Tipos de datos y estructuras de datos

- Un *tipo de datos* consiste en una colección de datos junto con unas operaciones básicas y unas relaciones definidas para esos valores.

- Un tipo de datos se llama *simple* si sus datos son atómicos; esto es, consisten en elementos simples que no pueden dividirse.

    - El tipo de dato entero consiste en un subconjunto del conjunto de enteros de las matemáticas, $\{\ldots,-3,-2,-1,0,1,2,3,\ldots\}$, las operaciones aritméticas básicas `+`, `-`, `*`, `/`, `DIV` y `MOD`, y las relaciones `<`, `>`, `=`, `<>`, `<=` y `>=`.

    - El tipo de dato `real` consiste en un subconjunto del conjunto de números reales junto con las operaciones aritméticas y relaciones usuales.

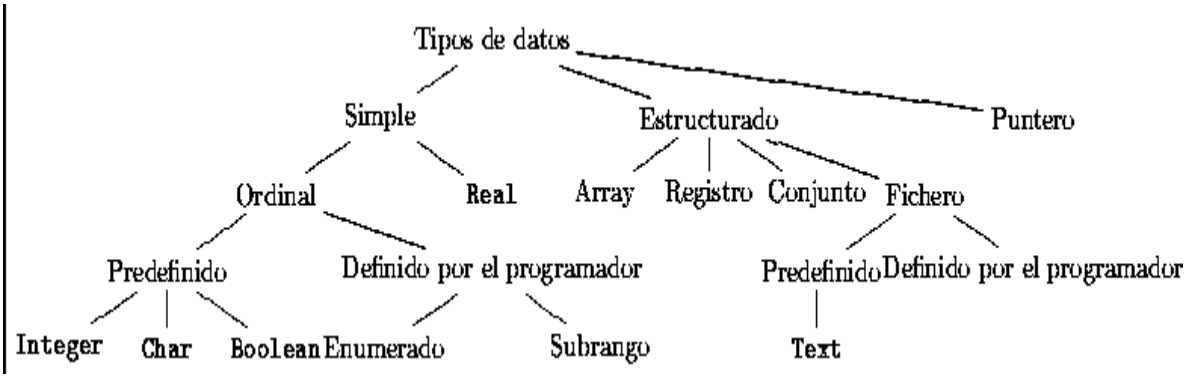
    - El tipo de dato `boolean` consta de dos valores, `true` y `false`, y las operaciones `NOT`, `AND` y `OR`.

    - El tipo de dato `char` consiste en el conjunto de caracteres del código ASCII, con las operaciones básicas para comparar los caracteres.

- Un tipo de datos se dice *estructurado* cuando sus datos son colecciones de elementos. El tipo `string, record o struct` es un tipo de dato estructurado.

- Un tipo de *datos apuntador o también conocido como apuntador* se utiliza para valores de datos que son direcciones de memoria.

- Un tipo de datos simple es de tipo *ordinal* si los valores del tipo están ordenados de manera que cada uno de ellos, excepto el primero tiene un predecesor inmediato y cada uno de ellos, excepto el último, tiene un sucesor inmediato.



Los tipos ordinales

- Un tipo de datos simple es *ordinal* si sus valores están ordenados de manera que cada uno de ellos, excepto el primero tiene un predecesor inmediato y cada uno de ellos, excepto el último, tiene un sucesor inmediato.

    - los tipos `enteros`, `lógicos` y `caracter` son tipos ordinales.

    - el tipo `real o flotante` no es un tipo ordinal.

- Las variable de los tipos ordinales pueden utilizarse como variables de control en instrucciones `FOR,WHILE o DO-WHILE`.

- Los tipos de datos ordinales simples definidos por el programador pueden dividirse en *enumerados* y *subrangos*.

C y C++ utiliza 5 palabras reservadas para definir los tipos de datos fundamentales. A diferencia de Pascal, un determinado tipo de datos puede ir cualificado por un conjunto de modificadores.

