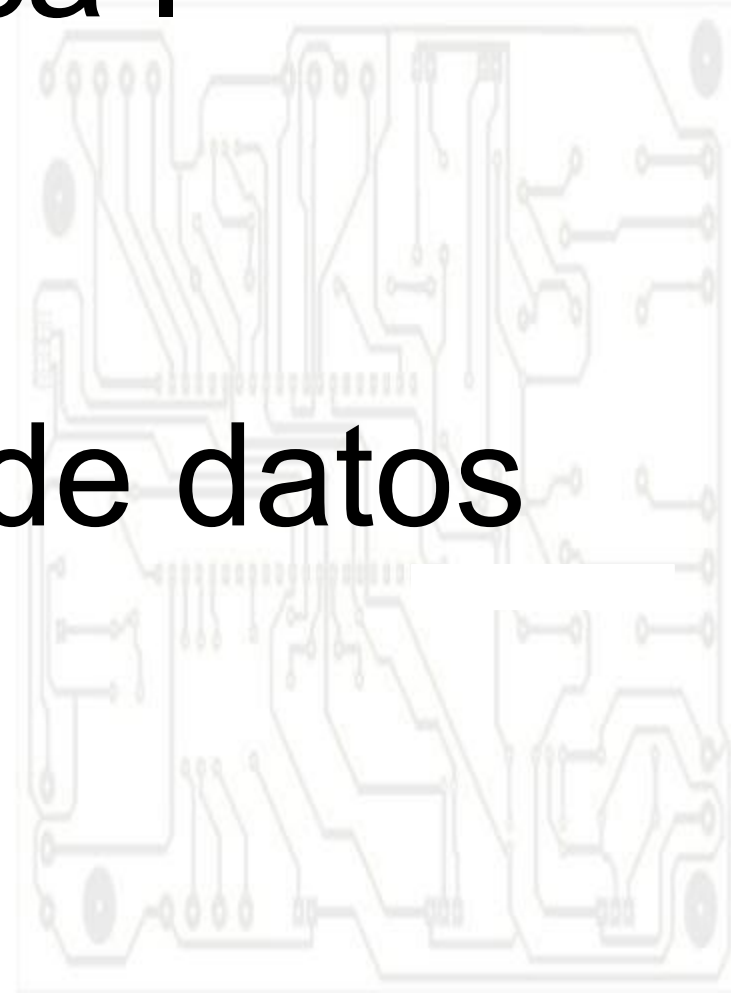
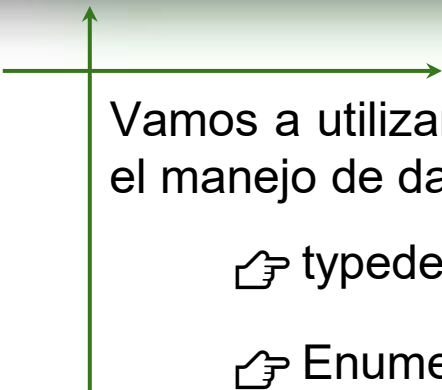


