la variable de tipo Complejo apuntada por ptr1.

```
delete ptr1;
```

Gráficamente, teniendo en cuenta que Complejo es un registro con dos campos, las operaciones de asignación/liberación de memoria dinámica anteriores pueden verse de la siguiente forma:



Punteros a Estructuras

C++ permite punteros a estructuras igual que permite punteros a cualquier otro tipo de variables. Sin embargo, hay algunos aspectos especiales que afectan a los punteros a estructuras que vamos a describir a continuación.

La *declaración de un puntero a una estructura* se realiza poniendo * delante del nombre de la variable de estructura. Por ejemplo, si tenemos la estructura DatosPersonales la siguiente línea declara un tipo de datos puntero a un dato de este tipo:

```
struct DatosPersonales  
{ char nombre[80];  
  unsigned int edad;  
};  
typedef DatosPersonales *TPunt_dpersonales;
```

Ó declarando primero el puntero (indicando que es a una estructura) y luego se declara la estructura.

```
typedef struct DatosPersonales *TPunt_dpersonales;
struct DatosPersonales
{ char nombre[80];
  unsigned int edad;
};
```

Los punteros a estructuras se usan mucho con el fin de crear listas enlazadas y otras estructuras de datos dinámicas utilizando el sistema de asignación dinámica de memoria.

En el siguiente punto de este tema describiremos en profundidad las listas enlazadas.

Para acceder a una variable anónima de tipo registro apuntada por un puntero, se usa, como ya vimos en el punto 8.3 el operador de indirección *. A continuación mostramos un ejemplo:

```
Tpunt_dpersonales p;// Puntero a un registro de tipo DatosPersonales
p = new DatosPersonales;
//reserva memoria para una variable de tipo DatosPersonales y pone el
//puntero p apuntando a dicha variable anónima
(*p).edad = 15;
//accede al campo edad del registro y le asigna el valor 15
```

Como se puede observar en la última sentencia para acceder a campo de una variable anónima de tipo registro apuntado por un puntero, se pone el operador de indirección entre paréntesis (para evitar problemas de precedencia pues el operador * tiene mas precedencia que el .) seguido de un . y el nombre del campo al que se desea acceder. Debido a que esta notación resulta algo incomoda de escribir es posible usar en su lugar el operador ->. A -> se le denomina operador flecha y consiste en el signo menos seguido de un signo de mayor.

Cuando se accede a un miembro de una estructura a través de un puntero a la estructura, se usa el operador flecha en lugar del operador punto. Según esto la última sentencia del ejemplo anterior se suele escribir de la forma siguiente:

```
P -> edad = 15; // accede al campo edad del registro.
```

NOTA: Si p es un puntero a struct DatosPersonales, p.edad ES INCORRECTO

Para pedir memoria para mas de un elemento la sintaxis de new y delete cambia.

Nueva sintaxis:

```
p-var=new type[tamaño];
delete [tamaño]p-var; ó delete []p-var; // para un compilador reciente.
```

Ejemplo:

```

main()
{ int *v;
  int n;
  cout << "Cuantas posiciones\n"
  cin << n;
  v=new int[n];
  .
  .
  delete [n]v;
}

```

Listas encadenadas o enlazadas

Una lista enlazada está formada por una colección de elementos (denominados *nodos*) tales que cada uno de ellos almacena dos valores: (1) un valor de la lista y (2) un *apuntador* que indica la posición del nodo que contiene el siguiente valor de la lista.

Un arreglo almacena todos sus elementos agrupados en un bloque de memoria. Mientras que en una lista enlazada la memoria se va tomando según se necesita. Se van agregando “nodos”. Cuando queremos añadir un nuevo elemento (“nodo”) reservamos memoria para él y lo añadimos a la lista. La lista va tomando su estructura usando apuntadores que conectan todos sus nodos como los eslabones de una cadena, o las cuentas de un collar. Cuando queremos eliminar el elemento simplemente lo sacamos de la lista y liberamos la memoria usada.

Lo que se desea es una nueva forma de asignar memoria, para nuevos nodos de la lista, solo cuando se requiere y desasignar esa memoria cuando ya no se necesita. Además, la memoria recién asignada debe combinarse lógicamente en una sola entidad (la representación de la lista). Una lista encadenada proporciona estas capacidades.

Retomando, una lista encadenada comprende un conjunto de nodos, cuyo formato incluye dos campos macro: uno que contiene los datos (información) deseados; y el otro que contiene la dirección del nodo al cual apunta en la lista (Enlace).



Figura Formato de un nodo para una lista encadenada

```

struct nodo
{   int información;
    struct node* Enlace;
};

```

En la anterior definición del tipo de dato nodo, se supone que como dato solo podrá almacenar un entero.

Ventajas del uso de listas encadenadas

- Hay una ganancia implícita de memoria, porque se pueden usar partes de memoria de las que no pueden disponer las listas secuenciales.
- Los algoritmos de inserción y borrado son más sencillos.
- Es más sencillo unir dos listas, o dividir una lista.
- Permite abstracción con polimorfismo.
- Este esquema permite la creación de estructuras mucho más complejas.

Pero la ventaja más importante, es la de poder realizar la abstracción de las diferentes estructuras de datos, definidas dinámicamente.

Desventajas

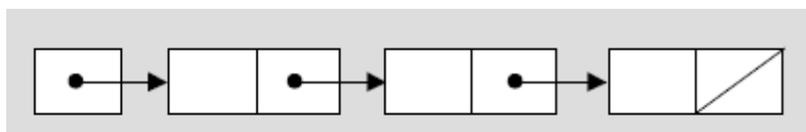
- Encontrar un nodo arbitrario en una lista encadenada requiere comenzar la búsqueda por el inicio de la lista y, usar la información de cada elemento hasta encontrar el nodo deseado.
- Los apuntadores requieren memoria adicional.
- Para entrar a una lista encadenada se requiere información sobre el primer nodo.

El uso de listas encadenadas implica la existencia de mecanismos para la asignación dinámica de la memoria requerida, que depende del tipo de herramienta de programación. Este es ambiente ideal del lenguaje C y C++

Operaciones básicas sobre listas enlazadas

Los punteros y la asignación dinámica de memoria permiten la construcción de estructuras enlazadas. Una estructura enlazada es una colección de nodos, cada uno de los cuales contiene uno o más punteros a otros nodos. Cada nodo es un registro en el que uno o más campos son punteros.