Los tipos de datos fundamentales son:

`Char`	`short int`	`int`
`long int`	`unsigned char`	`unsigned short int`

`unsigned int`	`unsigned long int`	`double`
`Float`	`long float`	`void`

Tipos de datos fundamentales

Char	Representa un carácter en código ASCII, también se puede interpretar como un entero.
short int	Indica un entero de tamaño corto.
Int	Entero igual que integer en Pascal.
long int	Entero largo.
unsigned short int	Como short int pero sin signo.
unsigned int	Como int pero sin signo.
unsigned long int	Como long int pero sin signo.
Flota	Flotante corto. Análogo al single de Pascal.
Double	Flotante largo. Análogo al double de Pascal.
Void	No indica ningún tipo. Es el tipo de las funciones que no devuelven nada.

Los tipos short int, long int, unsigned int y long float se pueden escribir como: short, long, unsigned y double.

Con respecto al tamaño que ocupan en memoria variables de estos tipos, todo lo que garantiza C y C++ es:

sizeof(char) = 1
sizeof(short) <= sizeof(int) <= sizeof(long)
sizeof(unsigned) = sizeof(int)
sizeof(float) =< sizeof(double)

Donde sizeof es un operador que está incorporado en C y C++ y devuelve el número de bytes que tiene un objeto.

Hay un grupo de modificadores que indican la forma que se almacena una determinada variable de un determinado tipo. Se indica antes del tipo de la variable.

static
Cuando se invoca a una función por segunda vez se pierden los valores que las variables locales de la función tenían al acabar la anterior llamada. Declarando una variable de este tipo cuando se llama por segunda vez a la subrutina la variable static (estática) contiene el mismo valor que al acabar la llamada anterior.

auto
Es lo mismo que si no se usara ningún modificador.

volatile
El compilador debe asumir que la variable está relacionada con un dispositivo y que puede cambiar de valor en cualquier momento.

register
El compilador procurará almacenar la variable cualificada de este modo en un registro de la CPU.

extern
La variable se considera declarada en otro archivo. No se le asignará dirección ni espacio de memoria.

Estructuras de datos

struct Es análoga al constructor record de Pascal. Su sintaxis es:

```
struct nombre {
 tipo miembro1;
 tipo miembro2;
```

...
} variable1, variable2, ...;

Nombre es el nombre de la estructura y los identificadores corresponden a variables del tipo de la estructura. Tanto el nombre como las variables pueden no existir. Los miembros son las componentes de la estructura. La forma de declarar variables del tipo definido en la estructura es:

[cualificador] struct nombre variable1, identificador2, ...;

Para acceder a los campos de la estructura se usa la misma técnica que en Pascal: variable.campo
Se admite el uso de estructuras dentro de la declaración de otra estructura ya que los miembros, en general, pueden tener cualquier tipo.

Ejemplo de struct

```
struct fecha {
  int mes;
  int día;
  int año;
};      /* Estructura fecha */
struct cuenta {
  int cuen_no;
  char cuen_tipo;
  float saldo;
  struct fecha ultimo_pago;
} cliente[100];   /* Estructura cuenta y declaración */


struct cuenta clientes[20];
clientes[0].cuen_no = cliente[99].ultimo_pago.anio;
```

union Es una construcción análoga al registro con variante de Pascal. Al igual que las estructuras constan de varios miembros aunque éstos comparten la misma ubicación de memoria. Se usa para hacer referencia a una misma área de memoria de varias formas. El

tamaño de la unión es el tamaño del miembro mayor.
La forma de declarar una unión es:

Ejemplo de unión

```
modificador union nombre {
 tipo miembro1;
 tipo miembro2;
 ...
} identificador1, identificador2, ...;
```

La forma de declarar variables del tipo unión es:

```
union nombre variable1, variable2, ...;
```

```
union palabra {           Se declaran 4 variables
  unsigned char l[2];     a,b,c y d
  unsigned int x;         A cada variable se puede
} a,b,c,d;                acceder como entero (x) o
                          como 2 bytes (l[2]).
```

```
a.x = 65535U;   a.x es un entero
a.l[0] = 0;    a.l[0] es un byte
a.l[1] = 255;    a.l[1] es un byte
 ¿Cuánto vale a.x en este punto?
```

typedef Permite definir un nuevo tipo de datos en función de los ya existentes. Es análogo a las declaraciones type de Pascal. La forma general de definir un nuevo tipo es:

```
typedef tipo nuevo-tipo;
```

nuevo-tipo es el nombre del tipo que se va a definir y tipo puede ser todo lo complicado que se quiera siempre que esté definido con los tipos ya existentes.