Informática I

R1042

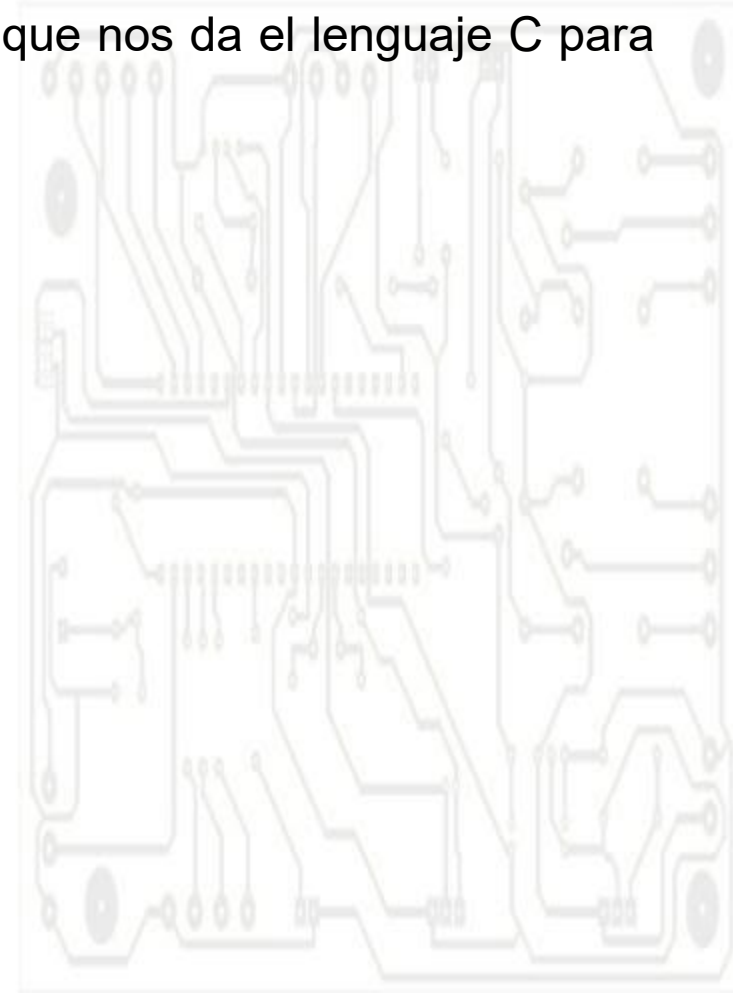
Nuevos tipos de datos





Vamos a utilizar las siguientes herramientas que nos da el lenguaje C para el manejo de datos:

- 👉 typedef
- 👉 Enumeración
- 👉 Estructuras
- 👉 Uniones



Operador typedef

typedef es un operador que me permite agregar un nombre alternativo o alias a un tipo de datos ya existente.

Muchas veces eso simplifica la definición de una variable o le asigna a dicho tipo un nombre más representativo.

Un ejemplo que conocemos es el **size_t** que utiliza el **malloc**.

En mi máquina su definición está en el archivo

`/usr/lib/gcc/x86_64-linux-gnu/5/include/stddef.h`

y dice algo así como

```
#ifndef __SIZE_TYPE__
#define __SIZE_TYPE__ long unsigned int
#endif
#if !(defined (__GNUG__) && defined (size_t))
typedef __SIZE_TYPE__ size_t;
```

typedef - Sintaxis

4

La sintaxis para su uso es:

```
typedef tipo_existente nuevo_tipo;
```

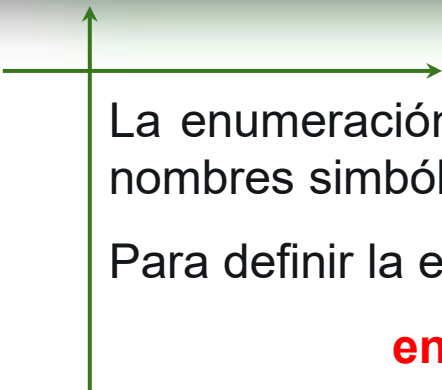
tipo_existente: Es el tipo de datos existente que quiero duplicar.

nuevo_tipo: Es el otro nombre o alias que le quiero asignar al tipo de datos existente.

No olvidarse de poner el ; al final

Como vemos en el caso de **size_t** es conveniente incluir los **typedef** en los archivos de **encabezado**.
Algunos ejemplos clásicos son:

```
typedef char int8;  
typedef unsigned int uint;
```



La enumeración es un recurso que me brinda el lenguaje para otorgarle nombres simbólicos a los valores de una **variable entera**.

Para definir la enumeración puedo hacer:

```
enum nombre {VALOR1, VALOR2, VALOR3};
```

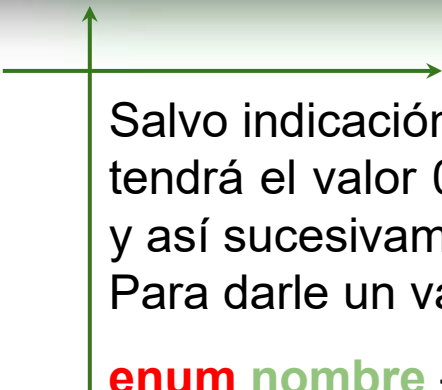
donde

enum: es la palabra reservada del lenguaje para definir la enumeración.

nombre: es el nombre que tendrá esta enumeración.