La estructura enlazada más sencilla es la *lista enlazada*. Una lista enlazada consiste de un número de nodos, cada uno de los cuales contiene uno o varios datos, más un puntero; el puntero permite que los nodos formen una lista.



Una variable puntero (puntero externo) apunta al primer nodo en la lista, el primer nodo apunta al segundo, y así todos los demás. El último nodo contiene el *valor nulo* en su campo de tipo puntero.

Para ilustrar las listas enlazadas, mostraremos como *definir*, *crear* y *manipular* una lista enlazada cuyos nodos contienen un único carácter. Los nodos de la lista se

implementan con registros, ya que cada nodo debe disponer de un enlace al siguiente elemento además de los datos de la lista propiamente dichos.

En nuestro ejemplo, declaramos el tipo TLista que define un puntero al registro Nodo que representa un nodo de la lista. Un valor del tipo Nodo será un registro con dos campos: c (el carácter almacenado en el nodo) y sig (un puntero al siguiente nodo en la lista).

```

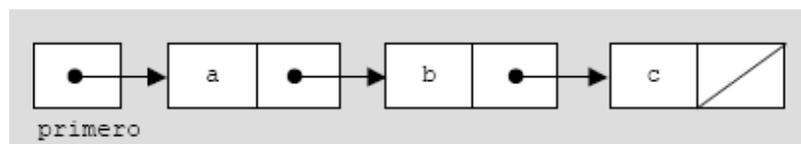
struct TNodo
{ char c;
  TNodo *sig;
};

typedef TNodo *TLista;
typedef struct TNodo *TLista;
struct TNodo
{ char c;
  TLista sig;
};

```

Tabla 1. Las 2 Posibilidades (equivalentes) de Definir una Lista Enlazada

Podemos dibujar una variable de tipo Nodo como una caja con dos campos. Una lista enlazada creada con nodos de tipo Nodo tendría la forma siguiente, donde primero es una variable de tipo TLista (puntero externo) que apunta al primer nodo de la lista: TLista primero;



Con esta estructura, para almacenar una cadena de caracteres en memoria iremos leyendo carácter a carácter desde teclado, y creando variables anónimas de forma dinámica, almacenando en cada una de ellas un carácter y enlazándolas mediante el campo puntero.

Procederemos así hasta que nos introduzcan un retorno de carro, que nos indicará el final de la cadena de caracteres.

Las operaciones básicas sobre listas enlazadas son:

- 1) Creación de una la lista enlazada
- 2) Insertar un nodo en la lista enlazada
- 3) Borrar un nodo de la lista enlazada

Creación de una lista enlazada

El procedimiento de creación de una lista enlazada es muy simple e inicializa un puntero del tipo TLista a NULL.

```

TLista crearListaEnlazada()
{ return NULL;
}

```

Insertar un nodo en la lista enlazada

La tarea de insertar un nodo en una lista enlazada podemos separarla en dos casos:

- 1) Insertar un nodo al comienzo de una lista
- 2) Insertar un nodo después de un determinado nodo existente (por ejemplo, para mantener una lista ordenada).

1) Insertar un nodo al comienzo de una lista

Para insertar un nuevo nodo al comienzo de una lista, debemos seguir los pasos siguientes:

1. Crear un nodo para una variable anónima de tipo TNode.

```
punt = new TNode;
```

2. Una vez que se ha creado un nodo apuntado por punt, se le asigna al campo c el carácter que queremos insertar (ej: "d").

```
punt->c = 'd';
```

3. Hacemos que el campo sig del nuevo nodo apuntado por punt, apunte al primer nodo de la lista (esto es, que apunte donde apunta primero).

```
punt->sig = primero;
```

4. Por último, hemos de actualizar primero, puesto que este puntero siempre ha de apuntar al primer elemento de la lista, y en este caso este es el nuevo nodo insertado.

```
primero = punt;
```

```
void insertarFrente(TLista &primero1, char car)
{ // Esta función inserta un nodo al principio de la lista
  TLista punt;
  punt = new TNode;
  if (punt == NULL)
  { cout << "No hay memoria suficiente" << endl;
  }
  else
  { punt->c = car;
    punt->sig = primero;
    primero = punt;
  }
}
```

¹ Para modificar la lista enlazada debemos pasar la variable `primero` por referencia. Recordar del Tema 4 que para pasar un valor por referencia a una función se hacía anteponiendo el operador `&` al nombre de la variable

}

2) Insertar un nodo en una posición determinada de una lista

Cuando queremos insertar un elemento de forma ordenada en una lista enlazada ordenada (por ejemplo, ordenada alfabéticamente):

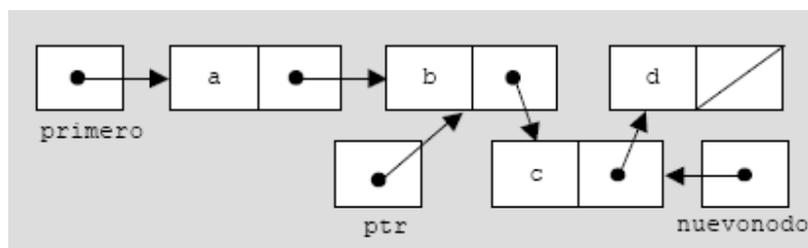
1. Se ha de crear un nodo nuevo, y en su campo c almacenar el valor del carácter que quiera insertar. Para ello creamos un nodo apuntado por la variable puntero nuevonodo, y se asigna al campo c el valor del carácter leído (por ejemplo, "c").

```
nuevonodo = new TNode;
if (nuevonodo == NULL)
{ cout << "No hay memoria suficiente" << endl;
}
else
{ nuevonodo->c = 'c';
  nuevonodo->sig = NULL;
}
```

2. Se necesita una variable puntero auxiliar, que llamaremos ptr, que recorra cada uno de los nodos de la lista, hasta que encuentre el lugar exacto donde insertar el nuevo nodo; pero ptr siempre ha de estar apuntando al nodo anterior, con respecto al nodo que establece la comparación, para que una vez que ha localizado el lugar exacto pueda enlazar el campo siguiente del nodo anterior (apuntado por ptr) al nodo a insertar (nuevonodo), y pueda también enlazar el campo siguiente de nuevonodo, haciendo que apunte al nodo apuntado por el campo siguiente de ptr.

```
while (ptr->sig != NULL) && (nuevonodo->c > ptr->sig->c)
{ ptr = ptr->sig;
}
nuevonodo->sig = ptr->sig;
ptr->sig = nuevonodo;
```

