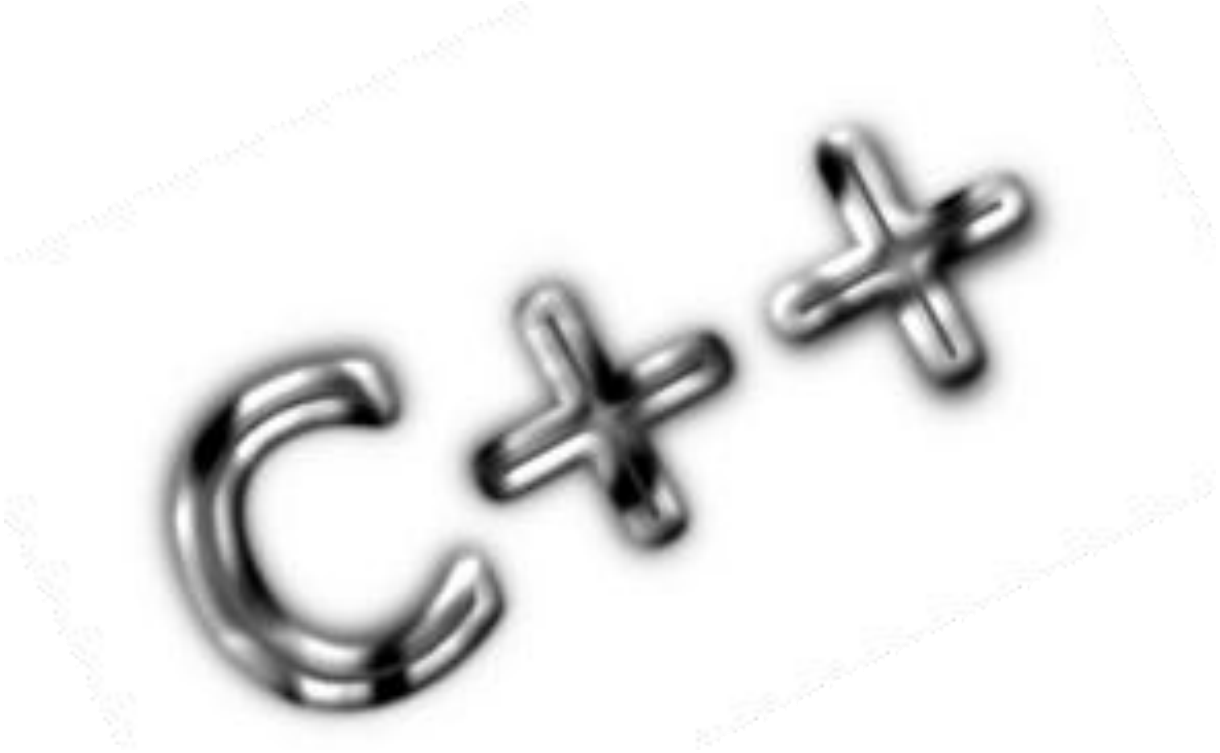


PUNTEROS O APUNTADORES C++

TRANSVERSAL DE PROGRAMACIÓN BÁSICA

INGENIERÍA DE SISTEMAS

En el presente documente se hace una breve explicación a la gestión dinámica de memoria en el lenguaje C++. Se incluyen los aspectos generales de operaciones con punteros y la creación y manejo básico de listas enlazadas.

CRISTIAN GUILLERMO GARCÍA
MONITORIA 2012-3 | Universidad Distrital francisco José de Caldas

PUNTEROS O APUNTADORES C++

Tabla de contenido

PUNTEROS O APUNTADORES C++

1. INTRODUCCIÓN

El presente documento pretende servir de guía para la enseñanza en la programación bajo el lenguaje C++. Pese a que se ha desarrollado siguiendo los lineamientos establecidos en el syllabus de la asignatura transversal de programación básica de la universidad distrital Francisco José de Caldas, es posible utilizarla como un manual en cualquier campo o ámbito siempre y cuando se relacione con el aprendizaje del lenguaje mencionado.