{VALOR1, VALOR2, VALOR3}: son los valores simbólicos asignados.

No olvidarse de poner el ; al final



Salvo indicación en contrario, el primer valor (**VALOR1** en el ejemplo), tendrá el valor 0, el siguiente tendrá el valor del primer valor más uno y así sucesivamente.

Para darle un valor particular lo puedo indicar como:

```
enum nombre {VALOR1 = 25, VALOR2, VALOR3};
```

entonces **VALOR2** tomará el valor 26.

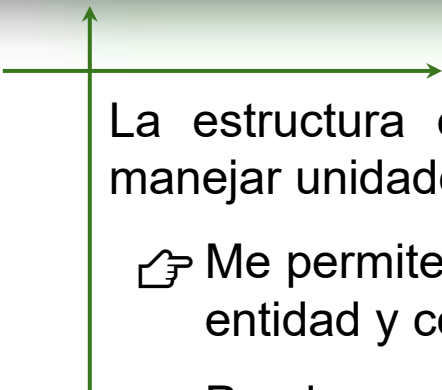
También puedo:

```
enum nombre {VALOR1, VALOR2 = 25, VALOR3};
```

entonces **VALOR1** tendrá el valor 0, **VALOR2** el valor 25 y el **VALOR3** el valor 26.

Para utilizarla simplemente declaro una variable como:

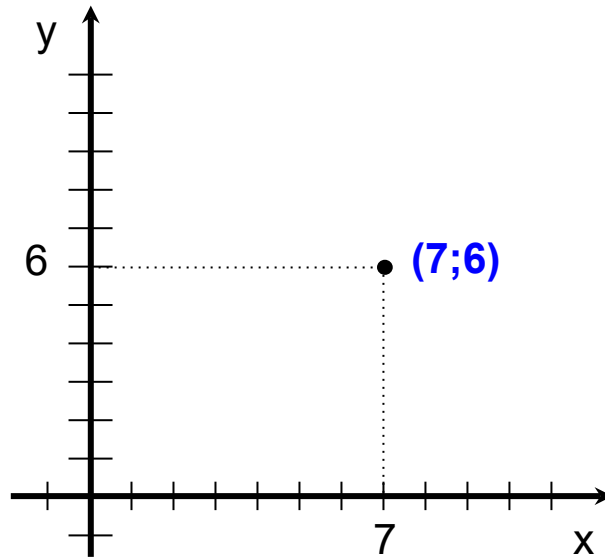
```
enum nombre variable;
```



La estructura es una funcionalidad que me brinda el lenguaje para manejar unidades de información más complejas.

- ☞ Me permite agrupar distintos tipos de datos ya existentes en una sola entidad y con un solo nombre.
- ☞ Pueden contener variables de muchos tipos diferentes de datos.
- ☞ La definición de una estructura crea un nuevo tipo de dato y a partir de ello, puedo definir variables de este nuevo tipo.
- ☞ Tiene el mismo comportamiento que otro tipo de variable, incluso puedo definir vectores de este tipo de datos que acabo de crear.
- ☞ Adicionalmente, se podrá acceder también a cada dato por separado.
- ☞ No se pueden comparar (con por ejemplo los operadores **==** y **!=**)

Veamos un ejemplo sencillo:



Definimos una estructura para hacer más fácil su utilización:

```
struct punto  
{  
    int x;  
    int y;  
}
```


Declarar una estructura significa simplemente informar al compilador que en el programa se utilizará un tipo de dato complejo compuesto por los tipos que se declaran entre las llaves.