Ejemplos de typedef

```
typedef struct {
  int mes;
  int día;
  int anio;
} fecha;    /* Declaración de un nuevo tipo llamado fecha */
fecha hoy;  /* Declaración de una variable de tipo fecha */
typedef unsigned char byte;  /* Tipo byte de Pascal */
typedef unsigned int word;  /* Tipo word de Pascal */
```

enum Es análogo a un tipo enumerado de Pascal. Sirve para especificar los miembros de un determinado tipo.

Ejemplo de enum

enum nombre { miembro0 = valor0 , miembro1 = valor1 , ... }
variable1, variable2, ...

Las expresiones = valorX son opcionales, encargándose el compilador de asignar un valor constante a cada miembro que no la posea.

```
enum colores {negro =-1,azul, cian, magenta, rojo = 2,
blanco} fondo, borde;
enum colores primer_plano;
fondo = cian;
```

Declaración de variables

La forma general de declarar variables en C es la siguiente:

cualificador tipo identificador = valor, identificador = valor, ... ;

Las expresiones = valor sirven para inicializar la variable y pueden ser opcionales.
Las variables pueden ser declaradas en dos puntos: dentro de un bloque antes de la primera línea ejecutable; en este caso el ámbito de la variable es el cuerpo del bloque y fuera de todos los procedimientos, en este caso, el ámbito abarca a todas las funciones, es decir son declaraciones globales. El cuerpo de una función es considerado como un bloque.

Ejemplos de declaración de variables.

```
int a,b,c;                  Tres variables enteras.
float raiz1, raiz2;         Dos variables de tipo real.
char caracter, texto[80];   Un caracter y una cadena de 80.
short int a;                Entero corto.
long int b;                 Entero largo.
unsigned short int d;       Entero corto sin signo.
unsigned char a;            Caracter sin signo.
signed char b;              Caracter con signo.
char texto[3] = "abc";      Declaración e inicialización
char a = '\n';              Inicialización con Return.
char texto[] = "abc";       Sin especificar tamaño.
extern unsigned short int   Variable externa.
```

Arrays

En C y C++ los arrays son de tamaño fijo, es decir, su tamaño se especifica en tiempo de compilación.

No se comprueba la longitud del array a la hora de acceder. Es responsabilidad del programador controlar los accesos fuera de rango.

Si se pretende escribir en la posición 20 de un array y este ha sido declarado de 10 componentes, no se generará ningún error (ni siquiera en tiempo de ejecución) y se escribirá en memoria en la dirección donde debería ir el elemento 20 si lo hubiera, por lo que se puede variar el contenido de otra variable, o incluso de parte del programa, o peor aún, se puede destruir parte del sistema operativo.

La forma de declarar arrays en C y C++ es la siguiente:

cualificador tipo variable[expresión][expresión]... ={valor, ...};

La expresión y los valores deben ser constantes. Si se declara la parte de valores se puede eliminar la expresión. En este caso el tamaño del vector es el mismo que el número de valores que se especifican. No se admite el cualificador register.
La forma de hacer referencia a un elemento del vector es:

nombre[expresión][expresión]...

Las cadenas son consideradas como vectores de caracteres. Acaban con el carácter nulo '\x0'. Pueden tener una longitud arbitraria y el primer byte corresponde al primer caracter. No se almacena la longitud de la cadena como parte de la misma (como ocurre en Turbo Pascal).

Constantes de caracteres

Es un sólo carácter o una secuencia de escape encerrado con comillas simples.
Las secuencias de escape son las que se presentan a continuación:

Secuencias de escape

```
sonido (campana)    '\a'
backspace           '\b'
tab horizontal      '\t'
tab vertical        '\v'
nueva línea         '\n'
form feed           '\f'
retorno de carro    '\r'
comillas(")         '\"'
comilla simple(')   '\''
signo interrogación '\?'
backslash (\)       '\\'
```

```
nulo                '\0'
```

Mediante secuencias de escape se puede expresar cualquier carácter ASCII indicando su código en octal (\ooo) o en hexa (\xhh). Donde los símbolos 'o' representan dígitos octales y las 'h' dígitos hexadecimales.

