TEMA 3: ARITMÉTICA Y TIPOS DE DATOS EN EL COMPUTADOR. PARTE II

Introducción

En la primera parte del tema se vieron los tipos simples de datos que es capaz de almacenar el ordenador:

Booleanos, caracteres, enteros y reales.

En general, la información tratada por el ordenador irá agrupada de una forma más o menos coherente en estructuras especiales, compuestas por datos simples. A este tipo de agrupaciones las denominaremos tipos de datos estructurados o tipos compuestos.

Los tipos de datos estructurados o tipos compuestos se distinguen por los elementos que las componen y por las operaciones que se pueden realizar con ellas o entre sus elementos.

En ocasiones interesará manejar las estructuras como elementos únicos, y otras nos interesará manejar los elementos que componen los tipos de datos estructurados como elementos separados.

Clasificaciones de las 'agrupaciones' de datos

Agrupaciones contiguas/Agrupaciones enlazadas

Las agrupaciones contiguas son aquellas que ocupan posiciones sucesivas o contiguas de bits en memoria. La posición de un elemento dentro de la agrupación se puede calcular sumando al comienzo de la agrupación una cantidad determinada.

Las agrupaciones enlazadas son aquellas que tienen la información dispersa en la memoria, y la manera de acceder desde un elemento a otro es mediante la dirección en memoria dónde está el siguiente elemento.

Agrupaciones homogéneas/Agrupaciones heterogéneas

Las agrupaciones homogéneas son aquellas que guardan grupos de elementos iguales (todos los elementos guardados son enteros, reales,...)

Las agrupaciones heterogéneas son las que agrupan elementos (campos) de diferentes tipos.

Agrupaciones estáticas/Agrupaciones dinámicas

Las agrupaciones estáticas son las que guardan un número predeterminado de elementos, que no puede ser modificado en el momento en que se ejecuta el programa. Hay que elegir el número cuando escribimos el programa, antes de compilarlo.

Las agrupaciones dinámicas son aquellas que pueden ir modificando el número de elementos que contienen a medida que va siendo necesario guardar más información.

En este tema veremos diferentes agrupaciones de datos, desde los más simples a los más complejos: *arrays* (vectores o *arrays* unidimensionales, matrices o *arrays* multidimensionales y cadenas), registros o estructuras y listas dinámicas.

Las agrupaciones, en general, no son exclusivas de manera que podremos encontrar combinaciones de ellas si es necesario.

Arrays

Los *arrays* son agrupaciones homogéneas, contiguas y estáticas. Los elementos que contienen son todos iguales y el número de elementos que podemos guardar se define cuando escribimos el programa.

Podemos distinguir entre los siguientes tipos de *arrays*: vectores (arrays unidimensionales), matrices (arrays bidimensionales) y arrays multidimensionales.

Vectores (o arrays *unidimensionales)*

Son agrupaciones en las que cada elemento tienen asociado un índice (un entero), de manera que se puede acceder a cada uno de los elementos mediante la utilización de ese índice.

Este tipo de datos estructurados sirven para agrupar variables de un mismo tipo con un único nombre.

Supongamos que queremos declarar 10 variables de tipo entero. La única forma de hacerlo hasta ahora sería declararlos como variables individuales:

```
int n1, n2, n3, n4, n5, n6, n7, n8, n9, n10;
```

Otra forma de hacerlo es utilizando un vector.

La declaración de un vector en C/C++ se realiza de la siguiente manera:

```
Tipo Nombre[Tamaño];
```

Donde **`Tipo`** es el tipo de los datos guardados en el vector, **`Nombre`** el nombre que le vamos a dar al vector y **`Tamaño`** el número de elementos que es capaz de guardar el vector.

Es importante destacar que en C/C++, los índices comienzan en **`0`** y terminan en **`Tamaño – 1`**.

Ejemplo:

Un vector capaz de guarda 10 enteros, que se llamase vect se declararía:

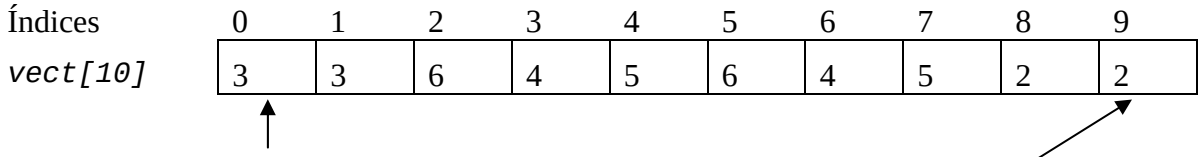
```
int vect[10];
```

De esta forma tenemos 10 enteros agrupados que se pueden tratar como una única variable.

Una cosa que debe quedar clara en la declaración, es que sólo se reserva espacio de memoria. NO se pone ningún valor en el vector.

Podemos acceder a cada uno de los elementos de un vector utilizando un índice. A esta operación se la denomina **indexación**. Este índice indica la posición del elemento dentro del vector.