Para declarar una estructura entonces:

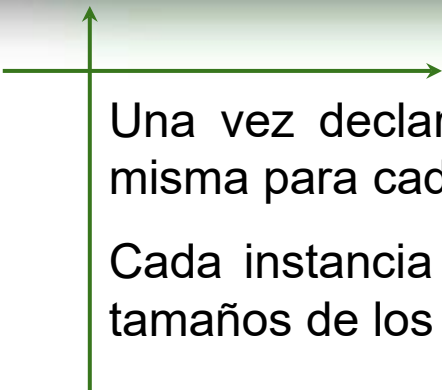
Por ejemplo:

```
struct nombre  
{  
    tipo_dato1  nombre1;  
    tipo_dato2  nombre2;  
    tipo_dato3  nombre3;  
    ...  
};
```

No olvidarse de poner el ; al final

```
struct de_todo  
{  
    int entero;  
    char caracter;  
    float real;  
};
```

Las variables definidas en la estructura se denominan **miembros**.



Una vez declarada la estructura, se debe generar una instancia de la misma para cada variable de ese tipo que debe utilizarse.

Cada instancia ocupa una cantidad de memoria igual a la suma de los tamaños de los miembros de la estructura.

Instanciar = Reservar memoria

Recordar:

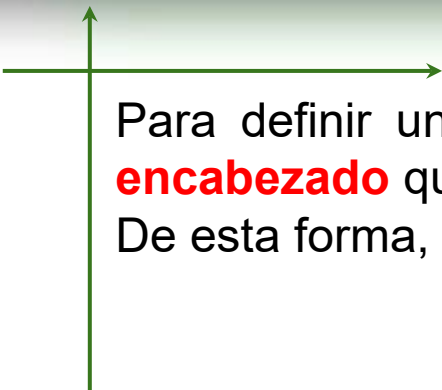
- 👉 Cuando declaro una estructura, esta no ocupa espacio en memoria.
- 👉 Recién cuando la instancio, es decir que defino una variable, me reserva la memoria necesaria para esa variable.

Y para declarar una variable:

```
struct nombre
{
    tipo_dato1  nombre1;
    tipo_dato2  nombre2;
    tipo_dato3  nombre3;
    ...
};
```

```
struct nombre var1;
```

```
struct nombre
{
    tipo_dato1  nombre1;
    tipo_dato2  nombre2;
    tipo_dato3  nombre3;
    ...
} var1, var2;
```



Para definir una estructura lo más conveniente es agregarla en el **encabezado** que declare las funciones que la utilicen.
De esta forma, al incluir el prototipo, también incluyo la estructura.

Es importante el lugar donde está definida porque el tipo de variable tiene existencia a partir de ese momento.