Es necesario resaltar que este documento es completamente tomado de [1] de quien no se precisa autor. Sin embargo, se han hecho algunas modificaciones en cuanto al formato de presentación.

2. APUNTADORES O PUNTEROS

Los tipos de datos vistos hasta ahora, tanto los simples como los estructurados, sirven para describir datos o estructuras de datos cuyos tamaños y formas se conocen de antemano. Sin embargo, hay muchos programas cuyas estructuras de datos pueden variar de forma y tamaño durante la existencia del mismo.

Las variables de todos los tipos de datos vistos hasta ahora son variables estáticas, en el sentido de que se declaran en el programa, se designan por medio del identificador declarado, 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 esta variable.

Sin embargo C++, permite la posibilidad de crear o destruir variables en tiempo de ejecución, a medida que van siendo necesitadas durante la ejecución del programa. 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.

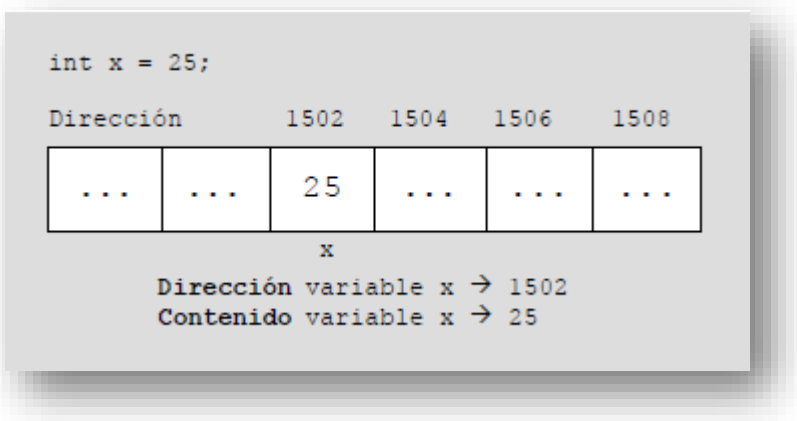
2.1. El tipo de Datos Puntero

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

PUNTEROS O APUNTADORES C++

Definición: 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.

2.2. Declaración de variables Puntero

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

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

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

```
typedef int *PuntAInt;// Tipo puntero a enteros (int)
typedef char *PuntACar;// Tipo puntero a caracteres (char)
```

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 PuntAInt
PuntACar carácter; // carácter es una variable de tipo PuntACar
```

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.

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 Complex
{
  float parte_real;
  float parte_imag;
};

typedef Complex *TPComplex; //Tipo puntero a un valor de tipo Complex
```

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

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

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

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 *caracter;// puntero a una variable de tipo caracter (char)
Complex *ptr1;// ptr1 es un puntero a una variable de tipo Complex
```

Siempre que sea posible evitaremos las declaraciones anónimas, y asociaremos un nombre de tipo a las variables que declaremos.

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

2.3. 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

2.3.1. Operadores específicos de punteros

Al trabajar con punteros se emplean dos operadores específicos:

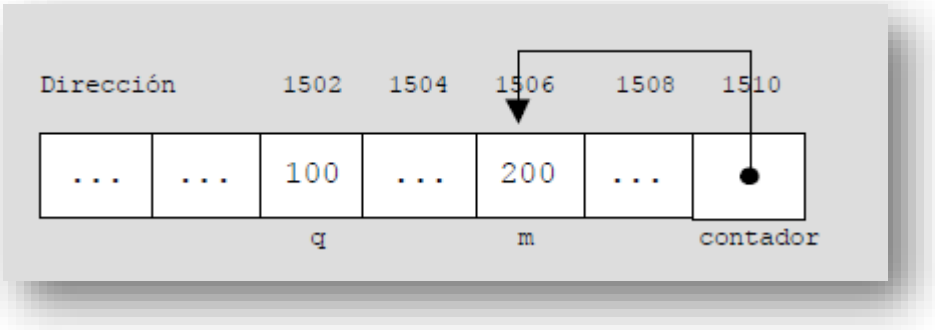
Operador de dirección: **&** Es un operador monario (sólo requiere un operando) que devuelve la dirección de memoria del operando.

Por ejemplo,

```
typedef int *PuntAInt;
int q,m;          // q y m son variables de tipo entero (int)
PuntAInt 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`.