Expresiones

El lenguaje C y C++ permite operar dos expresiones de cualquier tipo con cualquier operador. Es responsabilidad del programador conocer la interpretación que hará el compilador.
En operaciones lógicas y comparaciones el valor falso se representa con un cero y el verdadero es cualquier otro número, siendo 1 el que devuelven dichas operaciones.

Operaciones aritméticas

+	Suma
-	Resta
*	Multiplicación
/	División
++	Incremento
--	Decremento
%	Módulo

Operaciones lógicas

|| OR lógico
&& AND lógico
! NOT lógico
^ XOR lógico

Operaciones de bits

| OR
& AND
~ NOT
>> Desplazamiento a la derecha
<< Desplazamiento a la izquierda

Comparaciones

< Mayor que
> Menor que
<= Menor o igual que
>= Mayor o igual que
== Igual que
!= Distinto que

Asignaciones

= Asignación
+= ejemplo: `a = a + 3` es equivalente a: `a += 3`
-= ejemplo: `a = a - 3` es equivalente a: `a -= 3`
*= ejemplo: `a = a * 3` es equivalente a: `a *= 3`
/= ejemplo: `a = a / 3` es equivalente a: `a /= 3`
%= ejemplo: `a = a % 3` es equivalente a: `a %= 3`
^= ejemplo: `a = a ^ 3` es equivalente a: `a ^= 3`
&= ejemplo: `a = a & 3` es equivalente a: `a &= 3`
|= ejemplo: `a = a | 3` es equivalente a: `a |= 3`
>>= ejemplo: `a = a >> 3` es equivalente a: `a >>= 3`
<<= ejemplo: `a = a << 3` es equivalente a: `a <<= 3`

Operadores especiales

`? :`	Operador If de expresiones
`,`	Separador de expresiones
`*`	Valor apuntado por un apuntador (indirección)
`&`	Dirección de una variable
`(type)`	Typecasting, donde type indica el tipo de datos al que se convierte el resultado de la expresión.
`sizeof type`	Devuelve el número de bytes necesarios para representar un determinado tipo de datos.
`.`	Cualificador. Separador de campos
`->`	`(p*).a` es equivalente a: `p->a`

Precedencia de los operadores

```
() [] -> .
! ~ ++ -- - (type) * & sizeof       (todos son unarios)
* / %
+ -
<< >>
< <= > >=
== !=
&
^
|
&&
||
?:
= += -= *= /= %= ^= &= |=
,
```

Incremento ++ y decremento - -

Existen dos formas de incrementar y decrementar variables: postincremento y postdecremento, y preincremento y predecremento. En los primeros se calcula la expresión con el valor de la variable y luego se incrementa o decrementa dicha variable, con los segundos se incrementa o decrementa y luego se calcula la expresión.

Detalle sobre los operadores de asignación

Las asignaciones son operadores, es decir devuelven un valor. El valor que devuelven es el correspondiente al operando de la parte derecha.

Ejemplos de asignaciones

```
int a, b, c;
a = b = c = 0;
a += b = c;
```

Esta propiedad permite que se realicen expresiones como el condicional:

```
if ((x = f(y) == N) ...
```

Preferiremos habitualmente el código equivalente:

```
x = f(y);
if (x == N) ...
```

Sentencias

En los lenguajes C y C++ hay tres tipos de sentencias: las de expresión, las compuestas y las de control de flujo.
Las sentencias de expresión son aquellas en las que se especifica una expresión para evaluar y que devuelven un valor. Acaban siempre con el símbolo ";".

Las sentencias compuestas son las que comienzan por llaves abiertas y acaban con llaves cerradas, conteniendo en su interior en primer lugar las posibles declaraciones (locales al ámbito de la sentencia) y a continuación una lista de sentencias.

