

Comenzaremos haciendo un rápido repaso de conceptos básicos que usted debería conocer y dominar sobre el Lenguaje C .

Comenzaremos haciendo un rápido repaso de conceptos básicos que usted debería conocer y dominar sobre el **Lenguaje C**.

Pero antes: ¿Esta materia consiste en aprender a programar en **Lenguaje C** ?

Comenzaremos haciendo un rápido repaso de conceptos básicos que usted debería conocer y dominar sobre el **Lenguaje C**.

Pero antes: ¿Esta materia consiste en aprender a programar en **Lenguaje C** ?

-- No, nuestra materia trata sobre :

- Resolución de Algoritmos y
- Estructuras de datos .

Comenzaremos haciendo un rápido repaso de conceptos básicos que usted debería conocer y dominar sobre el **Lenguaje C** .

Pero antes: ¿Esta materia consiste en aprender a programar en **Lenguaje C** ?

-- No, nuestra materia trata sobre :

- Resolución de Algoritmos y
- Estructuras de datos .

El **Lenguaje C**, tan sólo es la herramienta de programación con la que resolveremos tales algoritmos, e implementaremos las Estructuras de Datos de que trata la primera parte de nuestra materia .

Comenzaremos haciendo un rápido repaso de conceptos básicos que usted debería conocer y dominar sobre el **Lenguaje C**.

Pero antes: ¿Esta materia consiste en aprender a programar en **Lenguaje C**?

-- No, nuestra materia trata sobre :

- Resolución de Algoritmos y
- Estructuras de datos .

El **Lenguaje C**, tan sólo es la herramienta de programación con la que resolveremos tales algoritmos, e implementaremos las Estructuras de Datos de que trata la primera parte de nuestra materia .

Como el **Lenguaje C** es una herramienta de '*tecnología básica*', ampliamente utilizada, usted debería incorporar esta herramienta de programación en su memoria de '*largo plazo*' . Ya verá que lo seguirá utilizando durante el resto de su carrera .

Comenzaremos haciendo un rápido repaso de conceptos básicos que usted debería conocer y dominar sobre el **Lenguaje C**.

Pero antes: ¿Esta materia consiste en aprender a programar en **Lenguaje C** ?

-- No, nuestra materia trata sobre :

- Resolución de Algoritmos y
- Estructuras de datos .

El **Lenguaje C**, tan sólo es la herramienta de programación con la que resolveremos tales algoritmos, e implementaremos las Estructuras de Datos de que trata la primera parte de nuestra materia .

Como el **Lenguaje C** es una herramienta de '*tecnología básica*', ampliamente utilizada, usted debería incorporar esta herramienta de programación en su memoria de '*largo plazo*' . Ya verá que lo seguirá utilizando durante el resto de su carrera .

En la parte final de la materia, veremos una introducción a la Programación Orientada a Objetos, para lo cual implementaremos tales temas iniciales de la POO (u OOP, de sus siglas en Inglés) valiéndonos del **Lenguaje C++**

Comenzaremos haciendo un rápido repaso de conceptos básicos que usted debería conocer y dominar sobre el **Lenguaje C**.

Pero antes: ¿Esta materia consiste en aprender a programar en **Lenguaje C**?

-- No, nuestra materia trata sobre :

- Resolución de Algoritmos y
- Estructuras de datos .

El **Lenguaje C**, tan sólo es la herramienta de programación con la que resolveremos tales algoritmos, e implementaremos las Estructuras de Datos de que trata la primera parte de nuestra materia .

Como el **Lenguaje C** es una herramienta de '*tecnología básica*', ampliamente utilizada, usted debería incorporar esta herramienta de programación en su memoria de '*largo plazo*' . Ya verá que lo seguirá utilizando durante el resto de su carrera .

En la parte final de la materia, veremos una introducción a la Programación Orientada a Objetos, para lo cual implementaremos tales temas iniciales de la POO (u OOP, de sus siglas en Inglés) valiéndonos del **Lenguaje C++**

¿Le queda claro que los lenguajes con que trabajaremos no son lo fundamental de nuestra materia?

Comenzaremos haciendo un rápido repaso de conceptos básicos que usted debería conocer y dominar sobre el **Lenguaje C**.

Pero antes: ¿Esta materia consiste en aprender a programar en **Lenguaje C**?

-- No, nuestra materia trata sobre :

- Resolución de Algoritmos y
- Estructuras de datos .

El **Lenguaje C**, tan sólo es la herramienta de programación con la que resolveremos tales algoritmos, e implementaremos las Estructuras de Datos de que trata la primera parte de nuestra materia .

Como el **Lenguaje C** es una herramienta de '*tecnología básica*', ampliamente utilizada, usted debería incorporar esta herramienta de programación en su memoria de '*largo plazo*' . Ya verá que lo seguirá utilizando durante el resto de su carrera .

En la parte final de la materia, veremos una introducción a la Programación Orientada a Objetos, para lo cual implementaremos tales temas iniciales de la POO (u OOP, de sus siglas en Inglés) valiéndonos del **Lenguaje C++**

¿Le queda claro que los lenguajes con que trabajaremos no son lo fundamental de nuestra materia?

¿Le queda claro también que para usted es muy importante poderlos dominar para así poder avanzar con los contenidos reales de nuestra materia?

Comenzaremos haciendo un rápido repaso de conceptos básicos que usted debería conocer y dominar sobre el **Lenguaje C** .

Pero antes: ¿Esta materia consiste en aprender a programar en **Lenguaje C** ?

-- No, nuestra materia trata sobre :

- Resolución de Algoritmos y
- Estructuras de datos .

El **Lenguaje C**, tan sólo es la herramienta de programación con la que resolveremos tales algoritmos, e implementaremos las Estructuras de Datos de que trata la primera parte de nuestra materia .

Como el **Lenguaje C** es una herramienta de '*tecnología básica*', ampliamente utilizada, usted debería incorporar esta herramienta de programación en su memoria de '*largo plazo*' . Ya verá que lo seguirá utilizando durante el resto de su carrera .

En la parte final de la materia, veremos una introducción a la Programación Orientada a Objetos, para lo cual implementaremos tales temas iniciales de la POO (u OOP, de sus siglas en Inglés) valiéndonos del **Lenguaje C++**

¿Le queda claro que los lenguajes con que trabajaremos no son lo fundamental de nuestra materia?

¿Le queda claro también que para usted es muy importante poderlos dominar para así poder avanzar con los contenidos reales de nuestra materia?

Por esto es que dedicaremos estas primeras horas de clase a dar un rápido repaso a temas del **Lenguaje C** que usted debería ya conocer a la vez que introduciremos otros conceptos nuevos, para usted, del lenguaje .

Comenzaremos haciendo un rápido repaso de conceptos básicos que usted debería conocer y dominar sobre el **Lenguaje C**.

Pero antes: ¿Esta materia consiste en aprender a programar en **Lenguaje C**?

-- No, nuestra materia trata sobre :

- Resolución de Algoritmos y
- Estructuras de datos .

El **Lenguaje C**, tan sólo es la herramienta de programación con la que resolveremos tales algoritmos, e implementaremos las Estructuras de Datos de que trata la primera parte de nuestra materia .

Como el **Lenguaje C** es una herramienta de '*tecnología básica*', ampliamente utilizada, usted debería incorporar esta herramienta de programación en su memoria de '*largo plazo*' . Ya verá que lo seguirá utilizando durante el resto de su carrera .

En la parte final de la materia, veremos una introducción a la Programación Orientada a Objetos, para lo cual implementaremos tales temas iniciales de la POO (u OOP, de sus siglas en Inglés) valiéndonos del **Lenguaje C++**

¿Le queda claro que los lenguajes con que trabajaremos no son lo fundamental de nuestra materia?

¿Le queda claro también que para usted es muy importante poderlos dominar para así poder avanzar con los contenidos reales de nuestra materia?

Por esto es que dedicaremos estas primeras horas de clase a dar un rápido repaso a temas del **Lenguaje C** que usted debería ya conocer a la vez que introduciremos otros conceptos nuevos, para usted, del lenguaje .

Usted debería complementar las clases con la lectura de una buena bibliografía .

Un programa en **Lenguaje C** (o simplemente **C**) se puede almacenar en al menos un archivo de texto (por ej.: `prog-1.c`) con un nombre (`prog-1`) y un punto (`.`) que separa la extensión del nombre del archivo (`c`) .

Un programa en **Lenguaje C** (o simplemente **C**) se puede almacenar en al menos un archivo de texto (por ej.: **prog-1.c**) con un nombre (**prog-1**) y un punto (.) que separa la extensión del nombre del archivo (**c**) .

Dentro de dicho(s) archivo(s) que compone(n) el programa debe haber una única función identificada con el identificador **main** .

Un programa en Lenguaje C (o simplemente C) se puede almacenar en al menos un archivo de texto (por ej.: `prog-1.c`) con un nombre (`prog-1`) y un punto (.) que separa la extensión del nombre del archivo (`c`) .

Dentro de dicho(s) archivo(s) que compone(n) el programa debe haber una única función identificada con el identificador `main` .

Además de la función `main` se podrá hacer uso diversas funciones . Estas pueden ser funciones de bibliotecas del compilador o desarrolladas por el programador .

Un programa en **Lenguaje C** (o simplemente **C**) se puede almacenar en al menos un archivo de texto (por ej.: **prog-1.c**) con un nombre (**prog-1**) y un punto (.) que separa la extensión del nombre del archivo (**c**) .

Dentro de dicho(s) archivo(s) que compone(n) el programa debe haber una única función identificada con el identificador **main** .

Además de la función **main** se podrá hacer uso de diversas funciones . Estas pueden ser funciones de bibliotecas del compilador o desarrolladas por el programador .

De este modo el programa más breve que se puede compilar y ejecutar en **C** podría ser :

```
main( )  
{  
  
}
```


Un programa en Lenguaje C (o simplemente C) se puede almacenar en al menos un archivo de texto (por ej.: `prog-1.c`) con un nombre (`prog-1`) y un punto (.) que separa la extensión del nombre del archivo (`c`) .

Dentro de dicho(s) archivo(s) que compone(n) el programa debe haber una única función identificada con el identificador `main` .

Además de la función `main` se podrá hacer uso de diversas funciones . Estas pueden ser funciones de bibliotecas del compilador o desarrolladas por el programador .

De este modo el programa más breve que se puede compilar y ejecutar en C podría ser :

```
main( )  
{  
  
}
```

El código de la función `main` será el que se comience a ejecutar en primer lugar .

Un programa en **Lenguaje C** (o simplemente **C**) se puede almacenar en al menos un archivo de texto (por ej.: **prog-1.c**) con un nombre (**prog-1**) y un punto (.) que separa la extensión del nombre del archivo (**c**) .

Dentro de dicho(s) archivo(s) que compone(n) el programa debe haber una única función identificada con el identificador **main** .

Además de la función **main** se podrá hacer uso diversas funciones . Estas pueden ser funciones de bibliotecas del compilador o desarrolladas por el programador .

De este modo el programa más breve que se puede compilar y ejecutar en **C** podría ser :

```
main( )  
{
```

El código de la función **main** será el que se comience a ejecutar en primer lugar .

Cada archivo fuente se compilará por separado, y luego se vinculará (link) todo en conjunto para producir el código ejecutable .

Un programa en **Lenguaje C** (o simplemente **C**) se puede almacenar en al menos un archivo de texto (por ej.: **prog-1.c**) con un nombre (**prog-1**) y un punto (.) que separa la extensión del nombre del archivo (**c**) .

Dentro de dicho(s) archivo(s) que compone(n) el programa debe haber una única función identificada con el identificador **main** .

Además de la función **main** se podrá hacer uso diversas funciones . Estas pueden ser funciones de bibliotecas del compilador o desarrolladas por el programador .

De este modo el programa más breve que se puede compilar y ejecutar en **C** podría ser :

```
main( )  
{
```

El código de la función **main** será el que se comience a ejecutar en primer lugar .

Cada archivo fuente se compilará por separado, y luego se vinculará (link) todo en conjunto para producir el código ejecutable .

Las funciones, variables, tipos de datos, etc., que declaremos en nuestro programa **C**, los podremos nombrar según nuestro gusto, pero respetando algunas pocas reglas de sintaxis .

Un programa en **Lenguaje C** (o simplemente **C**) se puede almacenar en al menos un archivo de texto (por ej.: **prog-1.c**) con un nombre (**prog-1**) y un punto (.) que separa la extensión del nombre del archivo (**c**) .

Dentro de dicho(s) archivo(s) que compone(n) el programa debe haber una única función identificada con el identificador **main** .

Además de la función **main** se podrá hacer uso de diversas funciones . Estas pueden ser funciones de bibliotecas del compilador o desarrolladas por el programador .

De este modo el programa más breve que se puede compilar y ejecutar en **C** podría ser :

```
main( )  
{
```

El código de la función **main** será el que se comience a ejecutar en primer lugar .

Cada archivo fuente se compilará por separado, y luego se vinculará (link) todo en conjunto para producir el código ejecutable .

Las funciones, variables, tipos de datos, etc., que declaremos en nuestro programa **C**, los podremos nombrar según nuestro gusto, pero respetando algunas pocas reglas de sintaxis .

Un identificador debe comenzar con una letra del alfabeto inglés en mayúscula o minúscula (no vale la letra ñ o Ñ que nos distingue, ni vocales con acento ni la ü con diéresis) .