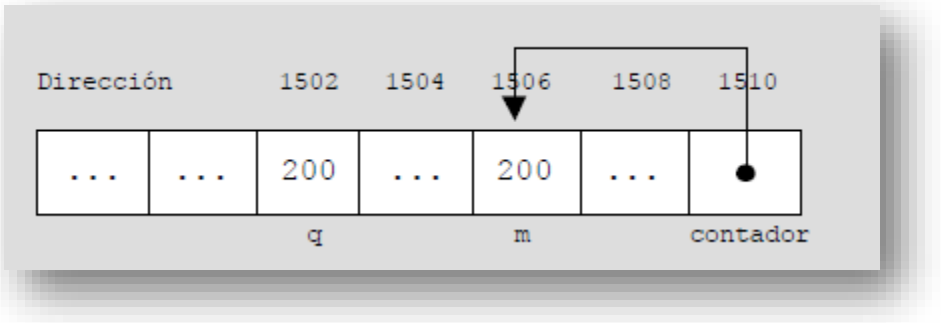


Operador de contenido o indirección: ***** Este operador es el complementario de &. Es un operador monario que devuelve el valor de la variable ubicada en la dirección que se especifica (contenido de la variable anónima apuntada por el puntero).

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;)

2.3.2. 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 ningúnobjeto 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, muchos de los archivos de cabecera de C++, como `<cstdio>`, 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`).

2.3.3. 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;
}
```

3. ASIGNACIÓN DINÁMICA DE MEMORIA

Cuando hablamos de *asignación dinámica* de memoria, nos referimos 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.

PUNTEROS O APUNTADORES C++

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*.

3.1. Funciones de Reserva de Memoria

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 cuál 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
}
```

3.2. Función de Liberación de Memoria Dinámica

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.

3.3. Ejemplo de Asignación de Memoria Dinámica

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

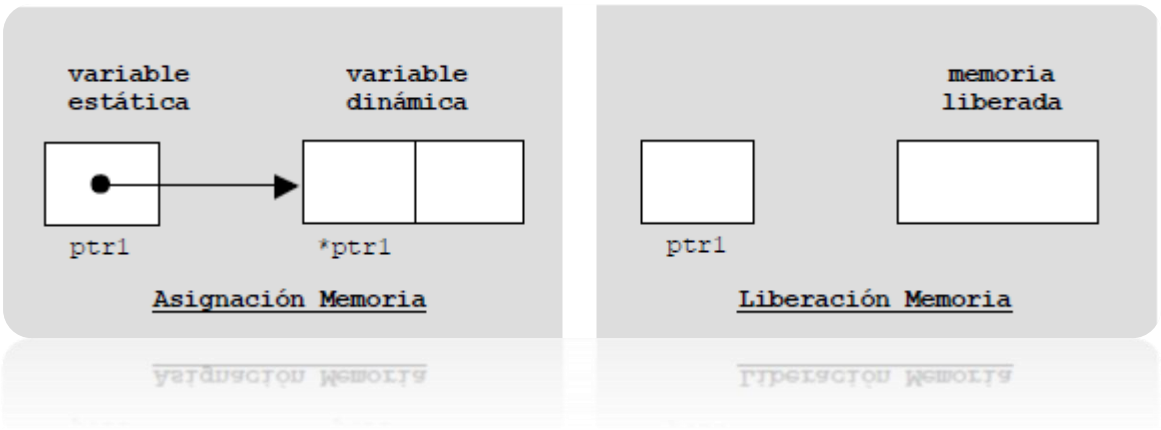
Para crear una variable anónima de tipo `Complex` hacemos una llamada a `new` que creará una variable anónima de tipo `Complex` apuntada por el puntero `ptr1`, donde ptr1 es un puntero del tipo definido anteriormente `TPComplex` (Tipo puntero al tipo registro `Complex`).

```
TPComplex ptr1;
ptr1 = new Complex;
```