En C/C++, los índices comienzan en **`0`** y terminan en **`(Tamaño – 1)`** por tanto.



`vect[0]` Primer elemento del vector

`vect[1]` Segundo elemento del vector

…

`vect[9]` Último elemento del vector. (Tamaño-1).

Operaciones:

Asignación

Se da un valor determinado a algún elemento del vector. La manera de hacerlo es mediante el nombre del vector y el índice que queremos asignar:

```
Nombre[indice] = Valor;
```

Ejemplo:

Para asignar el valor 4 al índice (o posición del vector) ocho:

```
vect [8] = 4;
```

NO se pueden realizar asignaciones entre vectores. NUNCA se debe realizar la siguiente asignación.

```
int a[10], b[10];
a=b;      // ¡¡¡ERROR!!!
```

Recorrido

Se pasa por los elementos del vector para realizar una tarea concreta en cada elemento.

Recorrido completo

```
Desde i ← 0 hasta Tamaño -1 hacer
  Procesar (Nombre [i])
  i ← i + 1
Fin_desde
```

Ejemplo: El recorrido completo del vector sería, en C/C++:

```
for (i = 0; i < 10; i++)
   Procesar (vect[i]);
```

Recorrido parcial

```
i ← 0
Mientras (Nombre[i] cumpla una cierta condición)
  Procesar (Nombre [i])
  i ← i + 1
Fin_Mientras
```

Ejemplo: Un posible recorrido parcial en C++:

```
i=0;
while (vect[i]!=2)
{
   vect[i]=0;
   i++;
}
```

Iniciación

Una cosa que debe quedar clara en la declaración, es que sólo se reserva espacio. NO se pone ningún valor en el vector. La iniciación de los elementos del vector la debe realizar el programador.

Para hacer la iniciación de todos y cada uno de los elementos del vector, habría que recorrerlo y asignar o pedir al usuario que introdujese cada uno de los elementos.

Ejemplo:

Supongamos que queremos pedir al usuario que nos dé todos los valores del vector:

```
for (i = 0; i < 10; i++)
   cin >> vect[i];
```

Supongamos que queremos poner todos los elementos del vector a cero:

```
for (i = 0; i < 10; i++)
   vect[i] = 0;
```

Para facilitar la realización de bucles y evitar posibles errores en el recorrido de los vectores se recomienda que el número de elementos que posee un vector se defina previamente como una constante.

Ejemplo 1: Realizar una función que rellene los elementos de un vector.

```
#define TAM 100

void RellenarVector (int vect[TAM])
{
   int i;

   for (i = 0; i < TAM; i++)
        cin >> vect[i];
}
```

Es importante destacar que los vectores cuando van a ser modificados (paso de parámetros por referencia) no se realiza nada especial, por el hecho de que en C/C++ SIEMPRE se pasan por referencia.

Ejemplo 2: Realizar una función que calcule la media de los elementos contenidos en un vector.

```
#define TAM 100

float MediaVector (int vect[TAM])
{
   int i;
   int acc = 0;

   for (i = 0; i < TAM; i++)
      acc = acc + vect[i];

   return ( float (acc) / TAM);
}
```

Búsqueda en vectores

Para buscar un cierto elemento en un vector, en principio hay que ir recorriendo todos y cada uno de los elementos, en busca del que se busca. A esto se le llama en general búsqueda secuencial.

Ejemplo: Realizar una función que diga si un cierto valor 'x' se encuentra o no en un vector 'vect'.

```
int Buscar1 (int vect[TAM], int x)
{
   int i, encontrado = 0;

   for (i = 0; i < TAM; i++)
      if (vect[i] == x)
         encontrado = 1;

   return (encontrado);
}
```

De todas formas este procedimiento puede mejorarse si cuando encontramos el elemento salimos. Es más, podemos incluir la comparación en la condición del bucle:

```
int Buscar2 (int vect[TAM], int x)
{
   int i;

   i = 0;
   while ( (i < TAM) && (vect[i] != x) )
         i++;

   if (i == TAM)
      return (0);
   else
      return (1);
}
```

Podemos evitar la comparación de '**i < TAM**' si estamos seguros de que el elemento '**x**' está en el vector. Y podemos estar seguros si lo ponemos al final del vector. A esta forma de buscar se le llama búsqueda secuencial con centinela.

```
int Buscar3 (int vect[TAM + 1], int x)
{
   int i;

   vect[TAM] = x;

   i = 0;
   while (vect[i] != x)
         i++;

   if (i == TAM)
      return (0);
   else
      return (1);
}
```

Si el vector estuviese ordenado se puede aplicar una técnica especial de búsqueda llamada búsqueda binaria o dicotómica:

1. Si mira el elemento central
2. Si es el elemento buscado terminar
3. Si no lo es:
   3.a. Determinar en que 'zona' del vector está (por arriba o por debajo del central)
   3.b. Repetir el proceso en la nueva zona.

```
int Buscar4 (int vect[TAM], int x)
{
   int primero, ultimo, central;

   primero = 0;
   ultimo = TAM – 1;
   central = (primero + ultimo) / 2;

   while ( (primero < ultimo) && (vect[central] != x) )
   {
      if (x < vect[central])
         ultimo = central - 1;
      else
         primero = central + 1;
      central = (primero + ultimo) / 2;
   }

   if (x == vect[central])
      return (1);
   else
      return (0);
}
```

Veamos cuál sería el programa principal que llamara a alguna de las distintas funciones buscar que hemos visto.

```
#define TAM 100
main (void)
{
    int v[TAM],x;

    encontrado= buscar(v,x);
     if (encontrado)
        cout <<"Se ha encontrado el elemento"<< x << endl;
     else
        cout <<"El elemento"<< x << "no se encuentra en el vector";
}
```

Matrices (o arrays *bidimensionales)*

Son agrupaciones similares a los vectores pero en las que cada elemento tiene asociados dos índices enteros, de manera que se puede acceder a cada uno de los elementos mediante la utilización de esos índices.

La declaración de una matriz en C/C++ se realiza de la siguiente manera:

```
Tipo Nombre[Tamaño1][Tamaño2];
```

Donde **`Tipo`** es el tipo de los datos guardados en la matriz, **`Nombre`** el nombre que le vamos a dar a la matriz y **`Tamaño1`** y **`Tamaño2`** el número de filas y columnas que tiene la matriz.

Operaciones:

Asignación

Se da un valor determinado a algún elemento de la matriz. La manera de hacerlo es mediante el nombre de la matriz y los índices que queremos asignar:

```
Nombre [indice1][indice2] = Valor;
```

Ejemplo:

Para asignar el valor 4 a los índices (o posición de la matriz) ocho, cinco:

```
mat[8][5] = 4;
```

Recorrido

Se pasa por los elementos de la matriz para realizar una tarea concreta en cada elemento.

Recorrido completo

```
Desde1 i ← 0 hasta Tamaño1 -1 hacer
  Desde2 j ← 0 hasta Tamaño2 - 1 hacer
    Procesar (Nombre [i][j])
    j ← j + 1
  Fin_desde2
  i ← i + 1
Fin_desde1
```

Ejemplo:

El recorrido completo de una matriz de diez por veinte elementos sería en C/C++:

```
int mat[10][20];

for (i = 0; i < 10; i++)
   for (j = 0; j < 20; j++)
      Procesar (mat[i][j]);
```

Iniciación

Para hacer la iniciación de todos y cada uno de los elementos de la matriz, habría que recorrerla y asignar o pedir al usuario que introdujese cada uno de los elementos.

Ejemplo:

Supongamos que queremos pedir al usuario que nos dé todos los valores de la matriz:

```
for (i = 0; i < 10; i++)
   for (j = 0; j < 20; j++)
      cin >> mat[i][j];
```

Supongamos que queremos poner todos los elementos del vector a cero:

```
for (i = 0; i < 10; i++)
   for (j = 0; j < 20; j++)
      mat[i][j] = 0;
```

El resto de operaciones serían similares a las vistas en vectores, pero siempre teniendo en cuenta que el acceso a cada uno de los elementos de la matriz se realiza mediante la utilización de dos índices.

Arrays *multidimensionales*

Son agrupaciones similares a los vectores pero en las que cada elemento tiene asociados más de dos índices enteros, de manera que se puede acceder a cada uno de los elementos mediante la utilización de esos índices.

La declaración de un array multidimensional en C/C++ se realiza de la siguiente manera:

```
Tipo Nombre[Tamaño1][Tamaño2]...[TamañoN];
```

Donde **Tipo** es el tipo de los datos guardados en el *array* multidimensional, **Nombre** el nombre que le vamos a dar y **Tamaño1, Tamaño2,...** el tamaño de las diferentes dimensiones del *array*.

Las operaciones que se puedan definir serán similares a las vistas en vectores o matrices pero ampliando el número de índices de forma adecuada, y recordando que para acceder a un elemento concreto hay que determinar el valor de todos y cada uno de los índices.

Paso de arrays como parámetros

Los arrays, como cualquier otro tipo de datos, se pueden pasar como parámetros a una función. En C++ siempre que se pasa un array a una función se pasa por REFERENCIA, aunque no pongamos nada especial en el parámetro.

Ejemplo 1: Realizar una función que rellene los elementos de un vector.

```
#define TAM 100

void RellenarVector (int vect[TAM])
{
   int i;
   for (i = 0; i < TAM; i++)
      cin >> vect[i];
}
```

El vector *vect* en este ejemplo está pasado por referencia, no es necesario escribir el &.