Gráficamente, quedaría del modo siguiente:



```
void insertarOrdenado(TLista &primero, char c)
{ // Esta función inserta un nodo en la lista de forma ordenada
  // declaración variables
  TLista nuevonodo, ptr;
  nuevonodo = new TNode;
  if (nuevonodo == NULL)
  { cout << "No hay memoria suficiente" << endl;
```

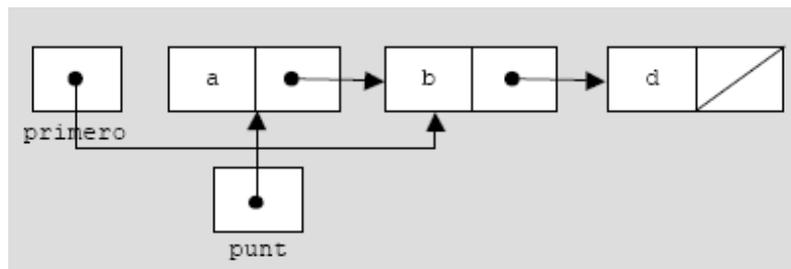


```
primero = primero->sig;
```

3. Liberar la memoria ocupada por la variable anónima a la que apunta la variable puntero
punto, que es el nodo que queremos eliminar.

```
delete punto;
```

Gráficamente, quedaría del modo siguiente:

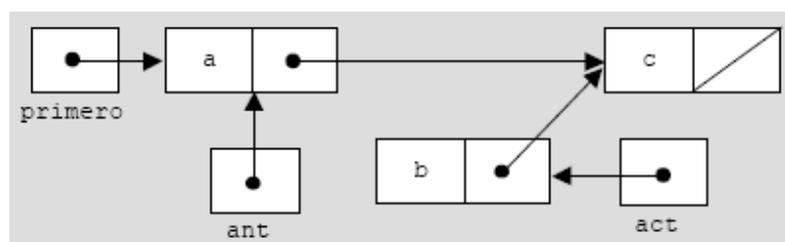


```
void borrarFrente(TLista &primero)
{ // Este procedimiento borra el primer nodo de la lista
  // declaración variables
  TLista punto;
  if (primero != NULL)
  { punto = primero;
    primero = primero->sig;
    delete punto;
  }
}
```

2) Borrar un nodo de una posición determinada de una lista

Para borrar un determinado nodo de una lista enlazada, necesitamos dos punteros auxiliares, uno que me apunte al nodo a borrar, puntero que llamaremos act; y otro que me apunte al nodo anterior al que deseamos borrar, de modo que podamos enlazar este con el nodo siguiente al borrado, puntero al que llamaremos ant.

Si quisiéramos borrar el segundo nodo (nodo que contiene “b” en la figura) de una lista ordenada cuyo primer elemento viene apuntado por primero, tendríamos lo siguiente:



```
void borrarOrdenado(TLista &primero, char c)
```

```

{ /* Este procedimiento borra un determinado elemento de
la lista*/
// declaración variables
TLista act,ant;
if (primero != NULL)
{ ant = NULL;
act = primero;
while (act->c != c)
{ ant = act;
act = act->sig;
}
if (ant == NULL)
{ primero = primero->sig;
}
else
{ ant->sig = act->sig;
}
delete act;
}
}

```

Escribir un programa para el mantenimiento de notas usando listas simplemente enlazadas.

```

struct nodo
{ int nota;
struct nodo *siguiente;
} cabeza;

int ingreso (int n)
{ struct nodo *nuevo;
if (!(nuevo=new struct nodo)) return 0;
nuevo->nota=n;
if (cabeza) nuevo->siguiente=cabeza;
else nuevo->siguiente=NULL;
cabeza->nuevo;
return 1;
}

void desplegar ();
{ struct nodo *actual;
actual=cabeza;
while (actual)
{ cout << actual->nota << '\n';
actual=actual->siguiente;
}
}

void desplegar_rec (struct nodo *p)
{ if (p)cout << p->nota << '\n'
desplegar_rec (p->siguiente);
}

void eliminar (struct nodo *p)
{ if (p)

```

```

{ eliminar (p->siguiente);
  delete p;
}
}

```

Otras clases de listas enlazadas

Otras clases de listas enlazadas son las listas doblemente enlazadas. Las listas doblemente enlazadas consisten en datos, más un enlace al elemento siguiente y otro enlace al elemento anterior.

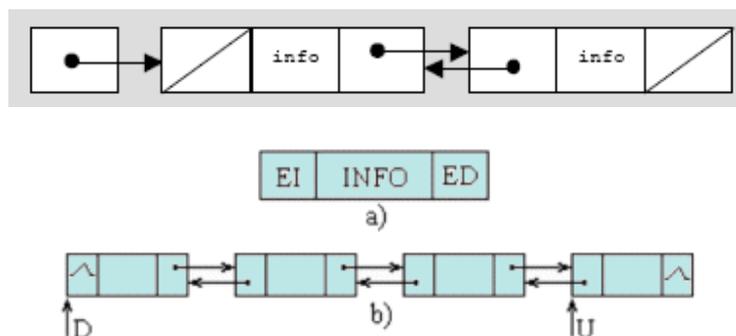


Figura (a) Formato del nodo de listas doblemente encadenadas EI, Enlace izquierdo; ED, Enlace derecho; INFO, la información que almacena el nodo. (b) Cola doblemente encadenada abierta.

Disponer de dos enlaces en lugar de sólo uno tiene varias ventajas:

O *La lista puede recorrerse en cualquier dirección*. Esto simplifica la gestión de la lista, facilitando las inserciones y las eliminaciones.

O *Más tolerante a fallos*. Se puede recorrer la lista tanto con los enlaces hacia delante como con los enlaces hacia atrás, así si se invalida algún enlace se puede reconstruir la lista utilizando el otro enlace.