Un programa en **Lenguaje C** (o simplemente **C**) se puede almacenar en al menos un archivo de texto (por ej.: **prog-1.c**) con un nombre (**prog-1**) y un punto (.) que separa la extensión del nombre del archivo (**c**) .

Dentro de dicho(s) archivo(s) que compone(n) el programa debe haber una única función identificada con el identificador **main** .

Además de la función **main** se podrá hacer uso de diversas funciones . Estas pueden ser funciones de bibliotecas del compilador o desarrolladas por el programador .

De este modo el programa más breve que se puede compilar y ejecutar en **C** podría ser :

```
main( )  
{
```

El código de la función **main** será el que se comience a ejecutar en primer lugar .

Cada archivo fuente se compilará por separado, y luego se vinculará (link) todo en conjunto para producir el código ejecutable .

Las funciones, variables, tipos de datos, etc., que declaremos en nuestro programa **C**, los podremos nombrar según nuestro gusto, pero respetando algunas pocas reglas de sintaxis .

Un identificador debe comenzar con una letra del alfabeto inglés en mayúscula o minúscula (no vale la letra ñ o Ñ que nos distingue, ni vocales con acento ni la ü con diéresis) .

Para el **Lenguaje C** el carácter subguión (**_**) es una letra más .

Un programa en **Lenguaje C** (o simplemente **C**) se puede almacenar en al menos un archivo de texto (por ej.: **prog-1.c**) con un nombre (**prog-1**) y un punto (.) que separa la extensión del nombre del archivo (**c**) .

Dentro de dicho(s) archivo(s) que compone(n) el programa debe haber una única función identificada con el identificador **main** .

Además de la función **main** se podrá hacer uso de diversas funciones . Estas pueden ser funciones de bibliotecas del compilador o desarrolladas por el programador .

De este modo el programa más breve que se puede compilar y ejecutar en **C** podría ser :

```
main( )  
{
```

El código de la función **main** será el que se comience a ejecutar en primer lugar .

Cada archivo fuente se compilará por separado, y luego se vinculará (link) todo en conjunto para producir el código ejecutable .

Las funciones, variables, tipos de datos, etc., que declaremos en nuestro programa **C**, los podremos nombrar según nuestro gusto, pero respetando algunas pocas reglas de sintaxis .

Un identificador debe comenzar con una letra del alfabeto inglés en mayúscula o minúscula (no vale la letra ñ o Ñ que nos distingue, ni vocales con acento ni la ü con diéresis) .

Para el **Lenguaje C** el carácter subguión (**_**) es una letra más .

Hay que recordar que **C** es sensible a mayúsculas y minúsculas (case sensitive), con lo que los identificadores **MAIN**, **MaIn**, **MaIn**, ... podrían ser distintos identificadores válidos .

La función **main**, funciones e identificadores . Palabras reservadas .

Un programa en **Lenguaje C** (o simplemente **C**) se puede almacenar en al menos un archivo de texto (por ej.: **prog-1.c**) con un nombre (**prog-1**) y un punto (.) que separa la extensión del nombre del archivo (**c**) .

Dentro de dicho(s) archivo(s) que compone(n) el programa debe haber una única función identificada con el identificador **main** .

Además de la función **main** se podrá hacer uso de diversas funciones . Estas pueden ser funciones de bibliotecas del compilador o desarrolladas por el programador .

De este modo el programa más breve que se puede compilar y ejecutar en **C** podría ser :

```
main( )  
{
```

El código de la función **main** será el que se comience a ejecutar en primer lugar .

Cada archivo fuente se compilará por separado, y luego se vinculará (link) todo en conjunto para producir el código ejecutable .

Las funciones, variables, tipos de datos, etc., que declaremos en nuestro programa **C**, los podremos nombrar según nuestro gusto, pero respetando algunas pocas reglas de sintaxis .

Un identificador debe comenzar con una letra del alfabeto inglés en mayúscula o minúscula (diéresis) .

Para el **Le**

A continuación puede continuar con cualquier carácter representativo de una letra (incluso el **_**), o los dígitos del 0 al 9 . Los siguientes son identificadores válidos .

Hay que re
lo que los
válidos .

x, **kxp1**, **_**, **__**, **ciclo**, **func**, **valor**, **algo**, **var1**, **var2**,
var3, **i**, **j**, **k**, **l**, **m**

La función **main**, funciones e identificadores . Palabras reservadas .

Un programa
archivo de
separa la

Dentro de
función id

Además d
ser funcio

De todos modos, aunque identificadores como `x`, `y`, `z`, `_`, `__`, `var1`, `var2`, `var3`, `i`, `j`, `k`, `l`, `m`; sean identificadores válidos, pueden llegar a ser poco representativos de para qué se emplean, por lo que además respetaremos reglas de estilo .
Los identificadores que elijamos serán representativos de su empleo .
Además, un identificador no debe coincidir con alguna de las palabras reservadas del C .

De este modo el programa más breve que se puede compilar y ejecutar en C podría ser :

```
main( )  
{
```

El código de la función main será el que se comience a ejecutar en primer lugar .

Cada archivo fuente se compilará por separado, y luego se vinculará (link) todo en conjunto para producir el código ejecutable .

Las funciones, variables, tipos de datos, etc., que declaremos en nuestro programa C, los podremos nombrar según nuestro gusto, pero respetando algunas pocas reglas de sintaxis .

Un identificador debe comenzar con una letra del alfabeto inglés en mayúscula o minúscula (o diéresis) .

Para el Le

A continuación puede continuar con cualquier carácter representativo de una letra (incluso el `_`), o los dígitos del 0 al 9 . Los siguientes son identificadores válidos .

Hay que re
lo que los
válidos .

`x`, `kxp1`, `_`, `__`, `ciclo`, `func`, `valor`, `algo`, `var1`, `var2`, `var3`, `i`, `j`, `k`, `l`, `m`

La función `main`, funciones e identificadores . Palabras reservadas .

Un programa
archivo de
separa la

Dentro de
función id

Además d
ser funcio

De este m
ser :

```
main  
{
```

Cada arch
conjunto p

Las funcio
los podre
de sintaxis

Un identifi
minúscula
diéresis) .

Para el Le

Hay que re
lo que los
válidos .

De todos modos, aunque identificadores como `x`, `y`, `z`, `_`, `__`, `var1`, `var2`, `var3`, `i`, `j`, `k`, `l`, `m`; sean identificadores válidos, pueden llegar a ser poco representativos de para qué se emplean, por lo que además respetaremos reglas de estilo .

Los identificadores que elijamos serán representativos de su empleo . Además, un identificador no debe coincidir con alguna de las palabras reservadas del C .

<code>asm</code>	<code>auto</code>	<code>bad_cast</code>	<code>bad_typeid</code>
<code>bool</code>	<code>break</code>	<code>case</code>	<code>catch</code>
<code>char</code>	<code>class</code>	<code>const</code>	<code>const_cast</code>
<code>continue</code>	<code>default</code>	<code>delete</code>	<code>do</code>
<code>double</code>	<code>dynamic_cast</code>	<code>else</code>	<code>enum</code>
<code>except</code>	<code>explicit</code>	<code>extern</code>	<code>false</code>
<code>finally</code>	<code>float</code>	<code>for</code>	<code>friend</code>
<code>goto</code>	<code>if</code>	<code>inline</code>	<code>int</code>
<code>long</code>	<code>mutable</code>	<code>namespace</code>	<code>new</code>
<code>operator</code>	<code>private</code>	<code>protected</code>	<code>public</code>
<code>register</code>	<code>reinterpret_cast</code>	<code>return</code>	<code>short</code>
<code>signed</code>	<code>sizeof</code>	<code>static</code>	<code>static_cast</code>
<code>struct</code>	<code>switch</code>	<code>template</code>	<code>this</code>
<code>throw</code>	<code>true</code>	<code>try</code>	<code>type_info</code>
<code>typedef</code>	<code>typeid</code>	<code>typename</code>	<code>union</code>
<code>unsigned</code>	<code>using</code>	<code>virtual</code>	<code>void</code>
<code>volatile</code>	<code>while</code>		

La función main, funciones e identificadores . Palabras reservadas .

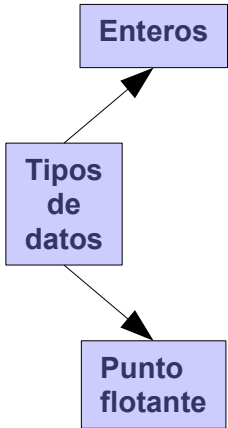
Las palabras con fondo gris son agregadas por el C++ .

Además de estas, algún compilador puede contemplar otras palabras reservadas, o incluso diferir en algunas (como por ejemplo : `asm`) .

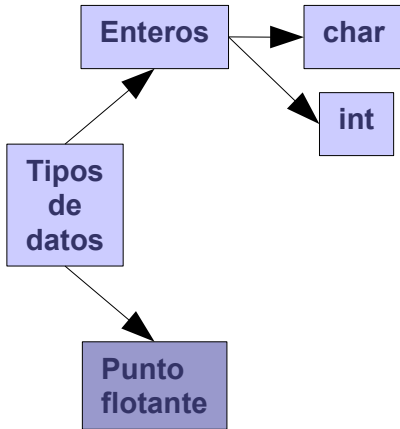
<code>asm</code>	<code>auto</code>	<code>bad_cast</code>	<code>bad_typeid</code>
<code>bool</code>	<code>break</code>	<code>case</code>	<code>catch</code>
<code>char</code>	<code>class</code>	<code>const</code>	<code>const_cast</code>
<code>continue</code>	<code>default</code>	<code>delete</code>	<code>do</code>
<code>double</code>	<code>dynamic_cast</code>	<code>else</code>	<code>enum</code>
<code>except</code>	<code>explicit</code>	<code>extern</code>	<code>false</code>
<code>finally</code>	<code>float</code>	<code>for</code>	<code>friend</code>
<code>goto</code>	<code>if</code>	<code>inline</code>	<code>int</code>
<code>long</code>	<code>mutable</code>	<code>namespace</code>	<code>new</code>
<code>operator</code>	<code>private</code>	<code>protected</code>	<code>public</code>
<code>register</code>	<code>reinterpret_cast</code>	<code>return</code>	<code>short</code>
<code>signed</code>	<code>sizeof</code>	<code>static</code>	<code>static_cast</code>
<code>struct</code>	<code>switch</code>	<code>template</code>	<code>this</code>
<code>throw</code>	<code>true</code>	<code>try</code>	<code>type_info</code>
<code>typedef</code>	<code>typeid</code>	<code>typename</code>	<code>union</code>
<code>unsigned</code>	<code>using</code>	<code>virtual</code>	<code>void</code>
<code>volatile</code>	<code>while</code>		

La función `main`, funciones e identificadores . Palabras reservadas .

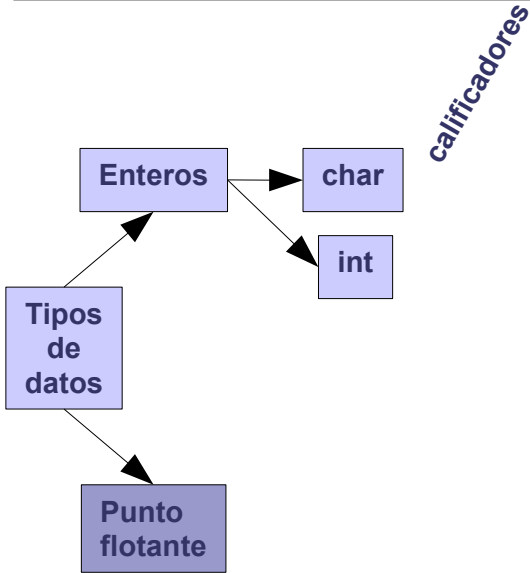
Los tipos de datos del `C` pertenecen 'casi' exclusivamente a dos grandes grupos .



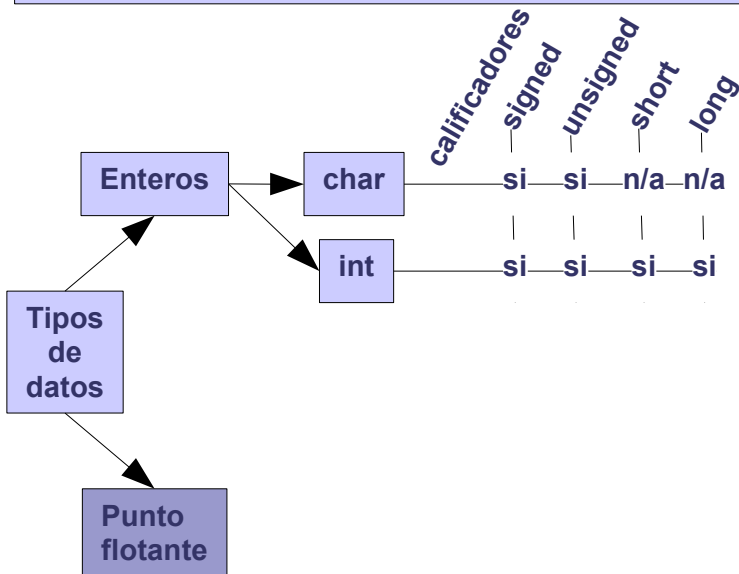
Los tipos de datos del `C` pertenecen 'casi' exclusivamente a dos grandes grupos .