Las sentencias de control de flujo son las de bucle (**for**, **while**, **switch** y **do**), la condicional (**if**) y las de salto (**goto**, **continue** y **break**).

Sentencia while

while (expresión) sentencia

Se repite la sentencia mientras el valor de expresión sea cierto (no 0). La condición se evalúa antes de ejecutar la sentencia.

Ejemplo de sentencia while

```
/* Cálculo de la media de un vector */
int v[100], i = 0, media, suma = 0;
while (i < 100)
  suma += v[i++];
media = suma / 100;
```

Sentencia for

for (expresión1; expresión2; expresión3) sentencia

Es la sentencia de control más potente y la más usada. Consta de tres expresiones: la primera es la inicialización del bucle, la segunda indica la condición en la que se debe continuar el bucle y la tercera es la que se encarga de incrementar los índices del bucle. Expresión1 se ejecuta una sola vez al principio del bucle. La sentencia se ejecuta mientras la expresión2 sea verdadera (no 0). Esta expresión es evaluada antes que la sentencia por lo que es posible que el bucle no se ejecute ni siquiera una vez. La expresión3 se ejecuta después de la sentencia. Las expresiones 1 y 3 pueden ser compuestas, expresiones simples separadas por comas. La instrucción for equivale directamente a lo siguiente.

```
expresion1;
while (expresion2) {
 sentencia;
 expresion3;
}
```

```
/* Cálculo de la media de un vector */
int v[100], i, media, suma = 0;
for (i = 0; i < 100; i++)
  suma += v[i];
media = suma / 100;
```

Figura 19 Ejemplo de bucle for

Sentencia if

if (expresión) sentencia1 else sentencia2

Igual que en Pascal la parte else es opcional. La sentencia1 se ejecuta si expresión tiene un valor verdadero (distinto de 0) y si tiene un valor falso (0) se ejecuta la sentencia2.

Ejemplo de sentencia if-else

```
if (estado == 'S')
   tasa = 0.20 * pago;
else
  tasa = 0.14 * pago;
```

```
/* Nótese que en C++ se escribe ";" antes del else */
```

Sentencia do-while

do sentencia while (expresión);

Se repite la sentencia mientras expresión sea cierta (no 0). Es análoga al repeat de Pascal. La expresión se evalúa después de ejecutar la sentencia, por lo que esta se ejecuta al menos una vez.

Ejemplo do-while

```
/* Cálculo de la media de un vector */
int v[100], i, media, suma = 0;
i = 0;
do
  suma += v[i++];
while (i < 100);
media = suma / 100;
```

Sentencia switch

switch (expresión) {
 case expresión1: sentencia; sentencia; ...
 case expresión2: sentencia; sentencia; ...
 case expresión3: sentencia; sentencia; ...
 default: sentencia; sentencia; ...
}

Es análoga a la sentencia case de Pascal. La expresión se evalúa y si su valor coincide con el valor de alguna expresión indicada en los case se ejecutan todas las acciones asociadas que le siguen. Las expresiones deben ser de tipo entero o carácter. Si el valor de expresión no se encuentra en la lista case se ejecuta/n la/s sentencia/s correspondiente/s a la opción default, si ésta no existe se continúa con la sentencia situada a continuación de switch. Una vez se elija una opción se ejecutan las sentencias asociadas y se continúan ejecutando todas las sentencia a partir de ésta (incluso las correspondientes a otras opciones) hasta que aparezca una sentencia break.

Sentencia break

break

Se usa para romper una sentencia **while**, **do-while**, **for** o **switch**. Si se ejecuta se sale del bucle más interno o de la sentencia **switch** que se esté ejecutando.

Ejemplo de break

```
char color;
switch (color) {
  case 'a':
  case 'A': cout<<"AMARILLO\n";
            break;
  case 'r':
  case 'R': cout<<"ROJO\n";
  case 'b':
  case 'B': cout<<"BLANCO\n";
  default: cout<<"OTRO\n";
}
```

Sentencia continue

continue

Esta sentencia anula la pasada actual de un bucle. Se ejecuta la siguiente pasada del bucle más interno en el que se encuentre ignorándose el código que se tendría que ejecutar para acabar la pasada actual. Se puede usar en los bucles **for**, **while** y **do-while**

Funciones

El lenguaje C++ sólo permite funciones, no hay procedimientos. La forma de declarar las funciones es la siguiente:

tipo nombre(tipo parámetro_1, ... tipo parámetro_N) sentencia

Donde tipo es el tipo del dato que devuelve la función, que si no aparece se supone siempre int, y nombre es el identificador de la función.