La forma de construir una lista doblemente enlazada es similar a la de una lista simplemente enlazada excepto en que hay que mantener dos enlaces. Por tanto, el registro tiene que contener esos dos enlaces. Usando el ejemplo anterior modificamos el registro y añadimos el nuevo enlace.

```

struct Nodo
{ char c;
  Nodo* sig;
  Nodo* ant;
};
typedef Nodo *TListaDobleEnlazada;

```

Las operaciones básicas son las mismas que en las listas simplemente enlazadas:

- 1) Creación una la lista doblemente enlazada
- 2) Insertar un nodo en la lista doblemente enlazada

3) Borrar un nodo de la lista doblemente enlazada

Creación de una lista doblemente enlazada

Una lista doblemente enlazada se crea de la misma forma que una lista enlazada, inicializando un puntero del tipo TListaDobleEnlazada a NULL.

```
TListaDobleEnlazada crearListaDobleEnlazada()
{ return NULL;
}
```

Insertar un nodo en la lista doblemente enlazada

La tarea de insertar un nodo en una lista doblemente enlazada podemos separarla en dos casos:

- 1) Insertar un nodo al comienzo de la lista
- 2) Insertar un nodo después de un determinado nodo existente (por ejemplo, para mantener una lista ordenada).

1) Insertar un nodo al comienzo de la lista doblemente enlazada

El procedimiento para insertar un nodo al comienzo de una lista doblemente enlazada (suponemos que anteriormente hemos creado una lista, cuyo primer nodo está apuntado por el puntero primero), se muestra a continuación:

```
void insertarFrenteDobleEnlazada(TListaDobleEnlazada &primero, char c)
{ // Esta función inserta un nodo al principio de la lista
  // declaración variables
  TListaDobleEnlazada punt;
  punt = new Nodo;
  if (punt == NULL)
  { cout << "No hay memoria suficiente" << endl;
  }
  else
  { punt->c = c;
    punt->sig = NULL;
    punt->ant = NULL;
    if (primero != NULL)
    { punt->sig = primero;
      primero->ant = punt;
    }
    primero = punt;
  }
}
```

2) Insertar un nodo en una posición determinada de una lista doblemente enlazada

Para insertar un nodo en una lista enlazada ordenada tenemos que buscar primero la posición donde hay que insertar el elemento. Un procedimiento para insertar un nodo en una posición determinada de una lista doblemente enlazada es el siguiente:

```
void insertarOrdenadoDobleEnlazada(TListaDobleEnlazada &primero, char c)
{ // Inserta un nodo en la lista doblemente enlazada ordenada
  TListaDobleEnlazada nuevonodo, ptr, ant;
  nuevonodo = new Nodo;
  if (nuevonodo == NULL)
  { cout << "No hay memoria suficiente" << endl;
  }
  else
  { nuevonodo->c = c;
    nuevonodo->sig = NULL;
    nuevonodo->ant = NULL;
    if (primero == NULL)
    { // La lista estaba vacía
      primero = nuevonodo;
    }
    else
    { if (nuevonodo->c <= primero->c)
      {
        /* c es menor que el primer elemento de la lista */
        nuevonodo->sig = primero;
        primero->ant = nuevonodo;
        primero = nuevonodo;
      }
      else
      { ant = primero;
        ptr = primero->sig;
        while ((ptr != NULL) && (nuevonodo->c > ptr->c))
        { ant = ptr;
          ptr = ptr->sig;
        }
        if (ptr == NULL)
        { /* c se inserta al final de la lista */
          nuevonodo->ant = ant;
          ant->sig = nuevonodo;
        }
        else
        { /* c se inserta en medio de la lista */
          nuevonodo->sig = ptr;
          nuevonodo->ant = ant;
          ant->sig = nuevonodo;
          ptr->ant = nuevonodo;
        }
      }
    }
  }
}
```

Borrar un nodo de la lista doblemente enlazada

En cuanto a la tarea de borrar un nodo en una lista enlazada, también hay dos posibilidades:

- 1) Borrar el primer nodo de la lista enlazada
- 2) Borrar un determinado nodo de la lista enlazada (por ejemplo, para mantener la lista ordenada).

1) Borrar el primer nodo de la lista doblemente enlazada

```
void borrarFrenteDobleEnlazada(TListaDobleEnlazada &primero)
{ // Este procedimiento borra el primer nodo de la lista
  // declaración variables
  TListaDobleEnlazada punt;
  if (primero != NULL)
  { punt = primero;
    primero = primero->sig;
    delete punt;
  }
}
```

2) Borrar un nodo de una posición determinada de una lista doblemente enlazada

Para borrar un determinado nodo de una lista enlazada, necesitamos dos punteros auxiliares, uno que me apunte al nodo a borrar, puntero que llamaremos act; y otro que me apunte al nodo anterior al que deseamos borrar, de modo que podamos enlazar este con el nodo siguiente al borrado, puntero al que llamaremos ant.

```
void borrarOrdenadoDobleEnlazada(TListaDobleEnlazada &primero, char c)
{ /* Este procedimiento borra un determinado elemento de la lista */
  // declaración variables
  TListaDobleEnlazada act, ant;
  ant = NULL;
  act = primero;
  while ((ptr != NULL) && (ptr->c != c))
  { ant = ptr;
    ptr = ptr->sig;
  }
  if (ptr != NULL)
  { /* Se ha encontrado el elemento */
    if (ant == NULL)
    { /* El elemento a borrar es el primero de la lista */
      primero = primero->sig;
      if (primero != NULL)
      { primero->ant = NULL;
      }
    }
    else if (ptr->sig == NULL)
    { /* El elemento a borrar es el ultimo de la lista */
      ant->sig = NULL;
    }
    else
    { /* El elemento a borrar esta en el medio de la lista */
```

```

    ant->sig = ptr->sig;
    ptr->sig->ant = ant;
}
delete ptr;
}
}

```

Ejercicios

1.- Dadas las siguientes declaraciones:

```

struct Node
{ int info;
  Node *next;
};
typedef Node *TPNode;
TPNode p, q;

```

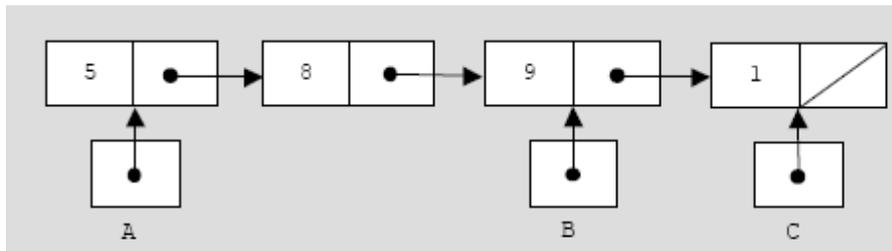
Muestra cual es la salida de los siguientes segmentos de código:

```

(a) q = new Node;
    p = new Node;
    P->info = 5;
    Q->info = 6;
    p = q;
    P->info = 1;
    cout << p->info;
    cout << q->info;
(b) p = new Node;
    P->info = 3;
    q = new Node;
    q->info = 2;
    p = new Node;
    p->info = q->info;
    q->info = 0;
    cout << p->info;
    cout << q->info;
(c) p = new Node;
    q = new Node;
    p->info = 0;
    p->next = q;
    q->next = NULL;
    q->info = 5;
    p->next->next = p;
    q->info = q->next->info;
    p = q;
    p->next->info = 2;
    cout << p->info;
    cout << q->info;

```

2.- Muestra el efecto de las siguientes sentencias sobre la siguiente lista enlazada. Asume que cada sentencia es ejecutada de forma aislada.