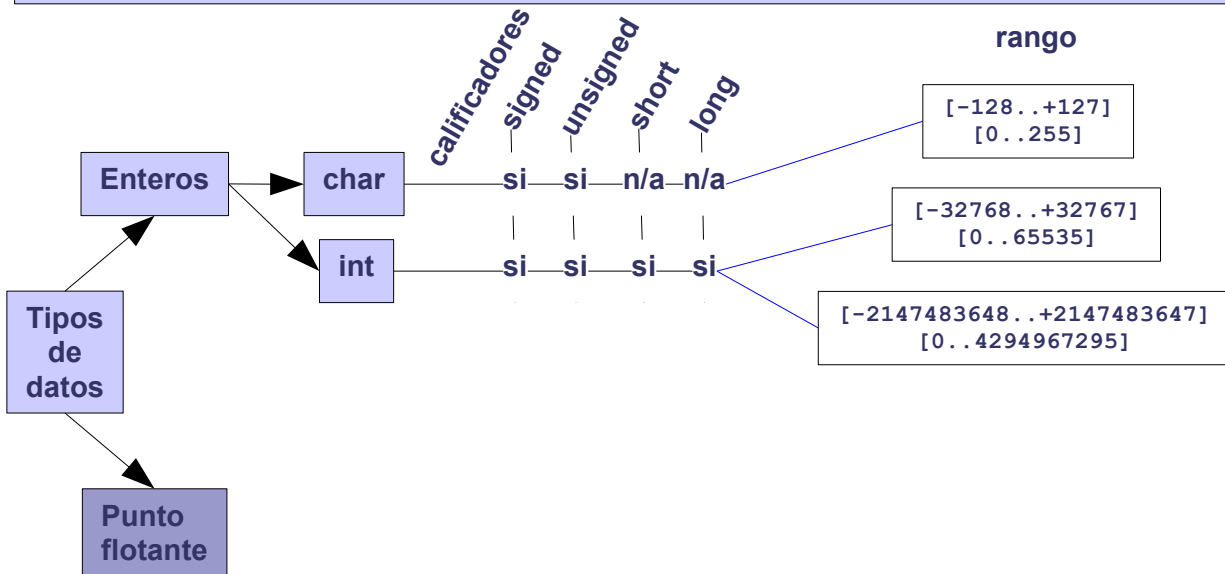
Los tipos de datos del `C` pertenecen 'casi' exclusivamente a dos grandes grupos .



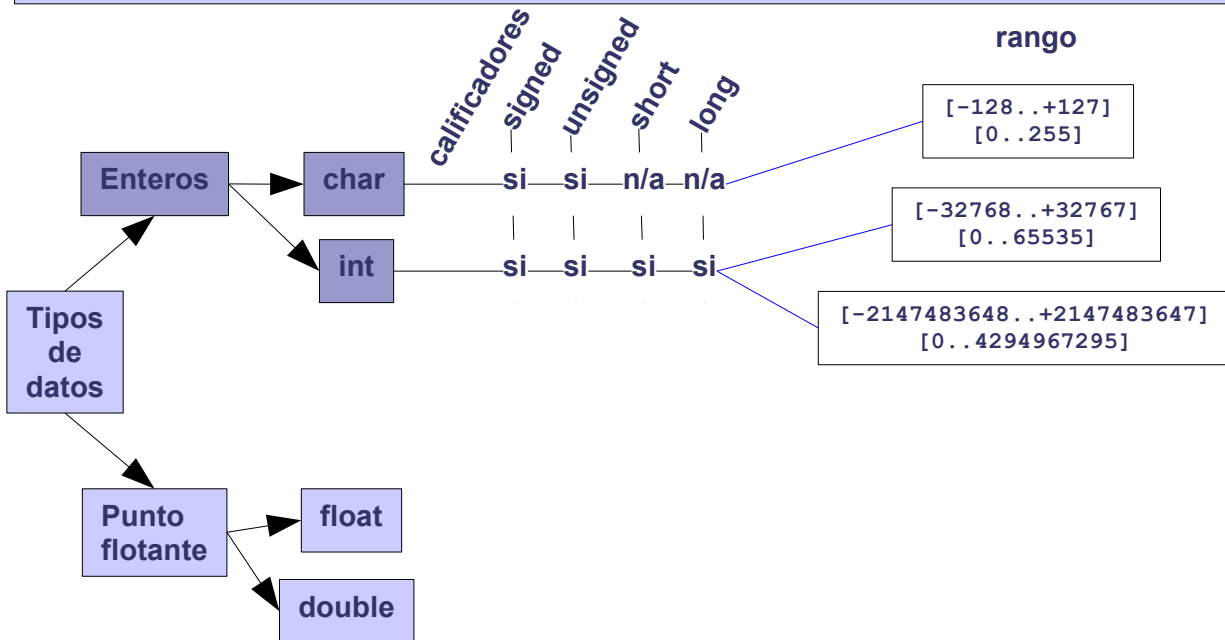
Los tipos de datos del `C` pertenecen 'casi' exclusivamente a dos grandes grupos .



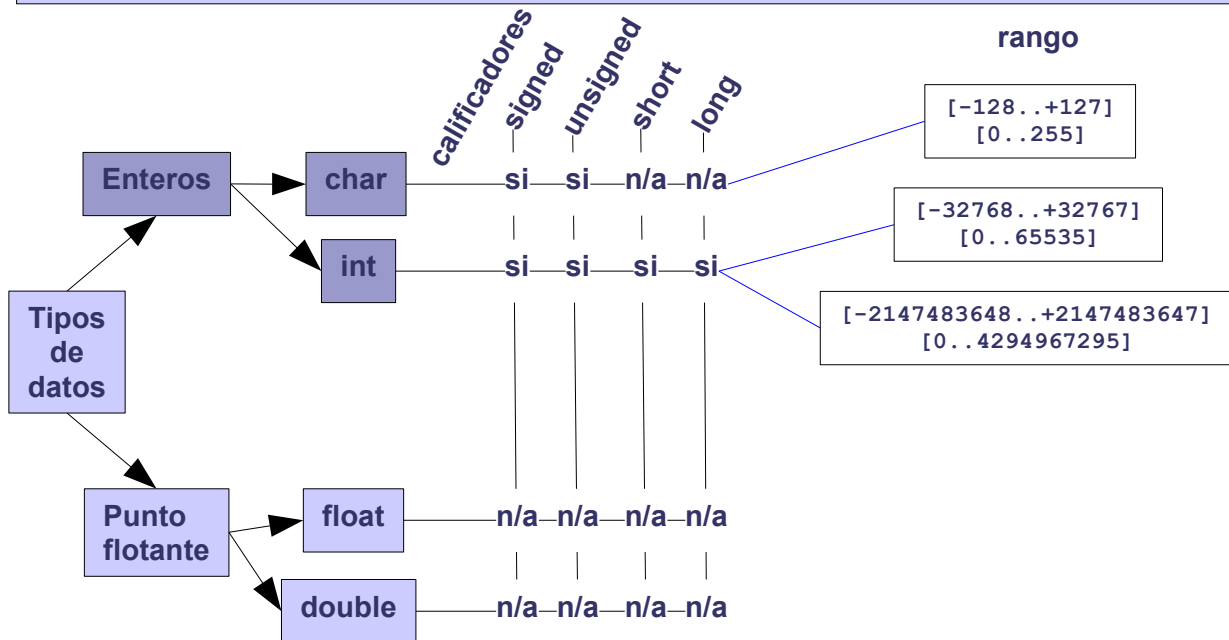
Los tipos de datos del C pertenecen 'casi' exclusivamente a dos grandes grupos .



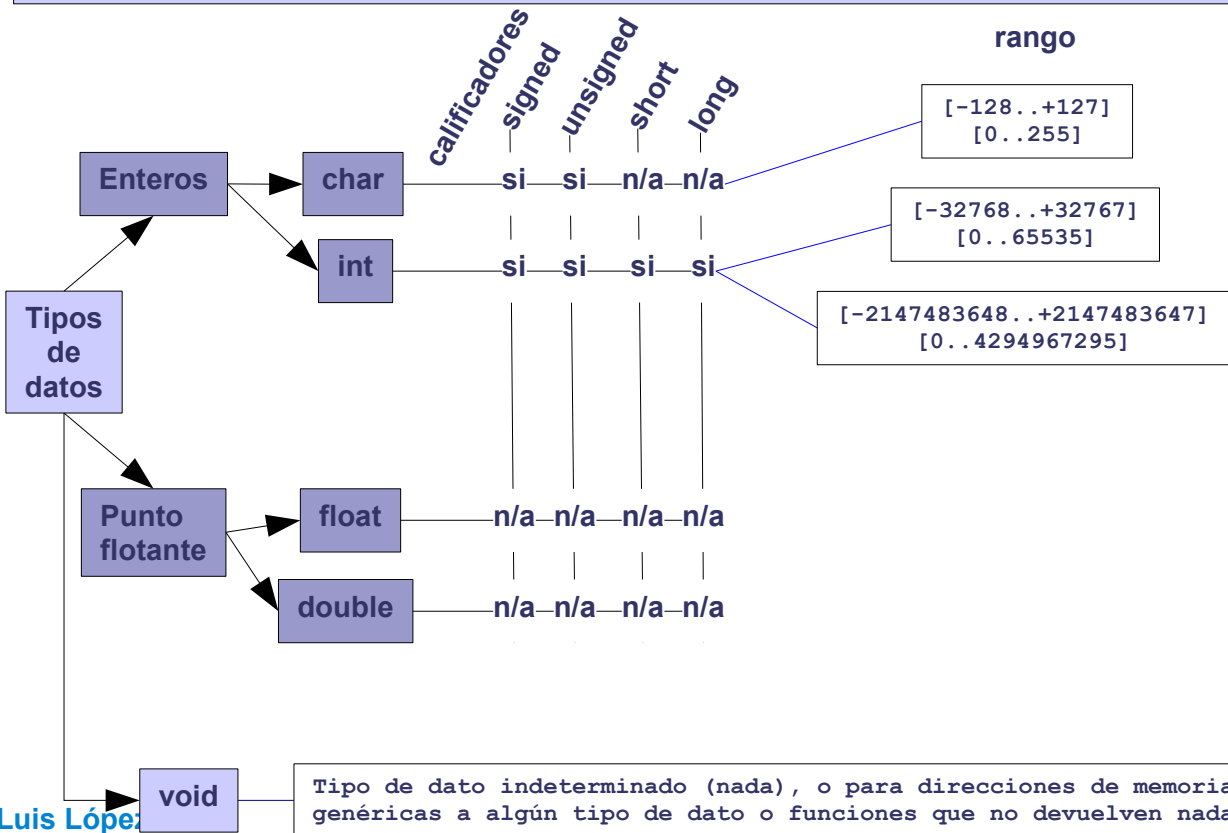
Los tipos de datos del C pertenecen 'casi' exclusivamente a dos grandes grupos .



Los tipos de datos del C pertenecen 'casi' exclusivamente a dos grandes grupos .



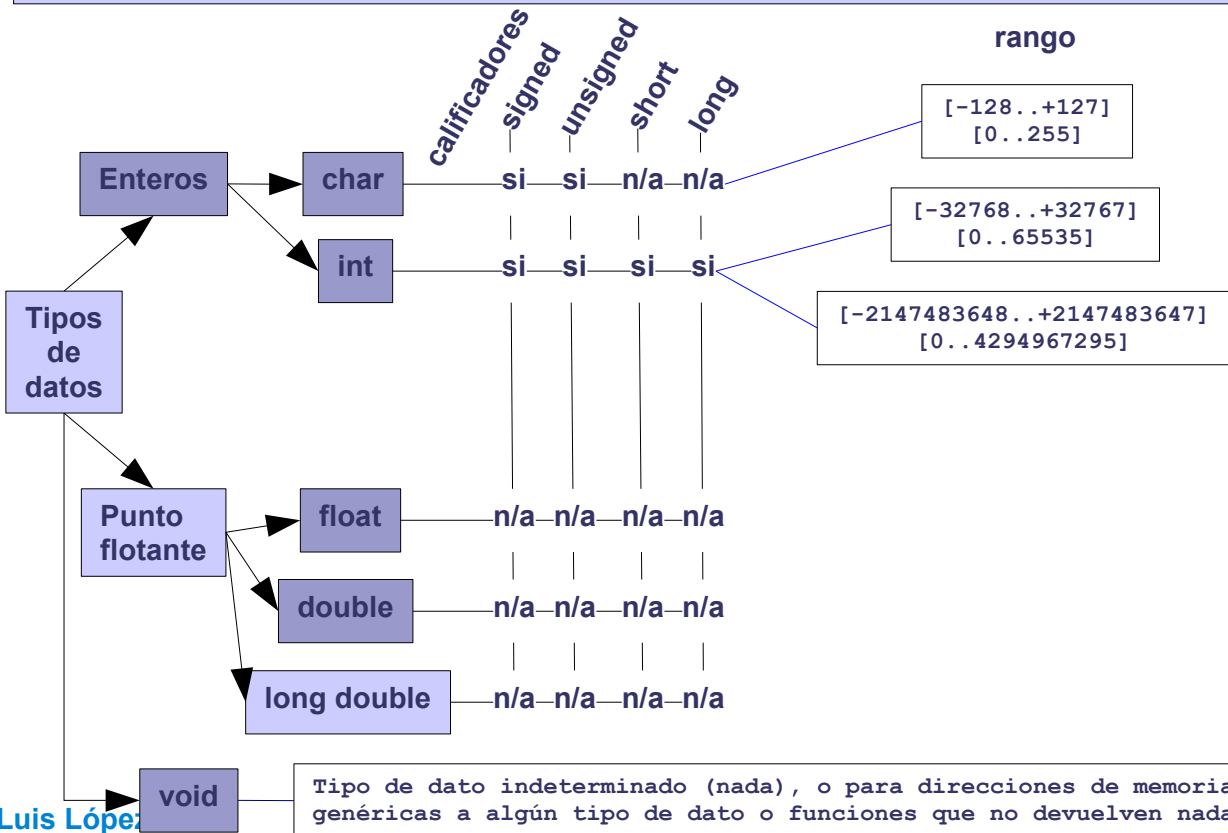
Los tipos de datos del C pertenecen 'casi' exclusivamente a dos grandes grupos .