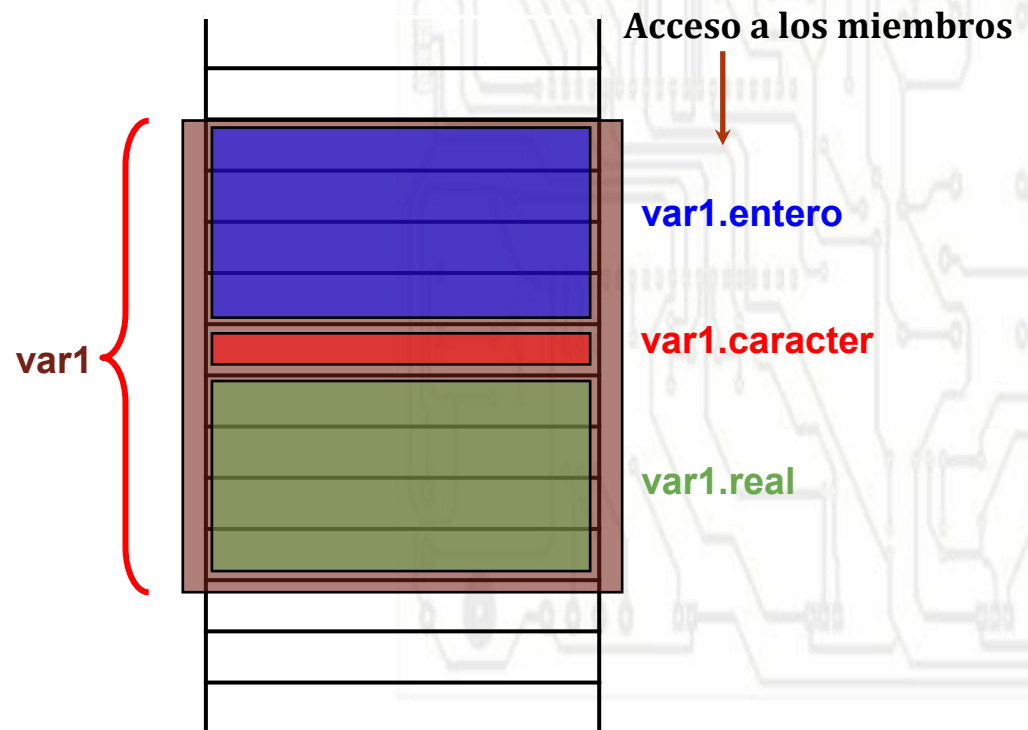
Estructuras - Acceso a los miembros

Para acceder a los miembros se utiliza el operador “.”

La sintaxis establece el formato **nombre.miembro**, donde **nombre** es el nombre de la variable que se le asignó al tipo cuando se definió la estructura, y **miembro**, es el nombre del tipo de dato interno de la estructura al que se quiere acceder.

```
struct de_todo
{
    int entero;
    char caracter;
    float real;
};

struct de_todo var1;
```



Suponiendo la siguiente declaración:

```
struct de_todo
{
    int entero;
    char character;
    float real;
};

struct de_todo var1, var2;
```

Puedo cargar datos de las siguientes formas:

```
var1.entero = 123;
var1.character = 'C';
var1.real = 456.789;
```

```
var2 = var1;
```

```
struct de_todo var3 = { 123, 'C',
456.789};
```

También puedo anidar estructuras:

```
struct de_todo
{
    int entero;
    char caracter;
    float real;
};
```

```
struct grande
{
    int ent;
    char cara;
    struct de_todo todo;
};
```

```
struct grande var1;
```

Acceso a los miembros:

```
var1.ent
var1.cara
var1.todo.entero
var1.todo.caracter
var1.todo.real
```

Estructuras - Tamaño

Para calcular el tamaño que ocupa la estructura, como dijimos, se puede sumar lo que ocupa cada miembro.

Sino:

```
struct de_todo
{
    int entero;
    char caracter;
    float real;
};

struct de_todo var1;
```

`sizeof(struct de_todo);`

`sizeof(var1);`

Estructuras con typedef

Como vimos, **typedef** me permite definir un alias para un tipo de datos existente, entonces aprovecharemos esta herramienta para simplificar el uso de la estructura.

Lo puedo definir:

👉 En una declaración aparte:

```
struct de_todo
{
    int entero;
    char caracter;
    float real;
};

typedef struct de_todo DeTodo;
```

👉 En la misma declaración:

```
typedef struct de_todo
{
    int entero;
    char caracter;
    float real;
} DeTodo;
```

Estructuras - Acceso a los miembros con punteros

Para acceder a los miembros utilizando punteros en lugar de los nombres de las variables, debemos cambiar el operador “.” por el operador “->” quedando de esta forma lo siguiente:

```
struct de_todo
{
    int entero;
    char caracter;
    float real;
};

struct de_todo var1, *p;

p = &var1;
```

```
p->entero = 123;
p->caracter = 'C';
p->real = 456.789;
```

Utilizando punteros, ahora puedo obtener el tamaño de otras formas adicionales:

```
struct de_todo
{
    int entero;
    char caracter;
    float real;
};

struct de_todo var1, *p;

p = &var1;
```

sizeof(struct de_todo);

sizeof(var1);