La forma de invocar a una función es la misma que en Pascal, pero aunque la función no tenga parámetros se deben colocar los paréntesis.

Las listas de parámetros formales y de parámetros actuales no tienen por qué coincidir en número, e incluso en tipo. Si hay más parámetros formales que actuales se les asigna un valor indefinido a los formales; si ocurre al revés se descartan los parámetros actuales que sobren. Para evitar este problema se puede declarar el prototipo de la función antes de invocarla. La forma de escribir el prototipo de una función es la siguiente:

tipo nombre(tipo parámetro_1, ... tipo parámetro_N);

O sea, la cabecera de la función acabada en punto y coma. Es análoga una declaración.
Como se comentó anteriormente sólo existe paso por valor.
Para terminar la ejecución de una función se usa la sentencia return. Su sintaxis es:

return [expresión];

Donde la expresión es el valor que retorna la función. Debe ser del mismo tipo que el de la función.

```
void funcion1(int a); /* Prototipo de la función */
void funcion2(void); /* Prototipo de la función */


/* trozo de programa */


void funcion1(int a) { /* Función 1 */
  /* Cuerpo de la función */
  funcion2();
}
```

```
  void funcion2(void) { /* Función 2 */
    /* Cuerpo de la función */
    funcion1(2);
  }
```
Figura 23 Funciones

Librerías

En el lenguaje C toda la entrada y salida de un programa se realiza a través de funciones definidas en librerías. También se encuentran definidas en librerías otros tipos de funciones. Dichas librerías, o la mayoría, son estándar (las funciones en ellas definidas tienen nombres estándar) lo que facilita la portabilidad de los programas. Al inicio de cada archivo se debe indicar las declaraciones de las librerías que se utilizarán. Esto se realiza utilizando la directiva #include, cuya sintaxis es:

#include nombre_archivo

donde el nombre del archivo donde se realizan las definiciones se coloca entre ángulos (<nombre_archivo>) o entre comillas dobles ("nombre_archivo") según el lugar en que haya que buscar el archivo sea en los directorios asignados por defecto a los archivos 'include' o en el actual.

Estos archivos de definiciones suelen tener la extensión .h (de header, cabeceras) y contienen definiciones necesarias para la utilización de las funciones contenidas en las librerías.
Por otra parte, utilizando la directiva #include se puede hacer inclusión de archivos de código del mismo modo que en cualquier otro lenguaje.

Ejemplo de inclusión de librerías

```
#include <conio.h>   Incluye la cabecera para usar las
librerias
                     de entrada/salida desde el
                     directorio estándar
```

```
#include "conio.h"   Igual al anterior pero las busca
                     en el directorio actual


#include "a:\librería\mia.h"     Incluye el archivo mia.h
                                 desde la unidad a:
```

Apuntadores

Los apuntadores son una de las características más útiles y a la vez más peligrosas de que dispone el lenguaje C++. En C++ se permite declarar una variable que contiene la dirección de otra variable, o sea, un apuntador. Cuando se declara un apuntador éste contiene una dirección arbitraria, si leemos a dónde apunta nos dará un valor indefinido y si se escribe en tal dirección estamos variando el contenido de una posición de memoria que no conocemos por lo que podemos hacer que el sistema tenga comportamientos no deseados.

Antes de hacer uso de un apuntador debemos asignarle una dirección de memoria en nuestro espacio de trabajo.

Declaración de apuntadores

La forma de declarar un apuntador es la siguiente:

cualificador tipo *nombre, *nombre;

El tipo indica al tipo de datos a los que apuntará el apuntador, pero como efecto de la declaración se reservará espacio en memoria para guardar un apuntador, no para el tipo de datos al que apunta.