Tipo de dato indeterminado (nada), o para direcciones de memoria genéricas a algún tipo de dato o funciones que no devuelven nada

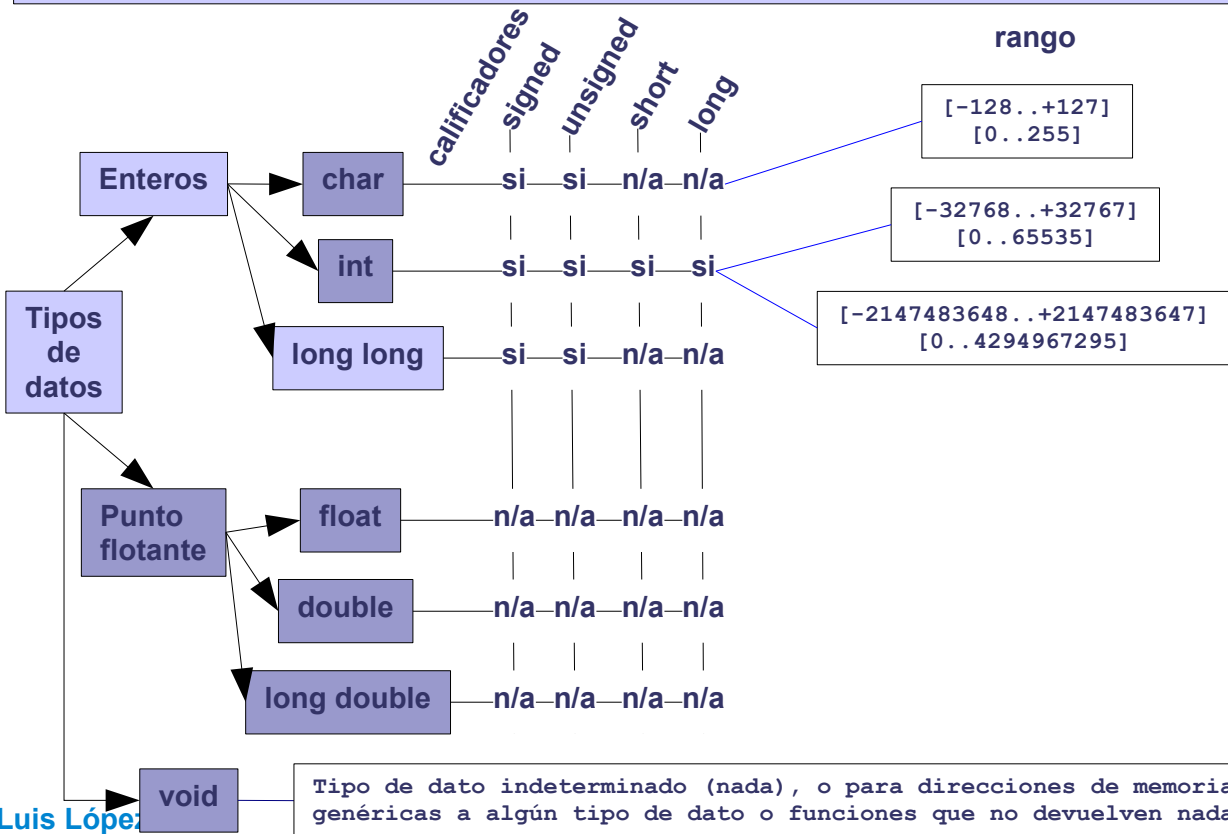
Tipos de datos y tipos de variables propios del Lenguaje C .

Los tipos de datos del C pertenecen 'casi' exclusivamente a dos grandes grupos .



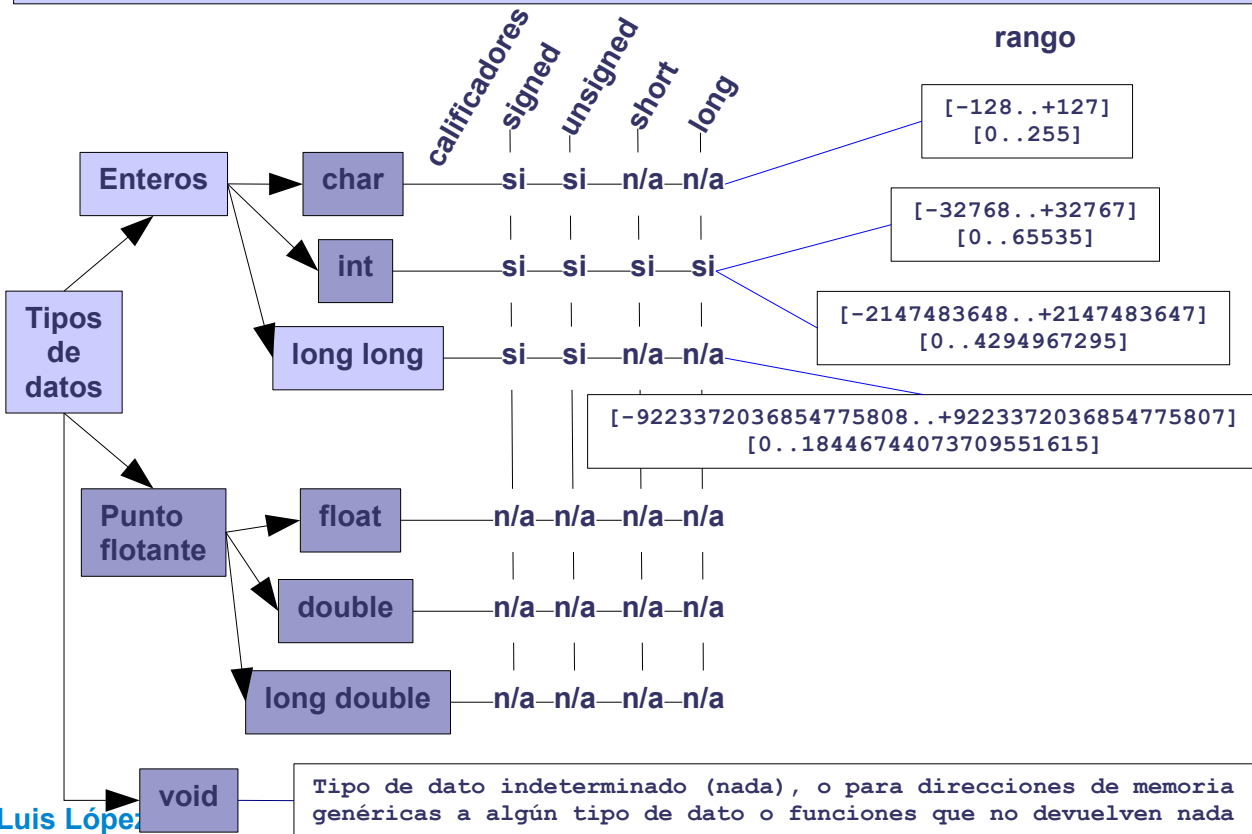
Tipos de datos y tipos de variables propios del Lenguaje C .

Los tipos de datos del C pertenecen 'casi' exclusivamente a dos grandes grupos .



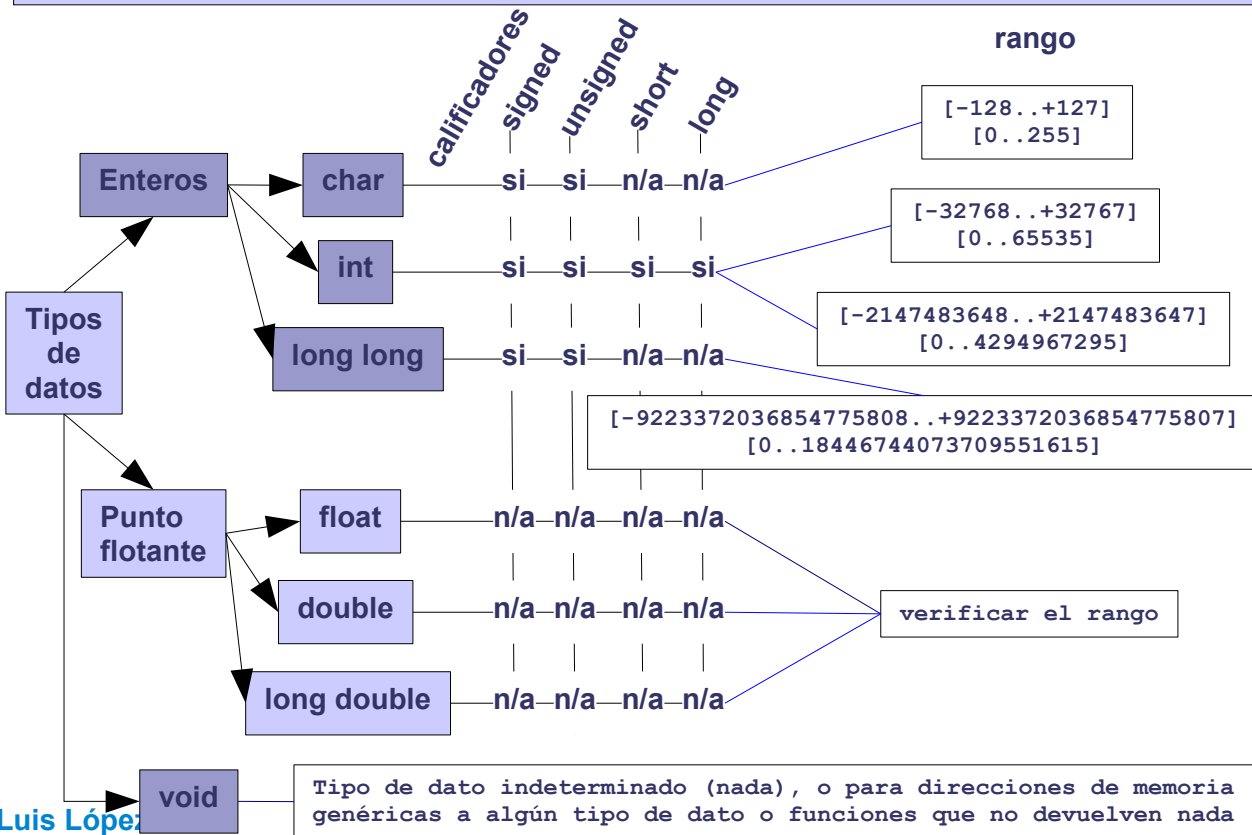
Tipos de datos y tipos de variables propios del Lenguaje C .

Los tipos de datos del C pertenecen 'casi' exclusivamente a dos grandes grupos .



Tipos de datos y tipos de variables propios del Lenguaje C .

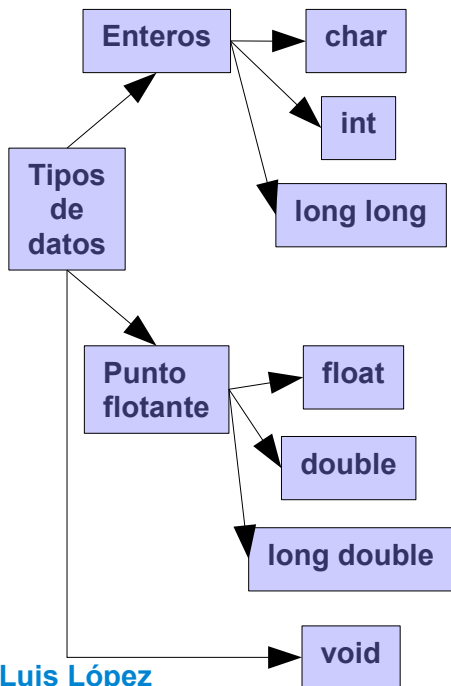
Los tipos de datos del C pertenecen 'casi' exclusivamente a dos grandes grupos .



Tipos de datos y tipos de variables propios del Lenguaje C .

Los tipos de datos del `C` pertenecen 'casi' exclusivamente a dos grandes grupos .

Como se puede observar, en `C` no se han previsto tipos de datos alfanuméricos .