Donde:

```

struct rec
{ int data;
  rec *next;
};
typedef rec *Tprec;
TPrec A, B, C;
(a) A = A->next;
(b) A = A->next->next;
(c) C->next = A;
(d) C->next = A->next;
(e) A->data = 0;
(f) B->data = A->next->data
(g) B->data = A->next->next->data
  
```

3. Haz un programa que implemente los siguientes procedimientos y funciones para el manejo de Listas Enlazadas. Para probarlas el programa principal tendrá un menú donde cada una de las operaciones sea una opción del menú.

```

TipoLista Crear();
(* crea una lista enlazada vacía*)
bool Lista_Vacia(TipoLista lista);
(* devuelve TRUE si la lista enlazada que se la pasas como parámetro está vacía, en caso contrario devuelve FALSE*)
void Imprimir(TipoLista lista);
(*Imprime el contenido de la lista que se la pasa como parámetro*)
void Insertar_Frente(TipoLista &lista, TipoElem elem);
(*Inserta en la cabeza de la lista el elemento que se le pasa como parámetro*);
void Eliminar_Frente(TipoLista &lista, TipoElem elem);
(* Elimina el primer elemento de la lista y devuelve su contenido en el parámetro elem*)
void Insertar_ordenado(TipoLista &lista, TipoElem elem);
(*Inserta en la lista (ordenada) el elemento elem de forma ordenada*)
void Eliminar_Ordenado(TipoLista &lista, TipoElem elem);
(*Elimina el elemento elem de la lista, si esta en ella, suponiendo que dicha lista esta ordenada*)
  
```

Donde:

```

typedef unsigned int TipoElem;
struct TipoLista
{ TipoElem c;
  TipoLista *p;
};
  
```

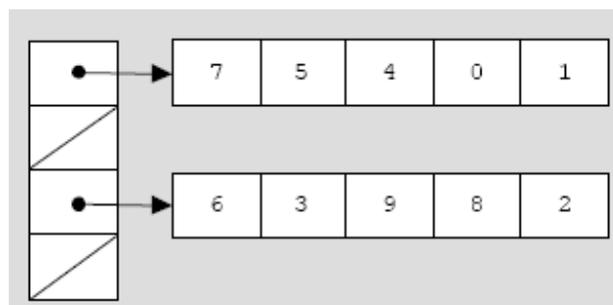
```
};
typedef TipoLista *TPTipoLista;
```

4.- Diseña un Algoritmo que tenga como entrada dos matrices cuadradas y devuelva la suma de ambas. Estas matrices tienen la peculiaridad de que algunas de sus filas tienen todos sus componentes a 0; por esta razón estas matrices se podrían representar de una forma más eficiente, que como las habíamos venido representando hasta el momento (arrays bidimensionales), como un array de punteros a arrays de cardinales:

Ejemplo, la matriz cuadrada siguiente (5X5):

```
7 5 4 0 1
0 0 0 0 0
6 3 9 8 2
0 0 0 0 0
0 0 0 0 0
```

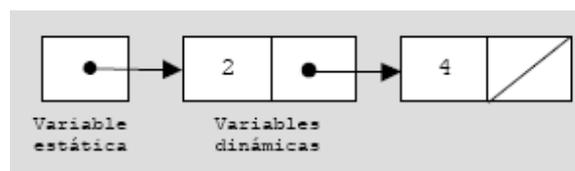
Vendrá representada como:



Diseña el programa utilizando la representación anterior.

5.- Se conoce que en C++ (32 bits) el tipo de dato entero unsigned int, sólo toma valores enteros positivos, con un valor máximo de 4.294`967.295 (32 bits unos). El objetivo de este ejercicio es ser capaces de manipular números cardinales incluso mayores que el anterior.

Con este fin se diseña un programa que permita manejar enteros largos. Estos enteros largos serán implementados mediante una lista enlazada con los dígitos que forman el número. De modo, que 42 se almacenaría como:

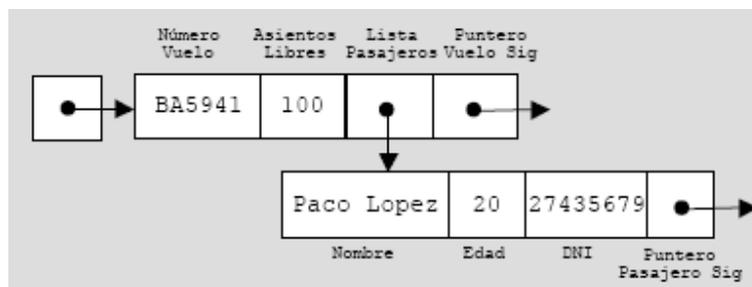


Este programa debe contener un menú que me permita llevar a cabo las siguientes operaciones: suma, resta y multiplicación de dos enteros largos, además de las de entrada/salida (lectura de un entero largo y escritura de un entero largo).

6.- Diseña un algoritmo para un sistema de reserva de vuelos de un aeropuerto. El algoritmo ha de ejecutar un menú que permita llevar a cabo las operaciones siguientes:

- Añadir un vuelo (dado un número de vuelo y un número de asientos libres) al sistema.
- Eliminar un vuelo del sistema.
- Hallar cuantos asientos libres hay para un vuelo dado.
- Añadir una persona a un vuelo dado.
- Eliminar a una persona de un vuelo dado.
- Listar por pantalla los datos de las personas (nombre, edad, DNI) que tienen reservas para un vuelo dado.

El algoritmo debería representar el sistema de reservas mediante una lista enlazada de vuelos. El número de vuelo (ej. BA5941), el número de asientos disponibles y una lista, también enlazada, de pasajeros deberían ser almacenados para cada vuelo. Cada nodo de la lista enlazada de pasajeros contendrá toda la información relacionada con una persona: nombre, edad y DNI.