Existe un carácter especial que se usa como prefijo y aplicado a las variables indica la dirección de memoria que ocupa la variable, no el contenido (valor). Este símbolo es &. Además existe otro prefijo, *, que aplicado a una variable de tipo apuntador indica el contenido de la dirección a la que apunta dicho apuntador. A estos dos símbolos se les llama dirección e indirección respectivamente.

Hay 3 formas de inicializar un apuntador:

a) Inicializarlo con el valor NULL (definido en un archivo header). De este modo estamos indicando que el apuntador no apunta a ninguna memoria concreta.
b) Inicializarlo haciendo que tome como valor la dirección de una variable.

```
int *p, a;
p = &a;
```

A partir de estas sentencias, *p y a son alias.
c) Asignarle memoria dinámica a través de una función de asignación de memoria. Las funciones más habituales son calloc y malloc, definidas en el archivo alloc.h o bien en stdlib.h

```
void *malloc(size_t size)
void *calloc(size_t n_items, size)
```

Una inicialización de un apuntador a un tipo T tendría la forma:

```
p = (T*)malloc(sizeof(T));
```

Apuntadores a una estructura

Existe una pequeña variación a la hora de acceder a una estructura mediante un apuntador, por ejemplo, (*p).miembro se puede escribir como p->miembro.

Paso de apuntadores a una función

El lenguaje C sólo admite paso de parámetros por valor, pero tiene una forma de simular un paso por referencia (variable), pasando un apuntador que es la dirección donde están los datos (p. ej. &v). En realidad se pasa un valor que es una dirección de una variable.

Ejemplo de paso de apuntadores a una función

```
void func(int *pa,int b) {
  *pa = 1;
  b = 2;
   return ;
}


main(void) {
  int a, b;
  a = b = 0;
  func(&a, b);
  /* En este punto ¿cuánto valen a y b? */
  return;
}
```

Apuntadores y arrays unidimensionales

El identificador de un array se considera un apuntador al primer elemento del array. Cualquier forma de acceder como un array puede ser sustituida por su forma equivalente como apuntador.

Ejemplos de apuntadores y arrays unidimensionales

```
int x[100];        Declaración de un array de 100 enteros
x[0]     *x        Primer elemento del array
x[2]  *(x + 2)     Tercer elemento del array
x     x            Dirección del array
&x[3] (x + 3)      Dirección del tercer elemento del array


char a[] ="Pulsa Return";


a[0]  *a           Carácter "P"
a[i]  *(a + i)     Carácter i-ésimo
```

```
&a[0]  a           Dirección de la cadena
```

Las strings (cadenas de caracteres) se consideran arrays de caracteres a todos los efectos.

Operaciones con apuntadores

A los apuntadores se les puede añadir o restar una cierta cantidad entera. Admiten comparaciones e incrementos y decrementos. Cuando un apuntador es incrementado en uno pasa a apuntar al siguiente elemento del array a que apuntaba, no al siguiente byte, es decir, se incrementa en el número de bytes que ocupa el tipo al que apunta. También se permite restar dos apuntadores para calcular la distancia entre ellos.

```
int *px, *py;
px < py
px <= py
px > py
px >= py
px == py
px != py
px == NULL
a = *(px++)           Postincremento del apuntador
a = *(++px)           Preincremento del apuntador
px - py               “Distancia” entre los apuntadores px y
py
```

Figura 15 Operaciones con apuntadores

Apuntadores y arrays multidimensionales

Una matriz bidimensional es implementada en C++ como un vector cuyos elementos son vectores. Es decir, la matriz se implementa en forma de vector, sin embargo, cuando accedemos a ella sigue teniendo la forma de matriz.

La forma de declarar un array multidimensional es:

tipo nombre[expresion1][expresion2]...;

La forma de acceder a un elemento de la matriz es:

```
nombre[expresion1][expresion2]...
```

También se puede hacer referencia a los elementos de una matriz usando la técnica de apuntadores pero el texto se hace demasiado críptico.

Arrays de apuntadores

Al igual que de cualquier otro tipo de dato se permite declarar un array de apuntadores. La forma de hacerlo es:

```
tipo *nombre[expresion1][expresion2];
```