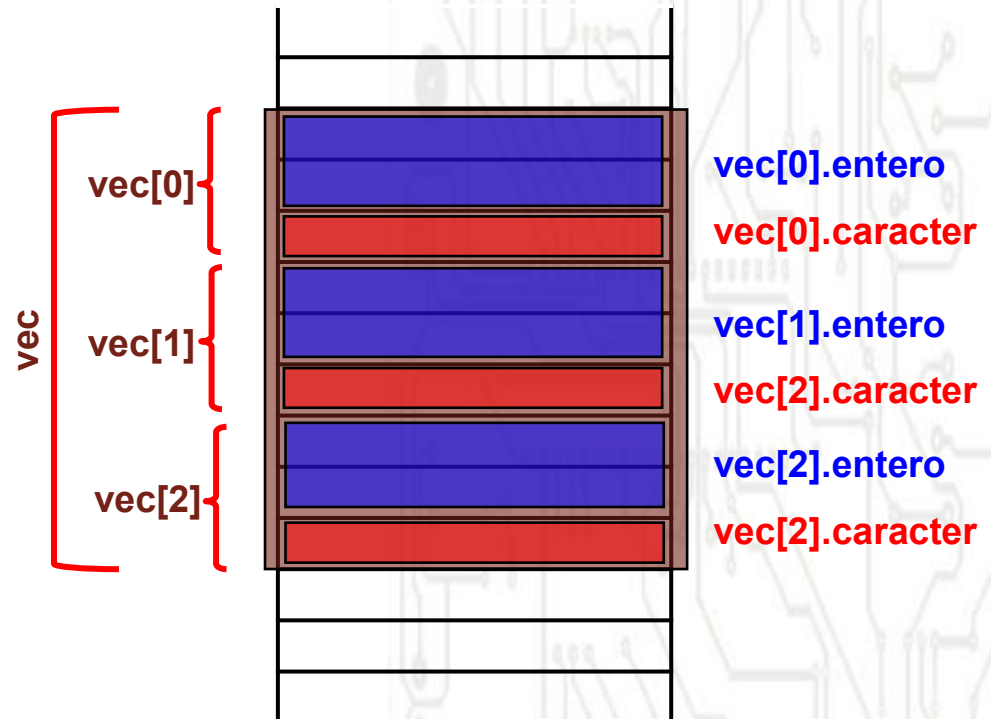
sizeof(*p);

~~sizeof(p)~~ **¡¡NO!!**

Dado que la estructura es un nuevo tipo de datos, podemos armar vectores con esta estructura:

```
typedef struct chico
{
    short int entero;
    char caracter;
} Chico;

Chico vec[3];
```

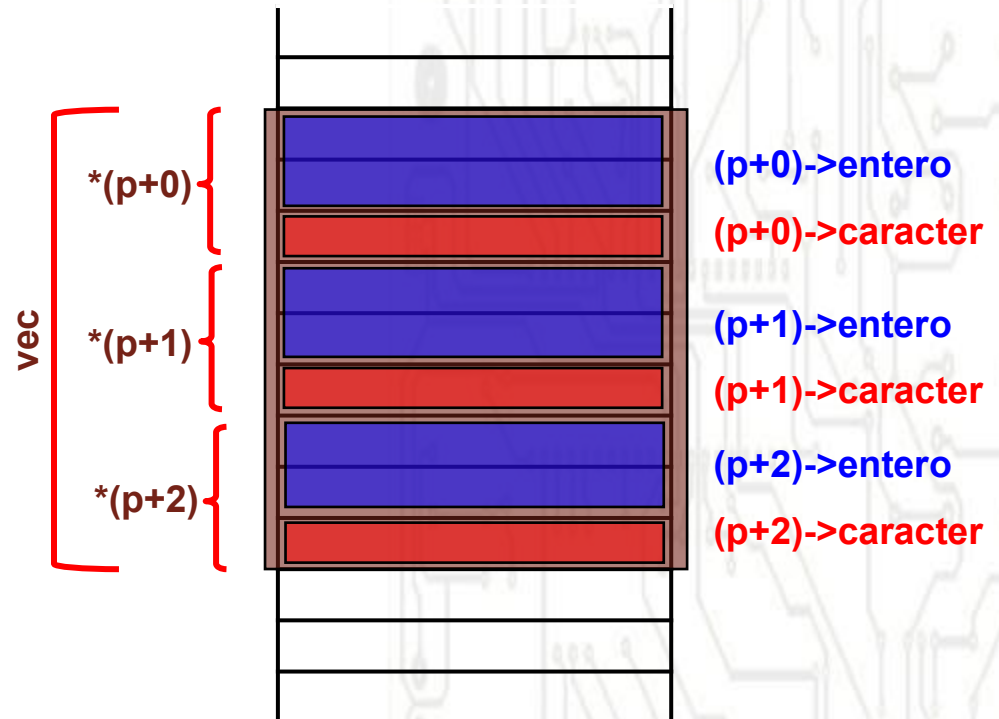


Y accediendo por punteros:

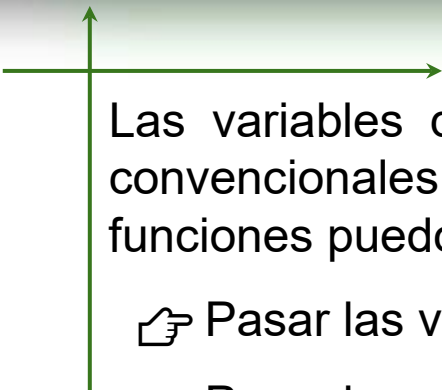
```
typedef struct chico
{
    short int entero;
    char caracter;
} Chico;

Chico vec[3], *p;

p = vec;
```



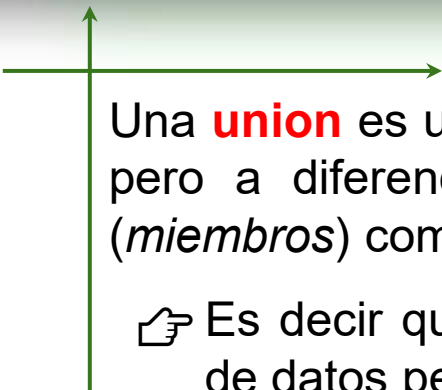
Nótese que al utilizar el operador “ \rightarrow ” no debe utilizar el operador “ $*$ ” para indicar el “*contenido de*”.



Las variables del tipo de la estructura, se utilizan como variables convencionales, por lo tanto para trabajar con estas variables en las funciones puedo:

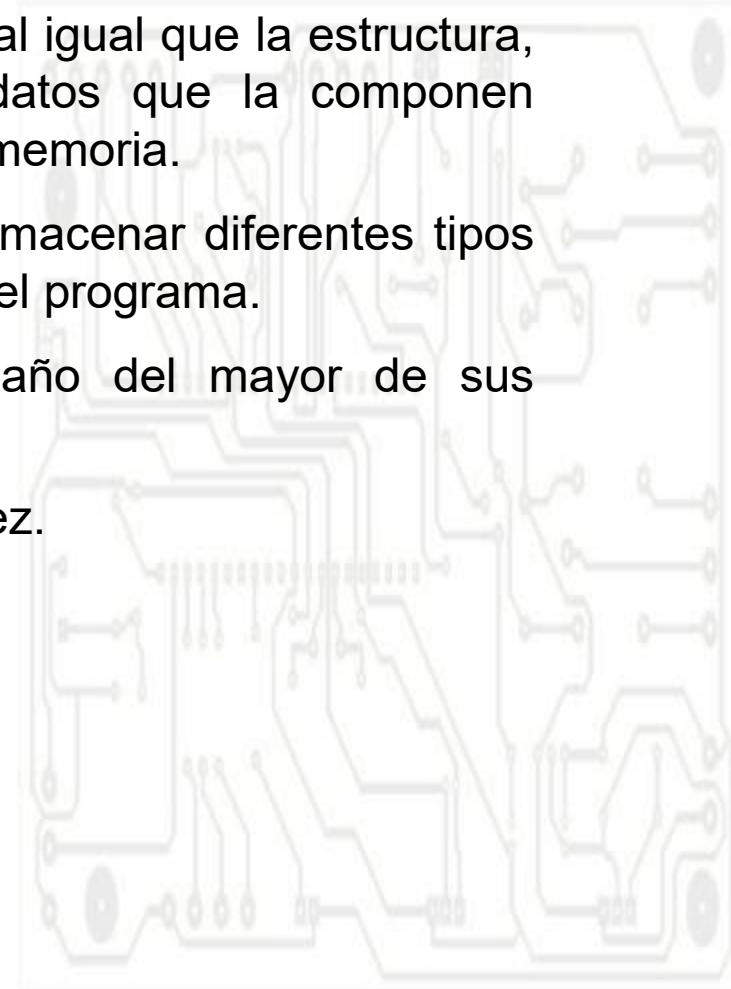
- 👉 Pasar las variables por valor
- 👉 Pasar las variables por referencia
- 👉 La función puede retornar una variable de una estructura
- 👉 Para obtener la dirección de la variable puedo usar el operador **&**

Si la estructura es grande, no conviene pasarla a la función por valor dado que ocuparía mucho lugar en el stack.