Al pasarse exclusivamente por referencia, el paso de arrays como parámetros es más eficiente, puesto que no hay que copiar cada vez toda la matriz (que normalmente es bastante grande). El inconveniente es que dentro de la función se puede modificar el array aunque no se quisiera.

Ejemplo 2: Realizar una función que calcule la media de los elementos contenidos en un vector.

```
#define TAM 100

float MediaVector (int vect[TAM])
{
   int i, acc = 0;
   for (i = 0; i < TAM; i++)
      acc = acc + vect[i];
   return ( (float)acc / TAM);
}
```

Representación en memoria de un array

Los elementos de un vector se almacenan en memoria de forma contigua, es decir, uno al lado del otro.

Ejemplo:

```
int Vect[10]; //Si empieza en la dirección de memoria 10:
```

Por lo tanto, para calcular la posición en memoria de un elemento se utiliza la fórmula:

$d = d_o + Ind$ * `sizeof` (*Elemento*)

donde:

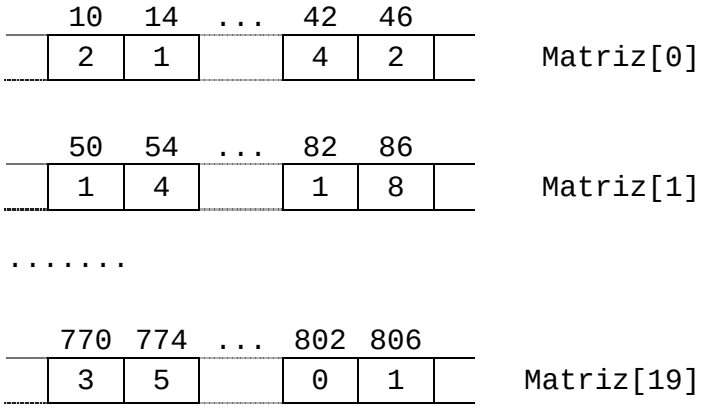
d_o es la dirección de memoria donde empieza el vector `Ind` es el índice del elemento del vector del que queremos saber su dirección, y *Elemento* es el tipo de elementos que contiene el vector.

`sizeof` devuelve el tamaño en bytes de una variable o tipo de dato. `sizeof(char)`, por ejemplo devolverá `1`.

Puesto que una matriz no es más que un vector de vectores, el almacenamiento en memoria es también contiguo:

Ejemplo:

```
int Matriz[20][10];   //Si empieza en 10 y sizeof(int)=4 bytes:
```



Para calcular la posición del elemento Matriz[19][1], lo que hay que hacer es primero resolver el primer índice (Matriz[19]), que tiene como elementos vectores enteros de 10 elementos, y después el segundo tomando como dirección inicial la que resulta del calculo anterior. Así, si el vector empieza por ejemplo en la posición 10, la fórmula será:

```
d = 10 + 19 * sizeof(int [10]) + 1 * sizeof(int)
  = 10 + 760 + 4 = 774
```

Uso de arrays con número de elementos variable

Como ya se ha comentado, el tamaño de un *array* ha de ser una constante, por lo tanto no es posible modificar el tamaño de un *array* después de haber compilado el programa. Para solucionar esto existen dos posibilidades: la primera es utilizar memoria dinámica, que veremos en el último apartado del tema, la segunda es declarar un *array* lo suficientemente grande y utilizar sólo una parte de este *array*.

Por ejemplo, si vamos a realizar un programa para sumar vectores matemáticos y sabemos que en ningún caso vamos a tener vectores de un tamaño mayor de 10, podemos declarar todos los vectores de

tamaño 10 y después utilizar sólo una parte del vector. Para ello necesitaremos utilizar también una variable entera extra para saber el tamaño actualmente utilizado por el vector, puesto que todas las operaciones se realizarán considerando el tamaño actual del vector y no el total.

Los inconvenientes de este método son que estamos desperdiciando memoria y además que no siempre hay un límite conocido al tamaño del *array*.

Cadenas y strings

Hablaremos de *cadenas* como un caso especial de vectores en el que guardamos caracteres. Una cadena en C, no es más que un vector que contiene caracteres, pero que tiene como característica especial que sólo se utiliza una parte del vector, de manera que se coloca un delimitador al final de los caracteres utilizados, que en C es el carácter con valor 0 (habitualmente indicado como '\0').

Por ejemplo: Supongamos una cadena de 10 caracteres, en la que deseamos guardar la palabra 'hola'. De los diez posibles caracteres, sólo vamos a utilizar cuatro. El contenido del vector será pues:

0	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**
'h'	'o'	'l'	'a'	'\0'	?	?	?	?	?

La declaración de una cadena se realizará como la de un vector en el que vayamos a guardar caracteres:

```
char Nombre [Tamaño];
```

Ejemplo:

Una cadena capaz de guardar 9 caracteres válidos:

```
char cad [10];
```

Recordar que hay que tener en cuenta que el delimitador ocupa una posición.