Apéndice. Manejo Tradicional de Memoria en C.

Además de lo mostrado en el tema, existen otras funciones y otras posibilidades de manejo de memoria dinámica y de uso de punteros que si bien no son pedagógicamente las más adecuadas, se muestran en este apéndice para que sirva como material de consulta para el alumno en caso de necesidad o de que desea ampliar sus conocimientos sobre el lenguaje.

Otras funciones de reserva de memoria de forma dinámica.

Otra posibilidad existente a la hora de reservar memoria dinámicamente es el uso de la función malloc ó calloc.

Antes de indicar como deben utilizarse las funciones se tiene que comentar el operador sizeof. Este operador es imprescindible a la hora de realizar programas

portables, es decir, programas que puedan ejecutarse en cualquier máquina que disponga de un compilador de C.

El operador `sizeof(tipo_de_dato)`, devuelve el tamaño que ocupa en memoria un cierto tipo de dato, de esta manera, se puede escribir programas independientes del tamaño de los datos y de la longitud de palabra de la máquina. En resumen si no se utiliza este operador en conjunción con las conversiones de tipo cast probablemente nuestro programa sólo funcione en el ordenador sobre el que lo ha programado.

Los prototipos de estas funciones son:

```
void *malloc(size_t número_de_bytes);
void *calloc(size_t número_de_elementos, size_t número_de_bytes);
```

Aquí, `número_de_bytes` es el número de bytes de memoria que se quieren asignar. El tipo `size_t` está definido en `<stdlib>` como un tipo entero sin signo.

```
void *calloc(size_tnobj, size_tsize)
```

`calloc` obtiene (reserva) espacio en memoria para alojar un vector (una colección) de `nobj` objetos, cada uno de ellos de tamaño `size`. Si no hay memoria disponible se devuelve `NULL`. **El espacio reservado se inicializa a bytes de ceros.**

Obsérvese que `calloc` devuelve un `(void *)` y que para asignar la memoria que devuelve a un tipo `Tipo_t` hay que utilizar un operador de ahormado: `(Tipo_T*)`

Ejemplo:

```
char *c;
c=(char *) calloc (40, sizeof(char));
```

`void *malloc(size_tsize)` Es necesario saber el tamaño exacto de las posiciones de memoria solicitadas.

El ejemplo anterior se puede reescribir:

```
char *c;
c=(char *) malloc (40*sizeof(char));

char *cpt;
...
if ((cpt = (char *) malloc(25)) == NULL)
{ printf("Error on malloc\n");
}
```

En resumen la función `malloc` ó `calloc()` devuelve un puntero de tipo `void *`, lo que significa que se puede asignar a cualquier tipo de puntero. Se tiene que convertir de forma explícita el resultado al tipo de puntero para el cual se está reservando memoria. En la línea de código siguiente se indica con `(char *)` que el puntero `p` es un puntero a `char`.

Tras una llamada con éxito a `malloc` ó `calloc()` devuelve un puntero al primer byte de la región de memoria dispuesta del montón. Si no hay suficiente memoria libre para satisfacer la petición de `malloc/calloc()`, se produce un fallo de asignación y `malloc` ó `calloc()` devuelve un `NULL`.

```
char *p;  
p = (char *) malloc(1000); /*reserva 1.000 bytes */
```

Para terminar un breve comentario sobre las funciones anteriormente descritas. Básicamente da lo mismo utilizar `malloc` y `calloc` para reservar memoria es equivalente:

```
mat = (float *)calloc (20,sizeof(float));  
mat = (float *)malloc (20*sizeof(float));
```

La diferencia fundamental es que, a la hora de definir matrices dinámicas `calloc` es mucho más claro y además inicializa todos los elementos de la matriz a cero. Nótese también que puesto que las matrices se referencian como un puntero la asignación dinámica de una matriz nos permite acceder a sus elementos con instrucciones de la forma:

NOTA: En realidad existen algunas diferencias al trabajar sobre máquinas con alineamiento de palabras.

`void *realloc(void *p, size_tsize)`

`realloc` cambia el tamaño del objeto al que apunta `p` y lo hace de tamaño `size`. El contenido de la memoria no cambiará en las posiciones ya ocupadas. Si el nuevo tamaño es mayor ue el antiguo, no se inicializan a ningún valor las nuevas posiciones. En el caso en que no hubiese suficiente memoria para “relojar” al nuevo puntero, se devuelve `NULL` y `p` no varía.

El puntero que se pasa como argumento ha de ser `NULL` o bien un puntero devuelto por `malloc()`, `calloc()` o `realloc()`.

Para la reserva de memoria dinámica en C, es muy útil la utilización de la función `sizeof()`.

```
int *p;  
/* reserva espacio para un valor de tipo entero (int)*/  
p = (int *) malloc(sizeof(int));
```

NOTA: Como el montón (heap) no es infinito, siempre que se reserve memoria con `malloc()` o `calloc` debe comprobarse, antes de usar el puntero, el valor devuelto para asegurarse que no es *NULL*.

```
char *p;
p = malloc(sizeof(char));
if (p == NULL)
{
// No se ha podido reservar la memoria deseada
// Tratar error
}
```

Función de liberación de memoria dinámica: *free()* .

Cuando se usa memoria de manera dinámica, es necesario que el programador libere la memoria luego de utilizarla, esto se hace usando la función de librería estándar *free*:

free() libera el espacio de memoria al que apunta *p*. Si *p* es *NULL* no hace nada.

Además *p* tiene que haber sido “alojado” previamente mediante `malloc()`, `calloc()` o `realloc()`.

La función `free()` es la complementaria de `malloc` ó `calloc()`. Una vez que la memoria ha sido liberada, puede ser reutilizada en una posterior llamada por `malloc` ó `calloc()`.

El prototipo de la función *free()* es: `void free(void *p);`

Aquí, *p* es un puntero a memoria que ha sido previamente asignado con `malloc` ó `calloc()` o `new`. Por ejemplo:

```
char *p;
p = (char *) malloc(sizeof(char)); //reserva memoria para un carácter
free(p);                          //libera la memoria previamente reservada
```

NOTA: Es muy importante no llamar **NUNCA** a `free()` con un argumento no válido; se dañará el sistema de asignación.

NOTA: La función `free(p)` NO asignan el valor nulo al puntero *p* después de liberar la memoria a la que apuntaba.