Una **union** es una herramienta del lenguaje al igual que la estructura, pero a diferencia de esta, los tipos de datos que la componen (*miembros*) comparten el mismo espacio en memoria.

- 👉 Es decir que una **union** es capaz de almacenar diferentes tipos de datos pero en diferentes momentos del programa.
- 👉 Su tamaño es el mismo que el tamaño del mayor de sus miembros.
- 👉 Solo se puede referenciar un dato a la vez.



Uniones - Acceso a los miembros

Para acceder a los miembros también se utiliza el operador “.” o el operador “->”

```
union de_todo
```

```
{
```

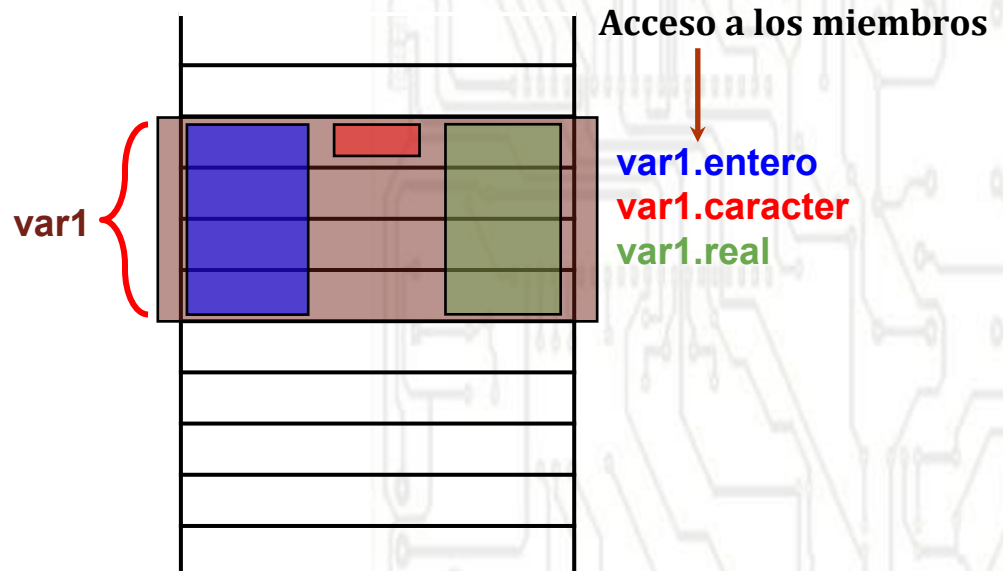
```
    int entero;
```

```
    char caracter;
```

```
    float real;
```

```
};
```

```
union de_todo var1;
```



Algunas características para el manejo de uniones son las siguientes:

- 👉 Podemos asignar a una union otra del mismo tipo.
- 👉 Podemos obtener su dirección de memoria con el operador **&**.
- 👉 No se pueden comparar (con por ejemplo los operadores **==** y **!=**)
- 👉 Cuando se asigna un dato en forma estática:

union de_todo var1 = {123.456};

se toma como referencia el tipo de dato del primer miembro. Por lo tanto en este ejemplo, el número se guardará como un **int** y se trunca la parte decimal.

Puedo calcular el tamaño de una **union** de la misma forma en que lo hacía con una estructura:

```
union de_todo
```

```
{
```

```
    int entero;
```

```
    char caracter;
```

```
    float real;
```

```
};
```

```
union de_todo var1, *p;
```

```
p = &var1;
```

```
sizeof(union de_todo);
```

```
sizeof(var1);
```

```
sizeof(*p);
```