Teniendo en cuenta que una cadena no es más que un caso especial de vector, las operaciones sobre las cadenas se realizarían elemento a elemento al igual que en el caso general de vectores, además de considerar el elemento de final de cadena. Para ayudar a la realización de operaciones, en C existen funciones específicas que permiten copia cadenas, concatenarlas, etc., pero aún así son algo engorrosas. Por ello, casi todos los lenguajes definen un tipo nuevo para manejar cadenas de caracteres. Este tipo es básicamente un vector de caracteres que lleva asociado un entero y un conjunto de funciones que permiten realizar operaciones de cadenas. El entero representa la longitud actual de la cadena.

Un *string* es un tipo de dato definido en C++ que permite almacenar un conjunto de caracteres. El tamaño de dicho tipo de datos es siempre el del número de caracteres que almacena. El concepto de string es equivalente al de cadena pero las funciones que permiten realizar operaciones sobre ellos son más sencillas de manejar y además la longitud de la cadena no se determina por ningún carácter especial, por lo que siempre que manejemos variables de tipo *string* lo haremos a través de las operaciones sobre *string*.

Para utilizar la clase `string` de C++ tenemos que incluir la librería `string` en el programa principal.

```
#include <string>
```

La forma de utilizar estas funciones que operan sobre las variables de tipo *string* es poniendo el nombre de la variable seguido de un punto y del nombre de la función a usar. Es decir:

```
VarTipoString.NombreFuncion(parametros);
```

La declaración de una variable tipo string, en el caso de no asignarle un valor inicial, es:

```
string nombre;
```

La declaración de una variable tipo string, en el caso de asignarle un valor inicial, es:

```
string nombre ("valor inicial para la variable");
```

Ejemplo:

Un string que almacena inicialmente los caracteres “Hola a todo el mundo!”:

```
string cad (“Hola a todo el mundo!”);
```

Operaciones sobre strings

Asignación

Guarda una cadena de caracteres en una variable de tipo *string*

```
string cad; //Definición de la variable
cadena.assign("le asignamos una serie de caracteres");
```

También podemos usar el operador asignación:

```
cad = "le asignamos una serie de caracteres";
string cad2 = cad; //Copio el string cad en cad2
```

Concatenar strings

```
string cad1("El principio");
string cad2("el final");
cad2.insert(0,cad1); //Añade la cad1 al principio (lugar 0) de cad2
```

También lo podemos hacer usando el operador suma, ya que:

***s* + *t* devuelve un nuevo string que resulta de copiar s y concatenar t detrás y,**
***s* = *t* hace una copia de t y se la asigna a s.**

```
cad2 = cad1 + cad2;
```

Longitud de un string

```
cad.length(); // cantidad de caracteres del string cad
```

Obtener una subcadena de una cadena dada

```
s.substr(a, b);
// devuelve un string copia del substring de s desde a, con b caracteres.

string str = "Hello World!";
cout << str.substr(0, 5) << endl; //muestra "Hello".
cout << str.substr(6) << endl; //muestra "World!"
cout << str.length() << endl; //muestra la longitud, esto es "12".
```

Mostrar un elemento determinado del string

```
str[a]; // devuelve el char de la posición a en el string str.
cout << str[str.length() - 1] << endl;//muestra el último carácter "!"
```

Comparación lexicográfica de strings

Se hace directamente con los operadores (**==**, **!=**, **<**, **>**)

```
if (str == cad)
    cout<<"SI son iguales\n";
else
    cout << "No lo son\n";
```

Borrar caracteres de una cadena dada

```
s.erase (a, b);
// borra b caracteres del string s desde la posición a.

frase.erase (3,7); // borra 7 caracteres de frase desde la posición 3
```

Reemplazar caracteres de una cadena dada

```
s.remplace (a, b, nuevo_s);
// sustituye (reemplaza) b caracteres del string s, empezando en la posición
// a por la cadena nuevo_s
frase.replace (1, 6, palabra); // sustituye 6 caracteres de frase, empezando
                               //en la posición 1, por la cadena palabra
```

Buscar una subcadena dentro de una cadena dada

```
s.find(subcadena);
//busca una subcadena dentro de la cadena s devolviendo la posición
i = frase.find(palabra); //busca palabra como una subcadena dentro de frase,
                         // y devuelve la posición donde la encuentra
```

Estructuras o registros

Las estructuras o registros son agrupaciones heterogéneas, contiguas y estáticas. Los elementos contenidos en la agrupación pueden ser de distintos tipos y se suelen llamar 'campos de información', y suelen estar relacionados con la información referente a un objeto concreto, como por ejemplo la información relacionada con una persona (Nombre, edad, D.N.I.,...) o la información relacionada con un libro (Título, autor, precio, disponible/agotado,...)