NOTA: Usar siempre `free()` para liberar la memoria si se usó `malloc` ó `calloc()` para reservarla. Y, usar siempre `delete` para liberar la memoria si se usó `new` para reservarla.

Creación Dinámica de Arrays y Aritmética de Punteros.

En C/C++ los punteros y los arrays están muy relacionados. En realidad, un nombre de array es un puntero al primer elemento del array, ya que la forma de implementar los arrays es reservar un bloque de memoria de tamaño:

```
<dimensión>*sizeof(<tipo_base>) .
```

Tras lo visto con anterioridad en el tema no hay ningún problema en que esa reserva de memoria se haga en tiempo de compilación (caso habitual de la declaración estática de un array) o de forma dinámica (declarando sólo el puntero a la primera posición y reservando la memoria en tiempo de ejecución). Esto nos permitiría declarar un array de tamaño decidible en tiempo de ejecución pero nos obliga a gestionar la reserva/liberación de esa memoria.

Por ejemplo, el siguiente fragmento de código es válido en C++:

```
char cad[80]; // Declaración de un array de 80 caracteres
char *p; // Declaración de un puntero a char
p = cad; // El puntero p apunta a la primera posición del array
```

Aquí a p se le asigna la dirección del primer elemento del array cad, con lo que se puede decir que el puntero p apunta a un array de 80 caracteres.

Esta característica de los arrays hace que el paso de arrays como argumentos de una función sea una excepción al convenio de llamada por valor estándar². Cuando se usa un array como argumento de un procedimiento o función, sólo se pasa a la función la dirección del array, indicando su nombre. Por ejemplo, la función imprimir_mayúsculas(), que imprime su argumento de tipo cadena en mayúsculas se declara como:

```
typedef char *TPCadena;
void imprimir_mayúsculas(TPCadena cadena);
```

Y se invocaría de la siguiente forma:

```
imprimir_mayúsculas(c);
```

Donde c es el nombre de un array declarado de forma estática:

```
char c[80];
```

En este caso, el código que hay dentro del procedimiento o función opera sobre el contenido real del arreglo (array), pudiendo alterarlo. El procedimiento imprimir_mayúsculas() modifica el contenido del array cadena, de forma que después

² El paso de argumentos por valor y por referencia está explicado en el Tema4. Procedimientos y Funciones

de hacer una llamada a este procedimiento, la cadena estará también en mayúsculas en el programa principal.

```
#include <iostream>
#include <cstdlib>
using namespace std;
const char FINCAD = char(0);
const char MAXCAD = 80;
const int ENTER = '\n';
typedef char *TPCadena;
typedef char TCadena[MAXCAD+1];
void imprimir_mayusculas(TPCadena cadena);
void main()
{ TCadena c;
  cout << "Introduce una cadena: " << endl;
  cin.getline(c, MAXCAD, ENTER);
  imprimir_mayusculas(c); //El valor de c se ha modificado
  cout << endl << "c esta ahora en mayusculas: " << c << endl;
  system("PAUSE");
  return 0;
}
void imprimir_mayusculas(TPCadena cadena)
{ //Imprime una cadena en mayúsculas
  int i;
  i=0;
  while (cadena[i] != FINCAD)
  { cadena[i] = toupper(cadena[i]); //Modifica el valor de cadena
    cout << cadena[i];
    i++;
  }
}
```

En este ejemplo, si no quisiera que el contenido de la cadena se cambiara a mayúsculas en el programa principal tendría que implementar el procedimiento `imprimir_mayusculas()` de la siguiente forma, donde el contenido del array no se modifica dentro del procedimiento:

```
void imprimir_mayusculas(TPCadena cadena)
{ //Imprime una cadena en mayúsculas
  int i;
  i=0;
  while (cadena[i] != FINCAD)
  { cout << toupper(cadena[i]);
    i++;
  }
}
```

En la situación anterior el array se declara de forma estática asignándole una longitud fija, aún cuando consideremos el nombre del array como un puntero a dicho array. Sin embargo, hay ocasiones en las que se quiere trabajar con un array, pero no sabemos cual será el tamaño del array que vamos a necesitar. En este caso es posible usar la función `new` para crear un *array de forma dinámica*.

Para asignar un array de forma dinámica, primero hay que declarar un puntero que su tipo sea el tipo de los elementos que queremos almacenar en el array.

```
<nombre_tipo> *<var_puntero>; //declaramos un puntero a "nombre_tipo"
```

Posteriormente tenemos que usar la función de asignación dinámica new para reservar el espacio de memoria necesario indicando el número de datos de dicho tipo que quiere almacenar en el array.

```
<var_puntero> = new <nombre_tipo>[<tamaño>];
```

Para liberar la memoria asignada a un array de forma dinámica, no es necesario especificar el tamaño del array, sólo los corchetes [].

```
delete []<var_puntero>;
```

Por ejemplo, si se quiere crear un array de 80 char:

```
char *c; // puntero a char
c = new char[80]; // reserva memoria para 80 caracteres
```

El siguiente programa muestra cómo se puede usar un array asignado dinámicamente para mantener un array unidimensional; una cadena en este caso.

```
#include <iostream>
#include <cstdlib>
using namespace std;
const char FINCAD = char(0);
const char MAXCAD = 80;
const int ENTER = '\n';
/*Se define el TPCadena como puntero a un número variable de caracteres*/
typedef char *TPCadena;
int LongitudCadena(TPCadena cadena);
int main()
{ char *c;
  int i;
  // Reserva memoria para almacenar una cadena de caracteres
  c = new char[MAXCAD+1];
  if (c == NULL)
  { cout << "No hay memoria suficiente" << endl;
  }
  else
  { // Lee desde teclado la cadena de caracteres
    cin.getline(c, MAXCAD, ENTER);
    // Imprime la cadena al revés
    for (i=LongitudCadena(c)-1; i>=0; i--)
    { cout << c[i];
    }
    delete []c;
  }
  system("PAUSE");
```

```

    return 0;
}

int LongitudCadena(TPCadena cadena)
{ int i;
  i=0;
  while ((i<MAXCAD)3&& (cadena[i]!=FINCAD) )
  { ++i;
  }
  return i;
}