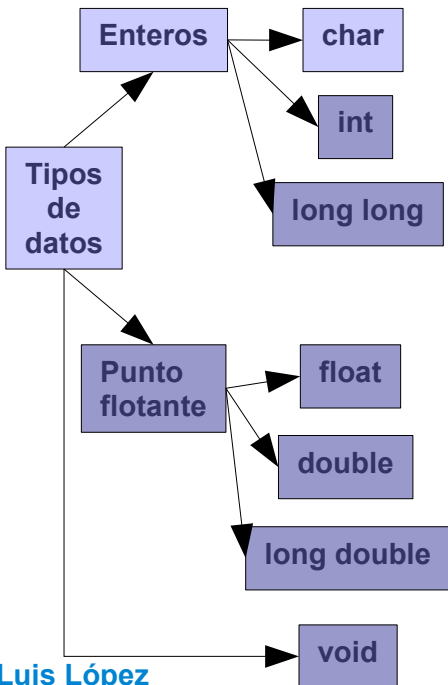
Luis López

Tipos de datos y tipos de variables propios del Lenguaje `C` .

Los tipos de datos del `C` pertenecen 'casi' exclusivamente a dos grandes grupos .

Como se puede observar, en `C` no se han previsto tipos de datos alfanuméricos .

Estos tipos de contenidos (carácteres o cadenas de carácteres) se pueden almacenar en variables o en arrays de tipo `char` .



Luis López

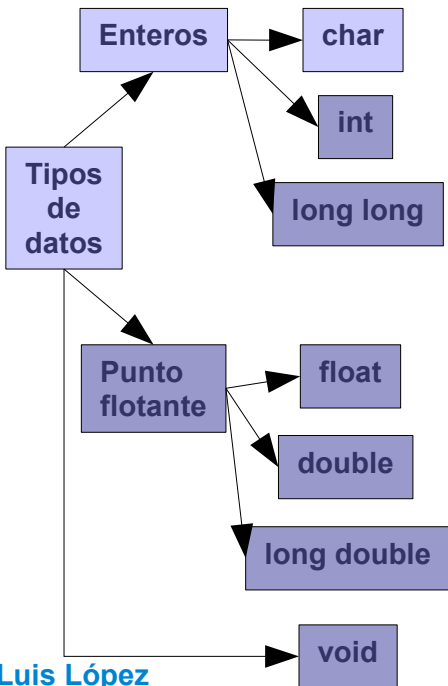
Tipos de datos y tipos de variables propios del Lenguaje `C` .

Los tipos de datos del `C` pertenecen 'casi' exclusivamente a dos grandes grupos .

Como se puede observar, en `C` no se han previsto tipos de datos alfanuméricos .

Estos tipos de contenidos (carácteres o cadenas de carácteres) se pueden almacenar en variables o en arrays de tipo `char` .

Lo que en realidad se almacenan son los **números de ASCII** (como números enteros, con los que se puede operar) de los carácteres o cadenas de carácteres .



Tipos de datos y tipos de variables propios del Lenguaje `C` .

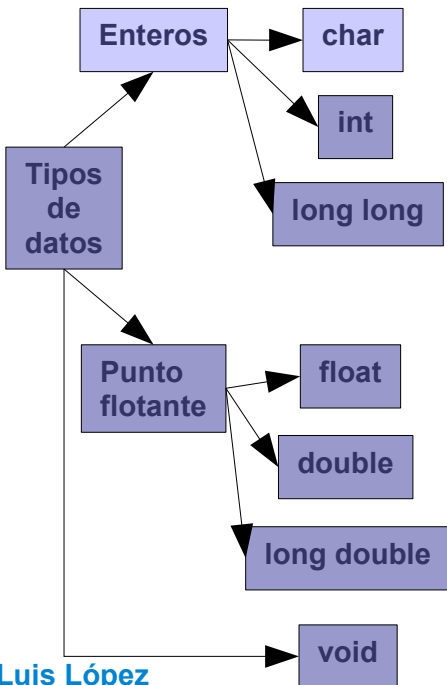
Los tipos de datos del `C` pertenecen 'casi' exclusivamente a dos grandes grupos .

Como se puede observar, en `C` no se han previsto tipos de datos alfanuméricos .

Estos tipos de contenidos (carácteres o cadenas de carácteres) se pueden almacenar en variables o en arrays de tipo `char` .

Lo que en realidad se almacenan son los **números de ASCII** (como números enteros, con los que se puede operar) de los carácteres o cadenas de carácteres .

A los efectos de su manejo (simulando estos tipos alfanuméricos) el compilador dispone de potentes funciones de biblioteca que permiten realizar las operaciones de entrada/salida de cadenas de carácteres, copia de las mismas, búsqueda de carácteres en una cadena (array de enteros `char`), etc. .



Tipos de datos y tipos de variables propios del Lenguaje `C` .

Los tipos de datos del `C` pertenecen 'casi' exclusivamente a dos grandes grupos .

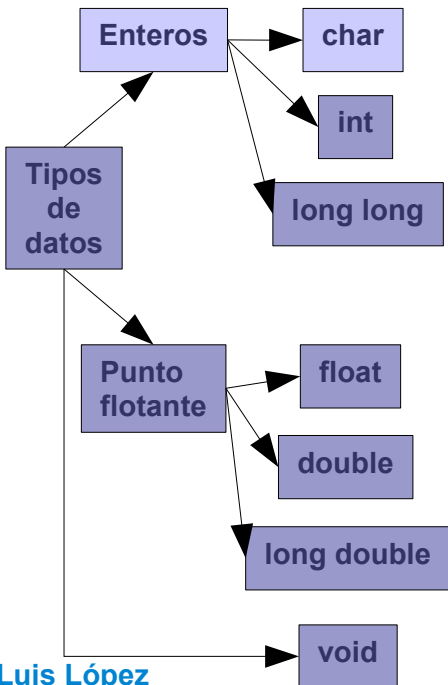
Como se puede observar, en `C` no se han previsto tipos de datos alfanuméricos .

Estos tipos de contenidos (carácteres o cadenas de carácteres) se pueden almacenar en variables o en arrays de tipo `char` .

Lo que en realidad se almacenan son los **números de ASCII** (como números enteros, con los que se puede operar) de los carácteres o cadenas de carácteres .

A los efectos de su manejo (simulando estos tipos alfanuméricos) el compilador dispone de potentes funciones de biblioteca que permiten realizar las operaciones de entrada/salida de cadenas de carácteres, copia de las mismas, búsqueda de carácteres en una cadena (array de enteros `char`), etc. .

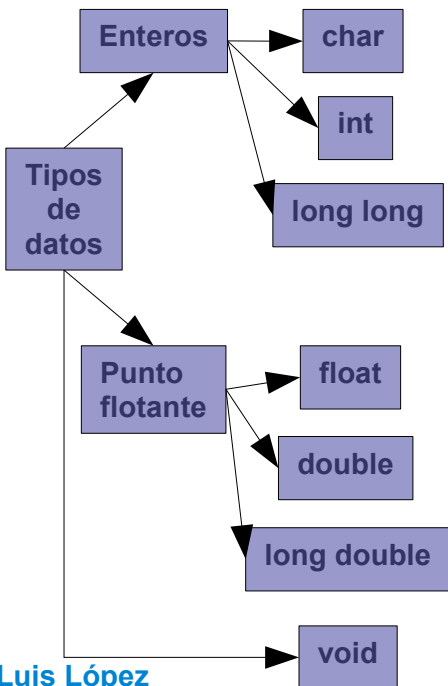
Usted deberá investigar (logrando familiarizarse con ellas), las funciones de biblioteca de `<string.h>` y `<ctype.h>`, además de las funciones de las familias de `'...printf'` y de `'...scanf'` de `<stdio.h>`.



Tipos de datos y tipos de variables propios del Lenguaje `C` .

Los tipos de datos del `C` pertenecen 'casi' exclusivamente a dos grandes grupos .

Ejemplos :



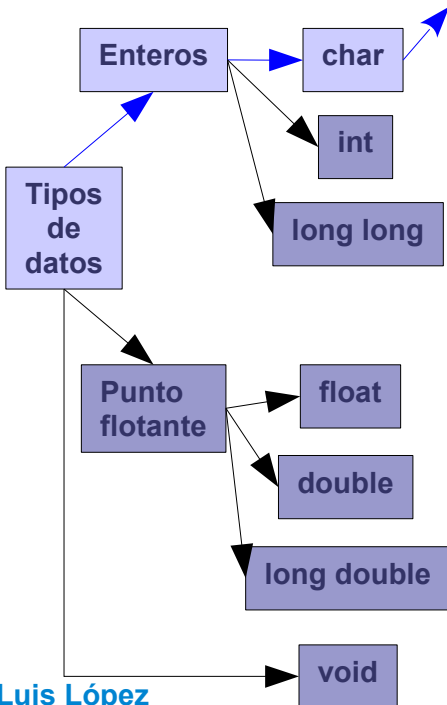
Luis López

Tipos de datos y tipos de variables propios del Lenguaje `C` .

Los tipos de datos del C pertenecen 'casi' exclusivamente a dos grandes grupos .

Ejemplos :

```
char letra;  
unsigned char x;  
signed char val;
```



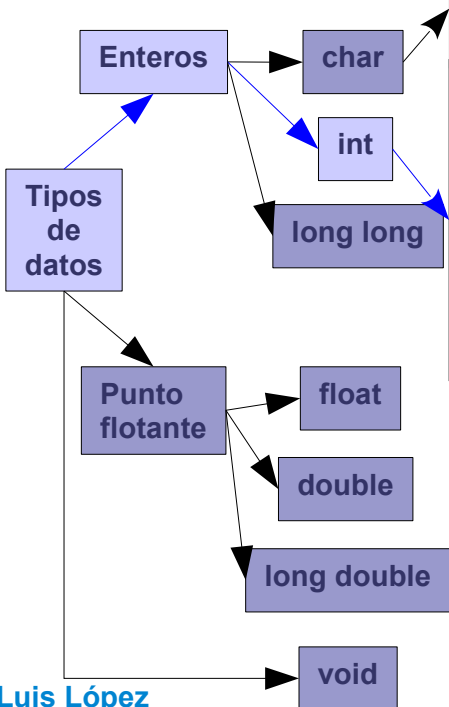
Tipos de datos y tipos de variables propios del Lenguaje C .

Los tipos de datos del C pertenecen 'casi' exclusivamente a dos grandes grupos .

Ejemplos :

```
char letra;  
unsigned char x;  
signed char val;
```

```
int numero;  
signed int y;           /* signed y;           */  
unsigned int x;         /* unsigned x;         */  
short int valor;        /* short valor;        */  
long int cantidad;      /* long cantidad;      */  
signed short int f;     /* signed short f;     */  
signed long int pp;     /* signed long pp;     */  
unsigned short int q;   /* unsigned short q;   */  
unsigned long int ra;   /* unsigned long ra;   */
```



Tipos de datos y tipos de variables propios del Lenguaje C .

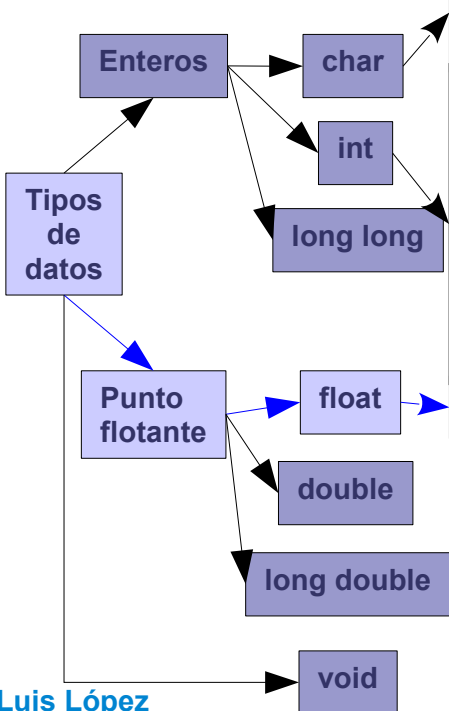
Los tipos de datos del C pertenecen 'casi' exclusivamente a dos grandes grupos .

Ejemplos :

```
char letra;  
unsigned char x;  
signed char val;
```

```
int numero;  
signed int y;           /* signed y;           */  
unsigned int x;         /* unsigned x;         */  
short int valor;        /* short valor;        */  
long int cantidad;      /* long cantidad;      */  
signed short int f;     /* signed short f;     */  
signed long int pp;     /* signed long pp;     */  
unsigned short int q;   /* unsigned short q;   */  
unsigned long int ra;   /* unsigned long ra;   */
```

```
float sueldo;
```



Tipos de datos y tipos de variables propios del Lenguaje C .

Los tipos de datos del C pertenecen 'casi' exclusivamente a dos grandes grupos .