En C/C++ los registros o estructuras son nuevos tipos de datos que podemos utilizar a lo largo de nuestro programa. Es decir, una vez declarada la estructura podemos utilizar y declarar variables de ese nuevo tipo (igual como si declarásemos variables de tipo entero o real.)

La manera de declarar una estructura es en C/C++ es:

```
struct Nombre
{
   Tipo1 Campo1;
   Tipo2 Campo2;
   ...
};
```

Donde **`Nombre`** es el nombre que va a recibir nuestra estructura, **`Tipo1`** será el tipo de datos del primer campo de información, **`Campo1`** será el nombre del primer campo de información, **`Tipo2`** el tipo de información guardado en el segundo campo de información, **`Campo2`** el nombre del segundo campo de información, y así para todos los campos de información que queramos tener.

Ejemplo: Declarar un registro capaz de guardar la información de una persona

Pondremos como información de una persona: Su nombre, su edad, su D.N.I. y si es Hombre o Mujer.

```
struct persona
{
   string Nombre;
   int Edad;
   long int DNI;
   char Sexo;
};
```

Una vez declarado el nuevo tipo de dato, éste es similar a los tipos de datos simples vistos hasta este momento, y podemos declarar variables de este nuevo tipo. En C:

```
struct NombreEstructura NomVar;
```

Donde '**`struct Nombre`**' es el nuevo tipo y **`NomVar`** el nombre de la variable de este tipo.

Ejemplo

```
struct persona per;
```

En C++, la propia declaración de la estructura ya declara un tipo nuevo con el nuevo nombre, con lo que no es necesario utilizar repetidamente la palabra reservada *struct*.

```
NombreEstructura NomVar;
```

Ejemplo

```
persona per
```

Acceso a los elementos de la estructura

Esta variable aglutina diferentes campos. La manera de acceder a cada uno de los campos es mediante el operador de acceso '**`.`**'

```
NomVar.Campo1 / NomVar.Campo2 / ...
```

El tratamiento de cada uno de estos campos depende de su tipo (si es un entero lo asignaremos y trataremos como entero, si es un vector como un vector, si es un string como un string,...)

Ejemplo: A partir de la estructura persona declarar una variable de este nuevo tipo

```
persona per;
```

Iniciación de una estructura:

Ejemplo: Asignar los valores Nombre=Ricardo Edad=32 DNI=20200200 Sexo=H

```
persona per;

cout << "Dame el nombre: \n";
cin >> per.Nombre;

cout << "Dame la edad: \n";
cin >> per.Edad;

cout << "Dame el D.N.I.: \n";
cin >> per.DNI;

cout << "Dame el sexo: \n";
cin >> per.Sexo;
```

Un caso particular sería asignar los valores al declarar la variable, por ejemplo:

```
struct Fecha
{
    int dia;
    int mes;
    int anyo;
};
Fecha f= {1,1,2001};
```

Pero esto No funciona con componentes string para la estructura.

Anidamiento de estructuras:

Las estructuras también se pueden anidar, es decir, uno de los campos de una estructura puede ser a su vez otra estructura:

```
struct alumno
{
    persona InfoPersonal;
    Fecha FechaNacimiento;
    int curso;
};
```

Asignación de estructuras:

Las estructuras sí se pueden asignar, al igual que los strings o cualquier tipo simple. Una asignación de estructuras es equivalente a una asignación de cada uno de los componentes.

```
Fecha hoy, ayer;
ayer= hoy;
```

Paso de parámetros:

Las estructuras se pasan como parámetros exactamente igual que cualquier otro tipo simple. Las funciones también pueden devolver estructuras:

```
persona QuienEsMayor(persona p1, persona p2)
{
    if (p1.edad >= p2.edad)
        return (p1);
    else
        return p2;
}
```

Introducción a las estructuras dinámicas: Punteros

Existe en casi todos los lenguajes de programación un tipo de datos simple que es el 'puntero'. Un puntero es un tipo de dato que es capaz de guardar una dirección de memoria. Las direcciones de memoria sirven al ordenador para saber en qué lugar de la memoria se sitúa exactamente una determinada información, y en general cualquier variable lleva asociada una dirección de memoria.

Los operadores básicos con punteros son:

- **`&`** Obtiene la dirección de memoria en donde se encuentra una variable, a partir del nombre de la variable.
- **`*`** Obtiene el contenido (valor) que se guarda en una determinada posición de memoria.
- **`malloc`** Reserva espacio en memoria para una determinada información y devuelve la dirección de memoria en donde se encuentra ese espacio.
- **`free`** Libera el espacio reservado para una información (le dice al ordenador que ese espacio puede ser utilizado por otros usuarios.)

Con este tipo de datos y los registros podemos construir agrupaciones de elementos que pueden ir creciendo en memoria a medida que va aumentando al información que deseamos guardar en el

ordenador, de manera que no tenemos que fijar el número máximo que queremos guardar en la agrupación en el momento de escribir el programa.