```

Aun más, si piensa en la implementación de los arrays en C/C++ y en que un puntero no es más que una dirección de memoria. ¿Qué ocurre si se incrementa o decrementa un puntero? La respuesta es sencilla, se va a la siguiente o anterior posición de memoria. ¿Qué ocurre si se hace eso en un array? Pues apuntaremos al elemento siguiente anterior. Los programadores clásicos de C usaban estas características para desplazarse por la memoria de un programa intentando aumentar su eficiencia temporal (rapidez) pero quedando códigos de tan poco aconsejable aspecto como el siguiente:

```

void imprimir_mayusculas(TPCadena cadena)
{ while (cadena!= NULL)
  { cout << toupper(*cadena);
    ++cadena;
  }
}

```

Suponga ahora que se desea programar una serie de funciones para trabajar con matrices. Una primera solución sería definir una estructura de datos matriz, compuesta por una matriz y sus dimensiones puesto que interesa que las funciones trabajen con matrices de cualquier tamaño. Por tanto la matriz dentro de la estructura tendrá el tamaño máximo de todas las matrices con las que se desea trabajar y como se tiene almacenadas las dimensiones se trabaja con una matriz de cualquier tamaño pero tendrá reservada memoria para una matriz de tamaño máximo.

Se desperdicia memoria. Una definición de este tipo sería:

```

typedef struct
  { float mat[1000][1000];
    int ancho,alto;
  } MATRIZ;

```

En principio esta es la única forma de definir el tipo de datos. Con esta definición una matriz 3x3 ocupará 1000x1000 floats al igual que una matriz 50x50.

³ Si bien esta comparación no es necesaria, es conveniente su uso para evitar errores producidos por datos "corruptos".

Sin embargo se puede asignar memoria dinámicamente a la matriz y reservar sólo el tamaño que se requiera. La estructura sería ésta.

```
struct mat
{   float *datos;
    int   ancho,alto;
};

typedef struct mat *MATRIZ;
```

El tipo MATRIZ ahora debe ser un puntero puesto que se tiene que reservar memoria para la estructura que contiene la matriz en sí y las dimensiones. Una función que pueda crear una matriz sería así:

```
MATRIZ inic_matriz (int x,int y)
{ MATRIZ temp;
  temp = (MATRIZ) malloc (sizeof(struct mat));
  temp->ancho = x;
  temp->alto = y;
  temp->datos = (float *) malloc (sizeof(float)*x*y);
  return temp;
}
```

En esta función se ha obviado el código que comprueba si la asignación ha sido correcta. Observe como se organizan los datos con estas estructuras.

```
temp----->datos----->x*x elementos
      ancho,alto
```

Esta estructura puede parecer en principio demasiado compleja, pero verá que es muy útil en el encapsulado de los datos.

En el programa principal, para utilizar la matriz declararía algo así:

```
main()
{ MATRIZ a;
  a = inic_matriz (3,3);
  ...
  borrar_matriz(a);
}
```

Dónde borrar_matriz(a) libera la memoria reservada en inic_matriz, teniendo en cuenta que se realizaron dos asignaciones, una para la estructura mat y otra para la matriz en sí.

Otra definición posible del problema podría ser así.

```
typedef struct mat MATRIZ;
void inic_matriz (MATRIZ *p,int x,int y)
{ p->ancho = x;
  p->alto = y;
  p->datos = (float *)malloc(sizeof(float)*x*y);
}
```

Con este esquema el programa principal sería algo como esto:

```
main()
{ MATRIZ a;
  inic_matriz (&a,3,3);
  .....
  borrar_matriz (&a);
}
```

Con este esquema el acceso a la matriz sería $*(a.datos+x*y*a.ancho)$, idéntico al anterior sustituyendo los puntos por flechas $->$. En el siguiente capítulo se justificará la utilización de la primera forma de implementación frente a esta segunda. En realidad se trata de la misma implementación salvo que en la primera el tipo MATRIZ es un puntero a una estructura, por tanto es necesario primero reservar memoria para poder utilizar la estructura, mientras que en la segunda implementación, el tipo MATRIZ es ya en si la estructura, con lo cual el compilador ya reserva la memoria necesaria. En el primer ejemplo MATRIZ a; define a como un puntero a una estructura struct mat, mientras que en la segunda MATRIZ a; define a como una variable cuyo tipo es struct mat.

Resumen

int *p;	p es un puntero a un entero
int *p[10];	p es un array de 10 punteros a enteros
int (*p)[10];	p es un puntero a un array de 10 enteros
int *p(void);	p es una función que devuelve un puntero a entero
int p(char *a);	p es una función que acepta un argumento que es un puntero a carácter, devuelve un entero
int *p(char *a);	p es una función que acepta un argumento que es un puntero a carácter, devuelve un puntero a entero
int (*p)(char *a);	p es un puntero a función que acepta un argumento que es un puntero a carácter, devuelve un puntero a entero

int (*p(char *a))[10];	p es una función que acepta un argumento que es un puntero a carácter, devuelve un puntero a un array de 10 enteros
int p(char (*a)[]);	p es un puntero a función que acepta un argumento que es un puntero a un array de caracteres, devuelve un puntero a entero
int p(char *a[]);	p es un puntero a función que acepta un argumento que es un array de punteros a caracteres, devuelve un puntero a entero
int *p(char a[]);	p es una función que acepta un argumento que es un array de caracteres, devuelve un puntero a entero
int *p(char (*a)[]);	p es una función que acepta un argumento que es un puntero a un array de caracteres, devuelve un puntero a entero
int *p(char *a[]);	p es una función que acepta un argumento que es un puntero a un array de punteros a caracteres, devuelve un puntero a entero
int (*p)(char (*a)[]);	p es una función que acepta un argumento que es un puntero a un array de caracteres, devuelve un puntero a entero
int *(*p)(char (*a)[]);	p es un puntero a una función que acepta un argumento que es un puntero a un array de punteros a caracteres, devuelve un puntero a entero
int *(*p)(char *a[]);	p es un puntero a una función que acepta un argumento que es un array de punteros a caracteres, devuelve un puntero a entero
int (*p[10])(void);	p es una array de 10 punteros a función, cada función devuelve un entero
int (*p[10])(char *a);	p es una array de 10 punteros a función; cada función acepta un argumento que es un puntero a carácter y devuelve un entero.
int *(*p[10])(char a);	p es una array de 10 punteros a función; cada función acepta un argumento que es un carácter, y devuelve un puntero a entero.
char *(*p[10])(char *a);	p es una array de 10 punteros a función; cada función acepta un argumento que es un carácter, y devuelve un puntero a caracter.