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

```
delete ptr1;
```

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



4. 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 2.3 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 podemos 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:

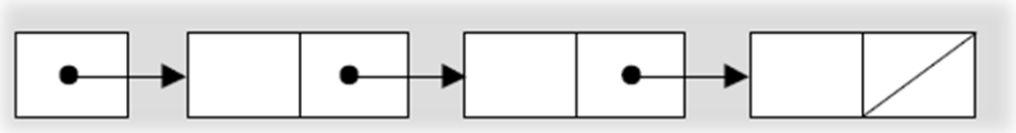
```
P -> edad = 15; // accede al campo edad del registro.
```

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

5. 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).

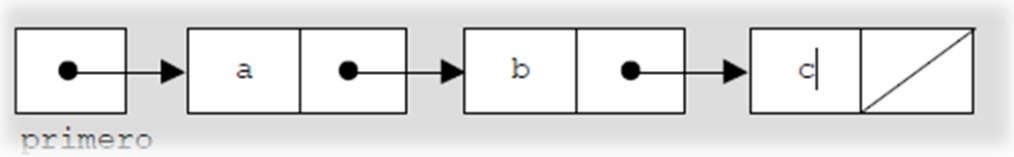
```
struct TNodo
{
  char c;
  TNodo *sig;
};
typedef TNodo *TLista;
```

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

Ilustración 1 Posibilidades (Equivalentes) de defirnir 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 los campos p untero. 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

5.1. 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;
}
```

5.2. Insertar Nodos en listas enlazadas

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 `TNodo`.

```
punt = new TNodo;
```

2. Una vez que hemos creado un nodo apuntado por `punt`, le asignamos 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 &primero[1], char car)
{
   // Esta función inserta un nodo al principio de la lista
   TLista punt;

   punt = new TNodo;
   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. 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):

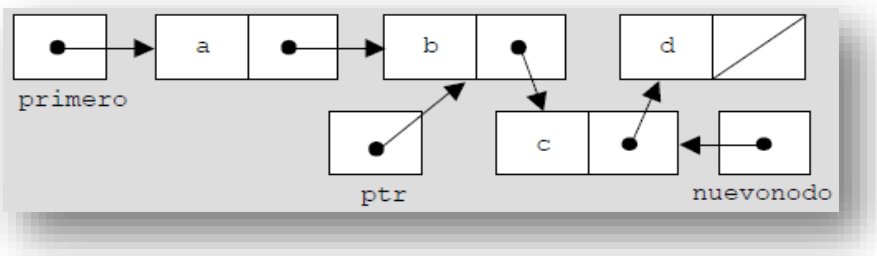
1. Hemos de crear un nodo nuevo, y en su campo `c` almacenar el valor del carácter que queremos insertar. Para ello creamos un nodo apuntado por la variable puntero `nuevonodo`, y le asignamos al campo c el valor del carácter leído (por ejemplo, "`c`").

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

2. Necesitamos 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 TNodo;
      if (nuevonodo == NULL)
      {
         cout << "No hay memoria suficiente" << endl;
      }

else
      {
         nuevonodo->c   = c;
         nuevonodo->sig = NULL;
         if (primero == NULL)
         {
            /* La lista estaba vacía y nuevonodo se convierte en el
               primer y único elemento de ésta */
            primero = nuevonodo;
         }
         else
         {
            if (nuevonodo->c <= primero->c)
            {
              /* El elemento a insertar es menor que el primer elemento
                 de la lista*/
              nuevonodo->sig = primero;
              primero = nuevonodo;
            }
            else
            {
              ptr = primero;
              while ((ptr->sig != NULL) &&
                     (nuevonodo->c > ptr->sig->c))
              {
                 ptr = ptr->sig;
              }
              nuevonodo->sig = ptr->sig;
              ptr->sig = nuevonodo;
            }
         }
      }
   }
```

5.3. Borrar Nodos

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 enlazada

Cuando queremos borrar el primer nodo de la lista enlazada:

- Necesitamos un puntero auxiliar `punt` que me apunte al nodo a borrar, que en este caso será el primero**.**

```
punt = primero;
```

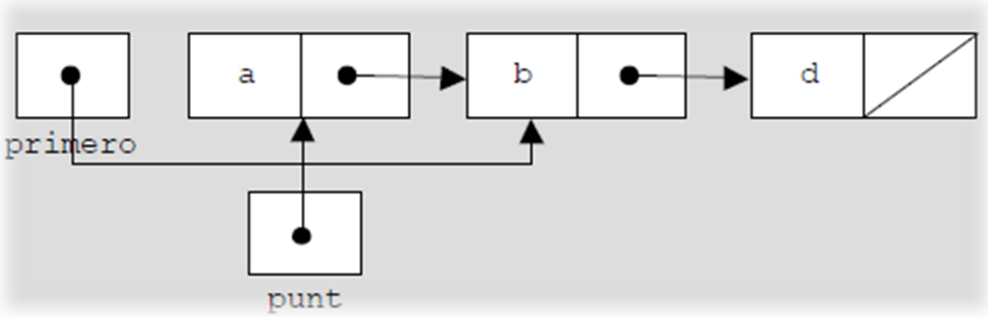
- Actualizar el valor de la variable puntero `primero`, de modo que apunte al siguiente elemento de la lista, pues el primero será eliminado.

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

- Liberar la memoria ocupada por la variable anónima a la que apunta la variable puntero `punt`, que es el nodo que queremos eliminar.

```
delete punt;
```

Gráficamente, quedaría del modo siguiente:

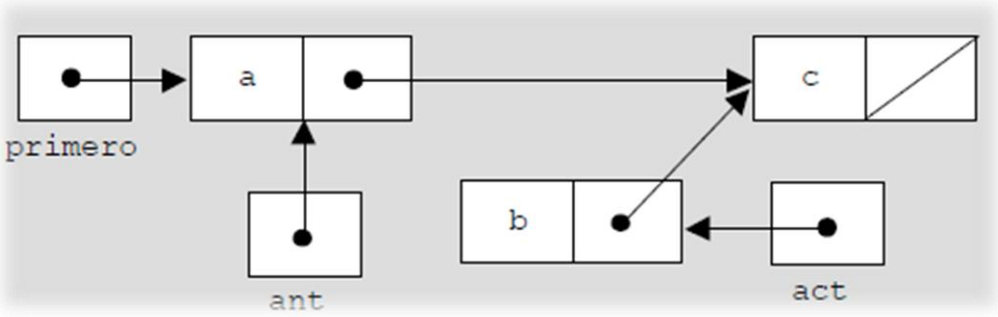


```
void borrarFrente(TLista &primero)
{
   // Este procedimiento borra el primer nodo de la lista

   // declaración variables
   TLista punt;

   if (primero != NULL)
   {
      punt = primero;
      primero = primero->sig;
      delete punt;
   }
}
```

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`.

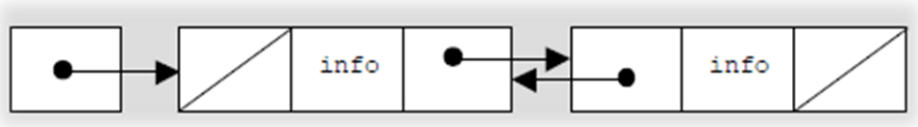
Si quisiéramos borrar el segundo nodo (nodo que contiene “b” en la figura) de una lista ordenada cuyo primer elemento viene apuntado por `primero`, tendríamos lo siguiente:



```
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;
  }
}
```

6. 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.



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

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

6.1. 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;
}
```

6.2. Insertar Nodo

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:

PUNTEROS O APUNTADORES C++

```
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;
              }
           }
         }
      }
   }
```

6.3. Borrar Nodo

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`.

PUNTEROS O APUNTADORES C++

```
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;
   }
}
```

7. REFERENCIAS

[1] Departamento de Lenguajes y Ciencias de la Computación, 2005-2006. [En línea]. Available: http://es.scribd.com/doc/94274637/tema2. [Último acceso: 08 11 2012].