En general podremos construir agrupaciones dinámicas incluyendo un puntero como campo de un registro, de manera que a través de este puntero puedo acceder a otros elementos de la agrupación.

Una definición de un registro de este tipo, podría ser en C:

```
struct Nodo
{
   Valor Info;
   struct Nodo * Sig;
};
```

A estos registros, que contienen un puntero se les suele llamar NODOS, y son la base de las agrupaciones dinámicas, ya que gracias a estos punteros podemos enlazar todos los elementos de información que queramos (limitados tan solo por la capacidad física de la memoria del ordenador) y además gracias a la instrucción '**malloc**' podemos reservar memoria para nuevos elementos a medida que nos vaya haciendo falta.

La manera de acceder a los campos de un registro a partir de un puntero es mediante el operador 'contenido' (el *****) y el operador de acceso '**.**'

```
struct Nodo * p_aux;

(*p_aux).Info / (*p_aux).Sig;
```

Esta combinación es equivalente al operador de acceso en punteros '**->**' (un menos seguido de un mayor) de manera que quedaría:

```
p_aux->Info / p_aux->Sig
```

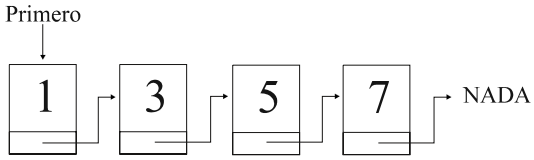
Listas enlazadas

Una lista es una agrupación de elementos entre los que existe uno que podemos llamar 'primero' y a partir del cual se puede acceder al resto de los elementos uno a uno.

Las listas enlazadas están definidas por:

1. Una estructura de tipo nodo.
2. Un puntero que nos marca el primer elemento, a partir del cual puede accederse a cualquier otro elemento de la agrupación.

La idea a la que se quiere llegar sería la siguiente: La lista de números 1, 3, 5, 7 quedaría en forma enlazada como sigue:



Primero sería el puntero que señalaría al primer elemento de la lista. Mediante el puntero situado en cada uno de los nodos es posible acceder al siguiente elemento desde uno cualquiera.

A '**NADA**' en C/C++ se le llama '**NULL**'

Operaciones

Creación

Cuando creamos una lista, en principio estará sin elementos. La creación de una lista llevará asociado, la declaración del tipo Nodo y una variable que sea capaz de guardar dónde está el primero de los elementos, así como la asignación de que no hay ningún elemento en la lista.

```
struct Nodo
{
   Valor Info;
   struct Nodo * Sig;
};
...
struct Nodo * Primero;
...
Primero = NULL;
```

La declaración del tipo Nodo se debe realizar al comenzar el programa, después de los "includes" y los "defines".

La variable de tipo puntero se declara en el lugar adecuado de la declaración de variables. Y la asignación en el punto del programa en el que deseemos iniciar la variable.

Búsqueda de un elemento

En el caso de listas dinámicas, sólo podremos realizar búsqueda secuencial, es decir, partiendo del primer elemento ir mirando si el que estamos buscando es o no el indicado.

```
int Buscar (struct Nodo * primero, int x)
{
   struct Nodo * p_aux;

   p_aux = primero;
   while ( (p_aux != NULL) && (p_aux->Info != x) )
      p_aux = p_aux->Sig

   if (p_aux == NULL)
      return (0);
   else
      return (1);
}
```

Inserción de un elemento delante del primero

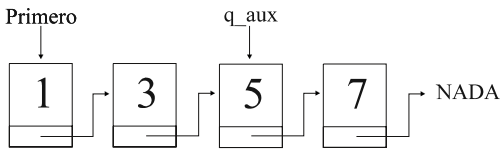
Supongamos que queremos insertar el elemento '**x**' delante del primero:

```
struct Nodo * p_aux;
...
/* Reservamos memoria para el nuevo elemento */
p_aux = malloc (sizeof (struct Nodo) );

/* Actualizamos su informacion */
p_aux->Info = x;

/* Enlazamos el elemento en la lista */
p_aux->Sig = primero;

/* El primero ahora es el que hemos insertado */
primero = p_aux;
```

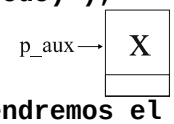
Inserción de un elemento detrás de uno dado

Supongamos que queremos insertar el elemento '**x**' detrás de uno marcado con '**q_aux**':

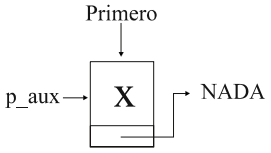
```
void InsertarDetras (struct Nodo * primero, struct Nodo * q_aux, Valor x)
{
   struct Nodo * p_aux;
```