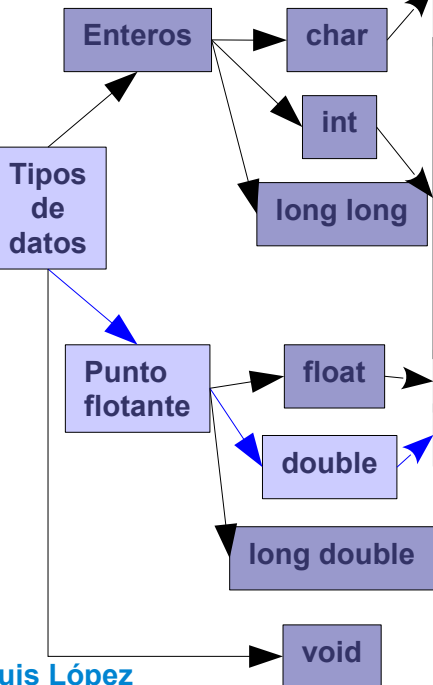
Ejemplos :

```
char letra;  
unsigned char x;  
signed char val;
```

```
int numero;  
signed int y;           /* signed y;           */  
unsigned int x;         /* unsigned x;         */  
short int valor;        /* short valor;        */  
long int cantidad;      /* long cantidad;      */  
signed short int f;      /* signed short f;     */  
signed long int pp;      /* signed long pp;     */  
unsigned short int q;    /* unsigned short q;   */  
unsigned long int ra;    /* unsigned long ra;   */
```

```
float sueldo;
```

```
double valor;
```



Tipos de datos y tipos de variables propios del Lenguaje C .

Los tipos de datos del C pertenecen 'casi' exclusivamente a dos grandes grupos .

Ejemplos :

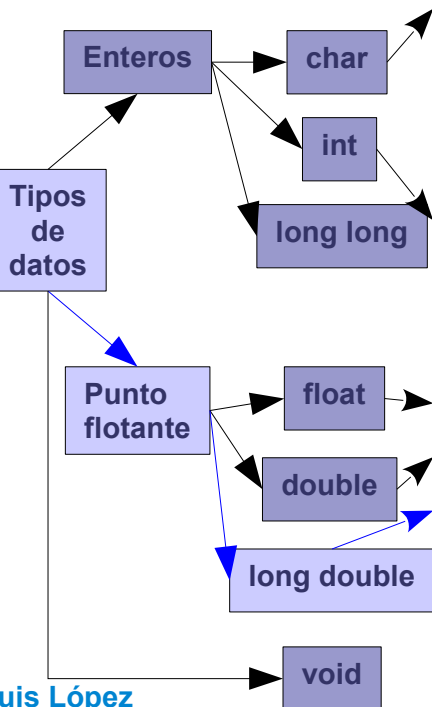
```
char letra;  
unsigned char x;  
signed char val;
```

```
int numero;  
signed int y;           /* signed y;           */  
unsigned int x;         /* unsigned x;         */  
short int valor;        /* short valor;        */  
long int cantidad;      /* long cantidad;      */  
signed short int f;      /* signed short f;     */  
signed long int pp;      /* signed long pp;     */  
unsigned short int q;    /* unsigned short q;   */  
unsigned long int ra;    /* unsigned long ra;   */
```

```
float sueldo;
```

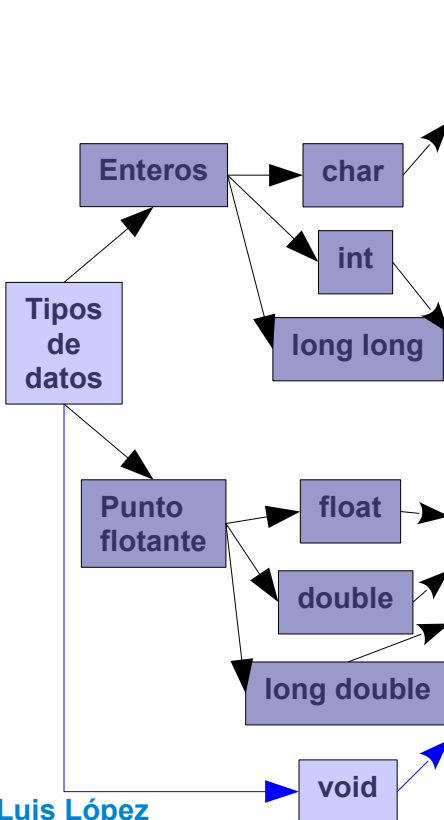
```
double valor;
```

```
long double importe;
```



Tipos de datos y tipos de variables propios del Lenguaje C .

Los tipos de datos del C pertenecen 'casi' exclusivamente a dos grandes grupos .



Ejemplos :

```
char letra;  
unsigned char x;  
signed char val;
```

```
int numero;  
signed int y;           /* signed y;           */  
unsigned int x;         /* unsigned x;         */  
short int valor;        /* short valor;        */  
long int cantidad;      /* long cantidad;      */  
signed short int f;     /* signed short f;     */  
signed long int pp;     /* signed long pp;     */  
unsigned short int q;   /* unsigned short q;   */  
unsigned long int ra;   /* unsigned long ra;   */
```

```
float sueldo;
```

```
double valor;
```

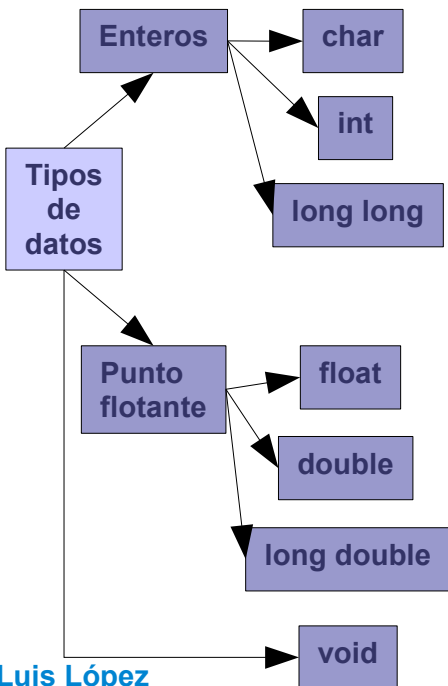
```
long double importe;
```

```
void main(void) /* main no devuelve, */  
{               /* ni recibe valores */  
    void *p = malloc(50); /*se declara un */  
                           /* puntero void */  
}
```

Tipos de datos y tipos de variables propios del Lenguaje C .

Los tipos de datos del `C` pertenecen 'casi' exclusivamente a dos grandes grupos .

Atención :



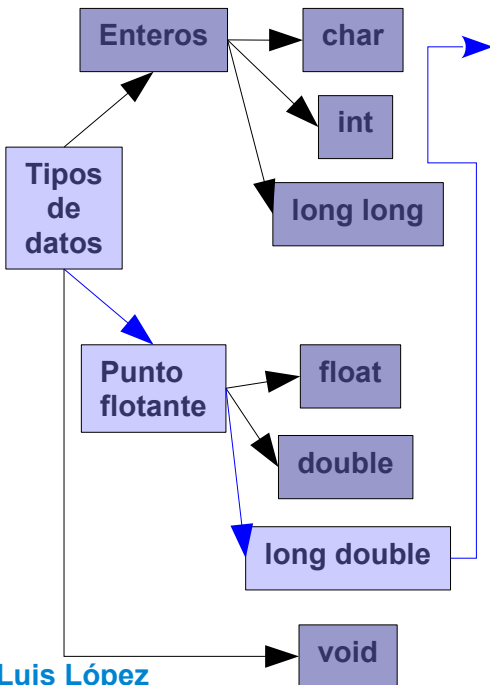
Luis López

Tipos de datos y tipos de variables propios del Lenguaje `C` .

Los tipos de datos del `C` pertenecen 'casi' exclusivamente a dos grandes grupos .

Atención :

No debe confundirse el tipo de dato `long double` (estándar de facto, brindado por la mayoría de los compiladores) con el calificador `long` aplicado a los `double` . Tan sólo es el nombre, compuesto por dos palabras, del tipo de dato .



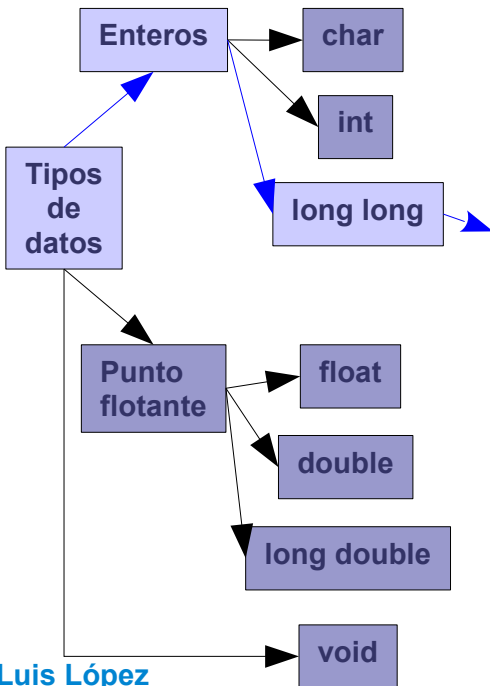
Tipos de datos y tipos de variables propios del Lenguaje `C` .

Los tipos de datos del `C` pertenecen 'casi' exclusivamente a dos grandes grupos .

Atención :

No debe confundirse el tipo de dato `long double` (estándar de facto, brindado por la mayoría de los compiladores) con el calificador `long` aplicado a los `double` . Tan sólo es el nombre, compuesto por dos palabras, del tipo de dato .

Algunos compiladores para plataformas de 64 bits brindan adicionalmente el tipo de dato `long long` que permite generar variables o constantes que se almacenan en 8 Bytes .



Tipos de datos y tipos de variables propios del `Lenguaje C` .

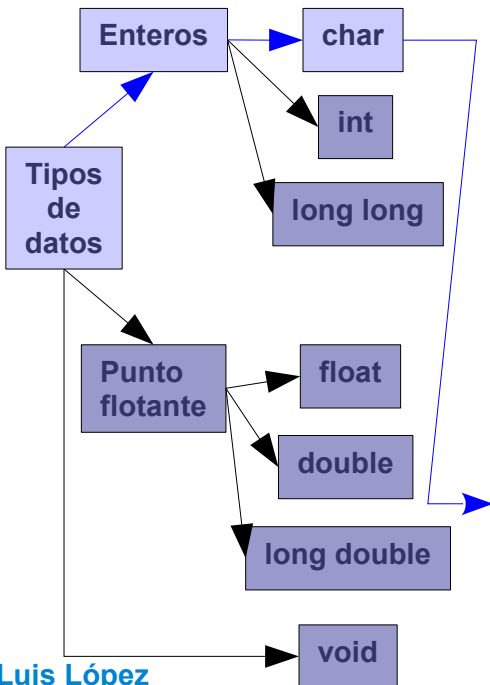
Los tipos de datos del C pertenecen 'casi' exclusivamente a dos grandes grupos .

Atención :

No debe confundirse el tipo de dato `long double` (estándar de facto, brindado por la mayoría de los compiladores) con el calificador `long` aplicado a los `double` . Tan sólo es el nombre, compuesto por dos palabras, del tipo de dato .

Algunos compiladores para plataformas de 64 bits brindan adicionalmente el tipo de dato `long long` que permite generar variables o constantes que se almacenan en 8 Bytes .

Los enteros `char` se almacenan en un solo Byte, por lo que son aptos para almacenar los números de ASCII de caracteres, y en arrays (o vectores de `char`) podemos almacenar cadenas de caracteres (emulando lo que en otros lenguajes son las variables alfanuméricas) . Para el manejo de estos array de `char` emulando variables alfanuméricas el C dispone de funciones de biblioteca al efecto . No se debe olvidar que con las variables `char` siempre se pueden efectuar operaciones aritméticas .



Tipos de datos y tipos de variables propios del Lenguaje C .

Constantes del Lenguaje C .

Se entiende por constantes a aquellas partes de un programa que no cambian su valor a durante la ejecución del programa .

Constantes del Lenguaje C .

Se entiende por constantes a aquellas partes de un programa que no cambian su valor a durante la ejecución del programa .

Constantes literales

Constantes del Lenguaje C .

Se entiende por constantes a aquellas partes de un programa que no cambian su valor a durante la ejecución del programa .

Constantes literales

Las constantes literales son aquellas con las que se declaran dichos valores constantes .

Constantes del Lenguaje C .

Se entiende por constantes a aquellas partes de un programa que no cambian su valor a durante la ejecución del programa .

Constantes literales

Constantes enteras

Las constantes literales son aquellas con las que se declaran dichos valores constantes .

Las constantes literales enteras son por defecto decimales, (p.ej.: 13) a menos que comiencen con cero (p.ej.:015) con lo que resultan estar expresadas en octal, o que comiencen con 0x ó 0X (p.ej.: 0xD) para estar expresadas en hexadecimal .

Constantes del Lenguaje C .

Se entiende por constantes a aquellas partes de un programa que no cambian su valor a durante la ejecución del programa .

Constantes literales

Constantes enteras

Las constantes literales son aquellas con las que se declaran dichos valores constantes .

Las constantes literales enteras son por defecto decimales, (p.ej.: 13) a menos que comiencen con cero (p.ej.:015) con lo que resultan estar expresadas en octal, o que comiencen con 0x ó 0X (p.ej.: 0xD) para estar expresadas en hexadecimal .

Si una constante decimal está seguida por u ó U se pone de manifiesto que es unsigned int .

Constantes del Lenguaje C .

Se entiende por constantes a aquellas partes de un programa que no cambian su valor a durante la ejecución del programa .

Constantes literales

Constantes enteras

Las constantes literales son aquellas con las que se declaran dichos valores constantes .

Las constantes literales enteras son por defecto decimales, (p.ej.: 13) a menos que comiencen con cero (p.ej.:015) con lo que resultan estar expresadas en octal, o que comiencen con 0x ó 0X (p.ej.: 0xD) para estar expresadas en hexadecimal .

Si una constante decimal está seguida por u ó U se pone de manifiesto que es unsigned int .

Si está seguida de l o L, se pone de manifiesto que es long .

Constantes del Lenguaje C .

Se entiende por constantes a aquellas partes de un programa que no cambian su valor a durante la ejecución del programa .

Constantes literales

Constantes enteras

Las constantes literales son aquellas con las que se declaran dichos valores constantes .

Las constantes literales enteras son por defecto decimales, (p.ej.: 13) a menos que comiencen con cero (p.ej.:015) con lo que resultan estar expresadas en octal, o que comiencen con 0x ó 0X (p.ej.: 0xD) para estar expresadas en hexadecimal .

Si una constante decimal está seguida por u ó U se pone de manifiesto que es unsigned int .

Si está seguida de l o L, se pone de manifiesto que es long .

Las constantes char se indican entre apóstrofes (o comillas simples) (p.ej.: 'd') .

Constantes del Lenguaje C .

Se entiende por constantes a aquellas partes de un programa que no cambian su valor a durante la ejecución del programa .

Constantes literales

Constantes enteras

Las constantes literales son aquellas con las que se declaran dichos valores constantes .

Las constantes literales enteras son por defecto decimales, (p.ej.: 13) a menos que comiencen con cero (p.ej.:015) con lo que resultan estar expresadas en octal, o que comiencen con 0x ó 0X (p.ej.: 0xD) para estar expresadas en hexadecimal .

Si una constante decimal está seguida por u ó U se pone de manifiesto que es unsigned int .

Si está seguida de l o L, se pone de manifiesto que es long .

Las constantes char se indican entre apóstrofes (o comillas simples) (p.ej.: 'd') .

Para indicar caracteres especiales se dispone de las llamadas secuencias de escape :

Constantes del Lenguaje C .

Se entiende por constantes a aquellas partes de un programa que no cambian su valor a durante la ejecución del programa .

Constantes literales

Constantes enteras

Secuencias de escape

Las constantes literales son aquellas con las que se declaran dichos valores constantes .

Las constantes literales enteras son por defecto decimales, (p.ej.: 13) a menos que comiencen con cero (p.ej.:015) con lo que resultan estar expresadas en octal, o que comiencen con 0x ó 0X (p.ej.: 0xD) para estar expresadas en hexadecimal .

Si una constante decimal está seguida por u ó U se pone de manifiesto que es unsigned int .

Si está seguida de l o L, se pone de manifiesto que es long .

Las constantes char se indican entre apóstrofes (o comillas simples) (p.ej.: 'd') .

Para indicar caracteres especiales se dispone de las llamadas secuencias de escape :

Constantes del Lenguaje C .

Se entiende por constantes a aquellas partes de un programa que no cambian su valor a durante la ejecución de

Constantes literales

Constantes
enteras

Secuencias
de escape

Las
decla

Las
(p.ej.
lo qu
con
hexa

Si un
man

Si es

Las
simp

Denominación	Secuen- cia de escape	Núme- ro de ASCII
carácter nulo	\0	0
beep (audible alert)	\a	7
retroceso (backspace)	\b	8
tab horizontal	\t	9
nueva línea (newline)	\n	10
tab vertical	\v	11
salto de página (ff)	\f	12
retorno de carro (cr)	\r	13
contrabarra	\\	92
apóstrofe	\'	39
comillas	\"	34
número octal	\0oo	
número hexadecimal	\xhh	

Para indicar caracteres especiales se dispone de las llamadas secuencias de escape :

Constantes del Lenguaje C .

Se entiende por constantes a aquellas partes de un programa que no cambian su valor a durante la ejecución del programa .

Constantes literales

Constantes enteras

Secuencias de escape

Constantes en punto flotante

Las constantes literales son aquellas con las que se declaran dichos valores constantes .

Las constantes literales enteras son por defecto decimales, (p.ej.: 13) a menos que comiencen con cero (p.ej.:015) con lo que resultan estar expresadas en octal, o que comiencen con 0x ó 0X (p.ej.: 0xD) para estar expresadas en hexadecimal .

Las constantes literales en punto flotante se expresan en decimal con signo, opcionalmente seguido por un exponente entero decimal especificado con e o E . Además puede estar seguido por f , F, l ó L indicando que es float (f, F) o long double (l o L), si no se especifica es double .

Constantes del Lenguaje C .

Se entiende por constantes a aquellas partes de un programa que no cambian su valor a durante la ejecución del programa .

Constantes literales

Constantes enteras

Secuencias de escape

Constantes en punto flotante

Las constantes literales son aquellas con las que se declaran dichos valores constantes .

Las constantes literales enteras son por defecto decimales, (p.ej.: 13) a menos que comiencen con cero (p.ej.:015) con lo que resultan estar expresadas en octal, o que comiencen con 0x ó 0X (p.ej.: 0xD) para estar expresadas en hexadecimal .

Las constantes literales en punto flotante se expresan en decimal con signo, opcionalmente seguido por un exponente entero decimal especificado con e o E . Además puede estar seguido por f , F, l ó L indicando que es float (f, F) o long double (l o L), si no se especifica es double .

valor	float	double	long double
2	2f	2.	2.L
2.03	2.03f	2.03	2.03L
0.125	.125f	.125	.125L
12.5×10^{-2}	12.5e-2f	12.5e-2	12.5e-2L
1.25×10^{-1}	1.25e-1f	1.25e-1	1.25e-1L

Constantes del Lenguaje C .

Constantes del Lenguaje C .

Se entiende por constantes a aquellas partes de un programa que no cambian su valor a durante la ejecución del programa .

Constantes literales

Constantes enteras

Secuencias de escape

Constantes en punto flotante

Constantes de caracteres

Las constantes literales son aquellas con las que se declaran dichos valores constantes .

Las constantes literales enteras son por defecto decimales, (p.ej.: 13) a menos que comiencen con cero (p.ej.:015) con lo que resultan estar expresadas en octal, o que comiencen con 0x ó 0X (p.ej.: 0xD) para estar expresadas en hexadecimal .

Las constantes literales en punto flotante se expresan en decimal con signo, opcionalmente seguido por un exponente entero decimal especificado con e o E . Además puede estar seguido por f , F, l ó L indicando que es float (f, F) o long double (l o L), si no se especifica es double .

Las constantes literales de cadena de caracteres se representan encerradas entre comillas . Internamente se almacena un array de enteros char conteniendo los números de ASCII de los caracteres, y a continuación se almacena siempre un carácter nulo (p.ej.: "Hola" es un arreglo de 5 bytes, en el último contiene el número 0).

Constantes del Lenguaje C .

Constantes del Lenguaje C .

Las constantes literales se emplean para inicializar las **variables** y **constantes** del programa, y para controlar sus ciclos repetitivos .

Constantes del Lenguaje C .

Las constantes literales se emplean para inicializar las variables y constantes del programa, y para controlar sus ciclos repetitivos .

```
char      let_1    = 'A';
char      let_2    = 65;
char      nom_1[]  = { "Ana" };
char      nom_2[]  = { 'A', 'n', 'a', '\0' };
char      nom_3[]  = { 0x41, 110, 0141, 0 };
char      nom_4[]  = { '\x41', '\x6E', '\x61', '\0' };
char      ojo[3]   = { "Ana" }; /*no termina con nulo*/
int       num_1    = 1234;
long      num_2    = 0xffffffffL;
unsigned long num_3 = 0xffffffffuL;
float     sue_1    = 12345.67f;
float     sue_2    = 12345.9f;
double    sue_3    = 12345.67;
```


Constantes del Lenguaje C .

Las constantes literales se emplean para inicializar las variables y constantes del programa, y para controlar sus ciclos repetitivos .

```
char      let_1    = 'A';
char      let_2    = 65;
char      nom_1[]  = { "Ana" };
char      nom_2[]  = { 'A', 'n', 'a', '\0' };
char      nom_3[]  = { 0x41, 110, 0141, 0 };
char      nom_4[]  = { '\x41', '\x6E', '\x61', '\0' };
char      ojo[3]   = { "Ana" }; /*no termina con nulo*/
int       num_1    = 1234;
long      num_2    = 0xffffffffL;
unsigned long num_3 = 0xffffffffuL;
float     sue_1    = 12345.67f;
float     sue_2    = 12345.9f;
double    sue_3    = 12345.67;
```

Acá se han inicializado diversas variables con distintos valores y modos de inicialización .

Constantes del Lenguaje C .

Las constantes literales se emplean para inicializar las variables y constantes del programa, y para controlar sus ciclos repetitivos .

```
char      let_1    = 'A';
char      let_2    = 65;
char      nom_1[]  = { "Ana" };
char      nom_2[]  = { 'A', 'n', 'a', '\0' };
char      nom_3[]  = { 0x41, 110, 0141, 0 };
char      nom_4[]  = { '\x41', '\x6E', '\x61', '\0' };
char      ojo[3]   = { "Ana" }; /*no termina con nulo*/
int       num_1    = 1234;
long      num_2    = 0xffffffffL;
unsigned long num_3 = 0xffffffffuL;
float     sue_1    = 12345.67f;
float     sue_2    = 12345.9f;
double    sue_3    = 12345.67;
```

Acá se han inicializado diversas variables con distintos valores y modos de inicialización .

Al ejecutar, 'inspeccionando' las variables del programa, se ven los valores asignados a las mismas .

Constantes del Lenguaje C .

Las constantes literales se emplean para inicializar las variables y constantes del programa, y para controlar sus ciclos repetitivos .

```
char    let_1    = 'A';
char    let_2    = 65;
char    nom_1[]  = { "Ana" };
char    nom_2[]  = { 'A', 'n', 'a', '\0' };
char    nom_3[]  = { 0x41, 110, 0141, 0 };
char    nom_4[]  = { '\x41', '\x6E', '\x61', '\0' };
char    ojo[3]   = { "Ana" }; /*no termina con nulo*/
int      num_1    = 1234;
long     num_2    = 0xffffffffL;
unsigned long num_3 = 0xffffffffuL;
float    sue_1    = 12345.67f;
float    sue_2    = 12345.9f;
double   sue_3    = 12345.67;
```

Acá se han inicializado diversas variables con distintos valores y modos de inicialización .

Al ejecutar, 'inspeccionando' las variables del programa, se ven los valores asignados a las mismas .

Name	Value
let_1	65 'A'
let_2	65 'A'
⊕ nom_1	0x0012ff74 "Ana"
⊕ nom_2	0x0012ff70 "Ana"
⊕ nom_3	0x0012ff6c "Ana"
⊕ nom_4	0x0012ff68 "Ana"
⊕ ojo	0x0012ff64 "AnaAna"
num_1	1234
num_2	-1
num_3	4294967295
sue_1	12345.7
sue_2	12345.9
sue_3	12345.670000000

Constantes del Lenguaje C .

Constantes del Lenguaje C .

Las constantes literales se emplean para inicializar las variables y constantes del programa, y para controlar sus ciclos repetitivos .

```
const char      let_1   = 'A';
const char      let_2   = 65;
const char      nom_1[] = { "Ana" };
const char      nom_2[] = { 'A', 'n', 'a', '\0' };
const char      nom_3[] = { 0x41, 110, 0141, 0 };
const char      nom_4[] = { '\x41', '\x6E', '\x61', '\0' };
const char      ojo[3]  = { "Ana" }; /*no termina con nulo*/
const int       num_1   = 1234;
const long      num_2   = 0xffffffffL;
const unsigned long num_3 = 0xffffffffuL;
const float     sue_1   = 12345.67f;
const float     sue_2   = 12345.9f;
const double    sue_3   = 12345.67;
```

Acá se han inicializado diversas constantes con los mismos valores y modos de inicialización .

Constantes del Lenguaje C .

Las constantes literales se emplean para inicializar las variables y constantes del programa, y para controlar sus ciclos repetitivos .

```
const char      let_1   = 'A';
const char      let_2   = 65;
const char      nom_1[] = { "Ana" };
const char      nom_2[] = { 'A', 'n', 'a', '\0' };
const char      nom_3[] = { 0x41, 110, 0141, 0 };
const char      nom_4[] = { '\x41', '\x6E', '\x61', '\0' };
const char      ojo[3]  = { "Ana" }; /*no termina con nulo*/
const int       num_1   = 1234;
const long      num_2   = 0xffffffffL;
const unsigned long num_3 = 0xffffffffuL;
const float     sue_1   = 12345.67f;
const float     sue_2   = 12345.9f;
const double    sue_3   = 12345.67;
```

Acá se han inicializado diversas constantes con los mismos valores y modos de inicialización .

La declaración de una constante comienza con la palabra clave `const` y su valor no podrá ser alterado durante la ejecución del programa .

Un macroreemplazo es una directiva para el preprocesador del c .

Un macroreemplazo es una directiva para el preprocesador del C .

Cuando se procede a compilar un programa C, lo primero que se ejecuta es el preprocesador de C .

Un macroreemplazo es una directiva para el preprocesador del C .

Cuando se procede a compilar un programa C, lo primero que se ejecuta es el preprocesador de C .

Empleando la directiva `#define` para el preprocesador de C, se pueden declarar lo que algunos autores denominan (a mi juicio, semánticamente mal) como 'constantes simbólicas' (ya que lo que se declaran son macroreemplazos) .

Un macroreemplazo es una directiva para el preprocesador del C .