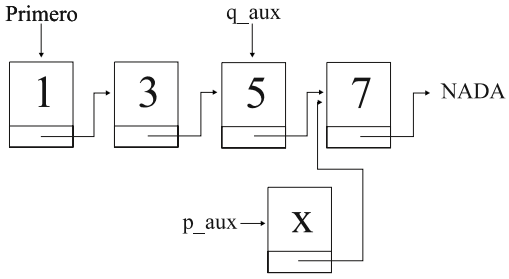

```
   p_aux = malloc (sizeof (struct Nodo) );
   p_aux->Info = x;
```



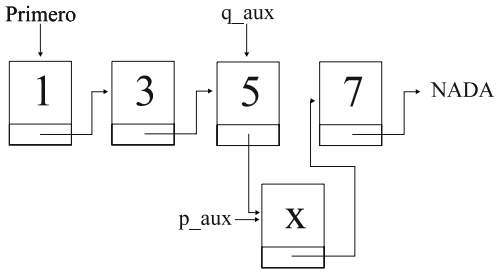

```
   /* Si no habia elementos solo tendremos el que acabamos de crear */
   if (primero == NULL)
   {
      p_aux->Sig = NULL;
      primero = p_aux;
```




```
   }
   else
   {
      p_aux->Sig = q_aux->Sig;
```




```
      q_aux->Sig = p_aux;
```

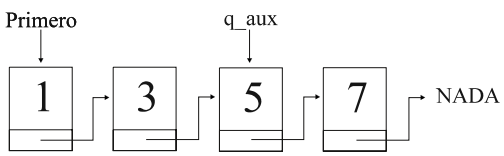



```
   }
}
```

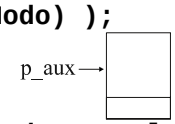
Inserción de un elemento delante de uno dado

Supongamos que queremos insertar el elemento '**x**' delante de uno marcado con '**q_aux**':

```
void InsertarDelante (struct Nodo * primero, struct Nodo * q_aux, Valor x)
{
   struct Nodo * p_aux;
```

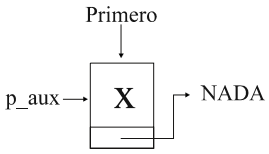


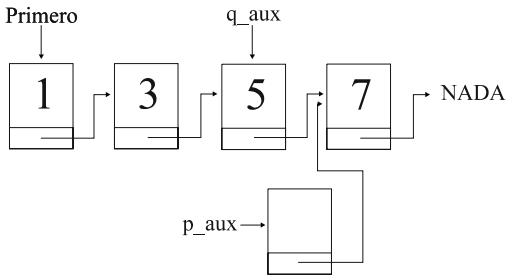

```
   p_aux = malloc (sizeof (struct Nodo) );
```



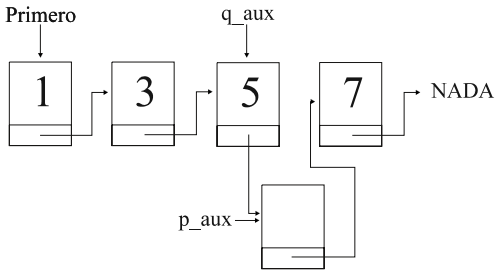

```
   /* Si no habia elementos solo tendremos el que acabamos de crear */
```

```
if (primero == NULL)
{
   p_aux->Info = x
   p_aux->Sig = NULL;
   primero = p_aux;


}
else
{
   p_aux->Sig = q_aux->Sig;
```

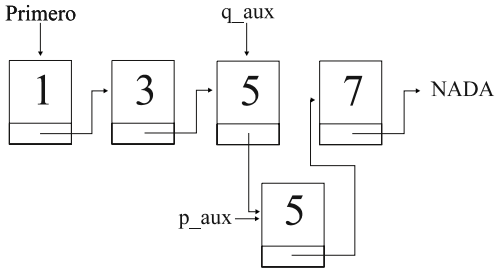




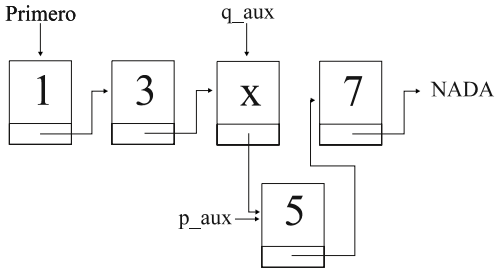

```
   q_aux->Sig = p_aux;
```




```
   p_aux->Info = q_aux->Info;
```




```
   q_aux->Info = x;
```



Ventajas e inconvenientes de las agrupaciones estáticas frente a las agrupaciones dinámicas:

- Mayor facilidad para insertar y eliminar elementos en las agrupaciones dinámicas.
- Más fácil y rápido realizar búsquedas en vectores.
- Si el número de elementos es conocido o no va a variar mucho → agrupaciones estáticas
- Si el número de elementos va a ser cualquiera → agrupaciones dinámicas.
- En general, para un mismo número de elementos las agrupaciones dinámicas ocuparán más memoria que las estáticas (por cada elemento se tiene también un puntero en las agrupaciones dinámicas.)