Cuando se procede a compilar un programa C, lo primero que se ejecuta es el preprocesador de C .

Empleando la directiva `#define` para el preprocesador de C, se pueden declarar lo que algunos autores denominan (a mi juicio, semánticamente mal) como 'constantes simbólicas' (ya que lo que se declaran son macroreemplazos) .

Esta función del preprocesador de hacer los macroreemplazos, la cumple 'mostrándole' al compilador que en todos los lugares donde figure la etiqueta declarada con `#define`, este asuma el valor asociado a la misma (o sea, el indicado a continuación de la etiqueta) .

Un macroreemplazo es una directiva para el preprocesador del C .

Cuando se procede a compilar un programa C, lo primero que se ejecuta es el preprocesador de C .

Empleando la directiva `#define` para el preprocesador de C, se pueden declarar lo que algunos autores denominan (a mi juicio, semánticamente mal) como 'constantes simbólicas' (ya que lo que se declaran son macroreemplazos) .

Esta función del preprocesador de hacer los macroreemplazos, la cumple 'mostrándole' al compilador que en todos los lugares donde figure la etiqueta declarada con `#define`, este asuma el valor asociado a la misma (o sea, el indicado a continuación de la etiqueta) .

El código que sigue	es equivalente a este
<pre>#define TAM 30 void main(void) { int vector[TAM]; for(x = 0; x < TAM; x++) vector[x] = 0; printf("Ingrese %d enteros\n" TAM) ; for(x = 0; x < TAM; x++) scanf("%d", &vector[x]); printf("Los %d valores son.\n", TAM) ; for(x = 0; x < TAM; x++) printf("%d ", vector[x]); }</pre>	<pre>void main(void) { int vector[30]; for(x = 0; x < 30; x++) vector[x] = 0; printf("Ingrese 30 enteros\n"); for(x = 0; x < 30; x++) scanf("%d", &vector[x]); printf("Los 30 valores son.\n"); for(x = 0; x < 30; x++) printf("%d ", vector[x]); }</pre>

Macroreemplazos 1 (`#define` sencillos o simples)

Un macroreemplazo es una directiva para el preprocesador del C .

Cuando se procede a compilar un programa C, lo primero que se ejecuta es el preprocesador de C .

Empleando la directiva `#define` para el preprocesador de C, se pueden declarar lo que algunos autores denominan (a mi juicio, semánticamente mal) como 'constantes simbólicas' (ya que lo que se declaran son macroreemplazos) .

Pero, la gran ventaja que tiene el uso de estos macroreemplazos, es que si en vez de un vector de 30 enteros se necesitara uno de 28, con modificar el programa en un solo lugar, este seguirá trabajando, tras compilarlo, con la nueva cantidad .

El código que sigue

```
#define TAM      30
void main(void)
{
    int vector[TAM];
    for(x = 0; x < TAM; x++)
        vector[x] = 0;
    printf("Ingrese %d enteros\n"
           TAM) ;
    for(x = 0; x < TAM; x++)
        scanf("%d", &vector[x]);
    printf("Los %d valores son.\n",
           TAM) ;
    for(x = 0; x < TAM; x++)
        printf("%d ", vector[x]);
}
```

es equivalente a este

```
void main(void)
{
    int vector[30];
    for(x = 0; x < 30; x++)
        vector[x] = 0;
    printf("Ingrese 30 enteros\n");
    for(x = 0; x < 30; x++)
        scanf("%d", &vector[x]);
    printf("Los 30 valores son.\n");
    for(x = 0; x < 30; x++)
        printf("%d ", vector[x]);
}
```

Macroreemplazos 1 (`#define` sencillos o simples)

En C el programador puede declarar estructuras y tipos de datos que no existen en el lenguaje .

Si el programador necesita declarar un tipo de dato que contenga, p. ej., un número de DNI, un apellido y nombres y un importe lo puede hacer de dos modos :

En C el programador puede declarar estructuras y tipos de datos que no existen en el lenguaje .

Si el programador necesita declarar un tipo de dato que contenga, p. ej., un número de DNI, un apellido y nombres y un importe lo puede hacer de dos modos :

```
struct s_pers
{
    long dni;
    char apyn[36];
    float importe;
};
```

En C el programador puede declarar estructuras y tipos de datos que no existen en el lenguaje .

Si el programador necesita declarar un tipo de dato que contenga, p. ej., un número de DNI, un apellido y nombres y un importe lo puede hacer de dos modos :

```
struct s_pers
{
    long dni;
    char apyn[36];
    float importe;
};
```

```
typedef struct s_pers
{
    long dni;
    char apyn[36];
    float importe;
} t_pers;
```


En C el programador puede declarar estructuras y tipos de datos que no existen en el lenguaje .

Si el programador necesita declarar un tipo de dato que contenga, p. ej., un número de DNI, un apellido y nombres y un importe lo puede hacer de dos modos :

```
struct s_pers
{
    long dni;
    char apyn[36];
    float importe;
};
```

```
typedef struct s_pers
{
    long dni;
    char apyn[36];
    float importe;
} t_pers;
```

Con `struct s_pers` se pueden declarar variables `struct` a conveniencia del programador

En C el programador puede declarar estructuras y tipos de datos que no existen en el lenguaje .

Si el programador necesita declarar un tipo de dato que contenga, p. ej., un número de DNI, un apellido y nombres y un importe lo puede hacer de dos modos :

```
struct s_pers
{
    long dni;
    char apyn[36];
    float importe;
};
```

```
typedef struct s_pers
{
    long dni;
    char apyn[36];
    float importe;
} t_pers;
```

Con struct s_pers se pueden declarar variables struct a conveniencia del programador

```
struct s_pers afi;
```

En C el programador puede declarar estructuras y tipos de datos que no existen en el lenguaje .

Si el programador necesita declarar un tipo de dato que contenga, p. ej., un número de DNI, un apellido y nombres y un importe lo puede hacer de dos modos :

```
struct s_pers
{
    long dni;
    char apyn[36];
    float importe;
};
```

Con struct s_pers se pueden declarar variables struct a conveniencia del programador

```
struct s_pers afi;
```

```
typedef struct s_pers
{
    long dni;
    char apyn[36];
    float importe;
} t_pers;
```

Con struct s_pers acá también se pueden declarar variables struct a conveniencia del programador

En C el programador puede declarar estructuras y tipos de datos que no existen en el lenguaje .

Si el programador necesita declarar un tipo de dato que contenga, p. ej., un número de DNI, un apellido y nombres y un importe lo puede hacer de dos modos :

```
struct s_pers
{
    long dni;
    char apyn[36];
    float importe;
};
```

Con struct s_pers se pueden declarar variables struct a conveniencia del programador

```
struct s_pers afi;
```

```
typedef struct s_pers
{
    long dni;
    char apyn[36];
    float importe;
} t_pers;
```

Con struct s_pers acá también se pueden declarar variables struct a conveniencia del programador

```
struct s_pers afi;
```

En C el programador puede declarar estructuras y tipos de datos que no existen en el lenguaje .

Si el programador necesita declarar un tipo de dato que contenga, p. ej., un número de DNI, un apellido y nombres y un importe lo puede hacer de dos modos :

```
struct s_pers
{
    long dni;
    char apyn[36];
    float importe;
};
```

Con struct s_pers se pueden declarar variables struct a conveniencia del programador

```
struct s_pers afi;
```

En C el uso de las palabras clave struct es obligatorio para declarar variables

```
typedef struct s_pers
{
    long dni;
    char apyn[36];
    float importe;
} t_pers;
```

Con struct s_pers acá también se pueden declarar variables struct a conveniencia del programador

```
struct s_pers afi;
```

En C el programador puede declarar estructuras y tipos de datos que no existen en el lenguaje .

Si el programador necesita declarar un tipo de dato que contenga, p. ej., un número de DNI, un apellido y nombres y un importe lo puede hacer de dos modos :

```
struct s_pers
{
    long dni;
    char apyn[36];
    float importe;
};
```

Con struct s_pers se pueden declarar variables struct a conveniencia del programador

```
struct s_pers afi;
```

En C el uso de las palabras clave struct es obligatorio para declarar variables

```
typedef struct s_pers
{
    long dni;
    char apyn[36];
    float importe;
} t_pers;
```

Con struct s_pers acá también se pueden declarar variables struct a conveniencia del programador

```
struct s_pers afi;
```

o mejor aún, con t_pers se pueden declarar variables idénticas a las anteriores, sin necesidad de la palabra clave struct

En C el programador puede declarar estructuras y tipos de datos que no existen en el lenguaje .

Si el programador necesita declarar un tipo de dato que contenga, p. ej., un número de DNI, un apellido y nombres y un importe lo puede hacer de dos modos :

```
struct s_pers
{
    long dni;
    char apyn[36];
    float importe;
};
```

Con struct s_pers se pueden declarar variables struct a conveniencia del programador

```
struct s_pers afi;
```

En C el uso de las palabras clave struct es obligatorio para declarar variables

```
typedef struct s_pers
{
    long dni;
    char apyn[36];
    float importe;
} t_pers;
```

Con struct s_pers acá también se pueden declarar variables struct a conveniencia del programador

```
struct s_pers afi;
```

o mejor aún, con t_pers se pueden declarar variables idénticas a las anteriores, sin necesidad de la palabra clave struct

```
t_pers afi;
```

En C el programador puede declarar estructuras y tipos de datos que no existen en el lenguaje .

Si el programador necesita declarar un tipo de dato que contenga, p. ej., un número de DNI, un apellido y nombres y un importe lo puede hacer de dos modos :

```
struct s_pers
{
    long dni;
    char apyn[36];
    float importe;
};
```

Con struct s_pers se pueden declarar variables struct a conveniencia del programador

```
struct s_pers afi;
```

En C el uso de las palabras clave struct es obligatorio para declarar variables

una u otra, no ambas, preferentemente la segunda

```
typedef struct s_pers
{
    long dni;
    char apyn[36];
    float importe;
} t_pers;
```

Con struct s_pers acá también se pueden declarar variables struct a conveniencia del programador

```
struct s_pers afi;
```

o mejor aún, con t_pers se pueden declarar variables idénticas a las anteriores, sin necesidad de la palabra clave struct

```
t_pers afi;
```


En C el programador puede declarar estructuras y tipos de datos que no existen en el lenguaje .

Si el programador necesita declarar un tipo de dato que contenga, p. ej., un número de DNI, un apellido y nombres y un importe lo puede hacer de dos modos :

```
struct s_pers
{
    long dni;
    char apyn[36];
    float importe;
};
```

Con struct s_pers se pueden declarar variables struct a conveniencia del programador

```
struct s_pers afi;
```

En C el uso de las palabras clave struct es obligatorio para declarar variables

una u otra, no ambas, preferentemente la segunda

```
typedef struct s_pers
{
    long dni;
    char apyn[36];
    float importe;
} t_pers;
```

Con struct s_pers acá también se pueden declarar variables struct a conveniencia del programador

```
struct s_pers afi;
```

o mejor aún, con t_pers se pueden declarar variables idénticas a las anteriores, sin necesidad de la palabra clave struct

```
t_pers afi;
```

en la declaración typedef se puede omitir el identificador de estructura s_pers

Tipos de datos, struct y typedef .

La declaración de tipo `typedef` además nos permite declaraciones como :

La declaración de tipo `typedef` además nos permite declaraciones como :

```
typedef int BOOL;
```

Con lo que estamos definiendo un nuevo tipo de dato `BOOL` (que en el fondo es un `int`) .

La declaración de tipo `typedef` además nos permite declaraciones como :

```
typedef int BOOL;
```

```
const int FALSE = 0;  
const int TRUE  = 1;
```

Con lo que estamos definiendo un nuevo tipo de dato `BOOL` (que en el fondo es un `int`) .

Si junto con esta declaración además declaramos :

La declaración de tipo `typedef` además nos permite declaraciones como :

```
typedef int BOOL;
```

```
const int FALSE = 0;  
const int TRUE  = 1;
```

```
#define FALSE    0  
#define TRUE     1
```

Con lo que estamos definiendo un nuevo tipo de dato `BOOL` (que en el fondo es un `int`) .

Si junto con esta declaración además declaramos :

o en su defecto

La declaración de tipo `typedef` además nos permite declaraciones como :

```
typedef int BOOL;
```

```
/* const int FALSE = 0; */  
/* const int TRUE  = 1; */
```

```
#define FALSE    0  
#define TRUE     1
```

Con lo que estamos definiendo un nuevo tipo de dato `BOOL` (que en el fondo es un `int`) .

Si junto con esta declaración además declaramos :

o en su defecto (una u otra)

La declaración de tipo `typedef` además nos permite declaraciones como :

```
typedef int BOOL;
```

```
/* const int FALSE = 0; */  
/* const int TRUE  = 1; */
```

```
#define FALSE    0  
#define TRUE     1
```

```
BOOL valor;    /* crea una var.  */
```

Con lo que estamos definiendo un nuevo tipo de dato `BOOL` (que en el fondo es un `int`) .

Si junto con esta declaración además declaramos :

o en su defecto (una u otra)

nos permite declarar variables

La declaración de tipo `typedef` además nos permite declaraciones como :

```
typedef int BOOL;
```

```
/* const int FALSE = 0; */  
/* const int TRUE  = 1; */
```

```
#define FALSE    0  
#define TRUE     1
```

Con lo que estamos definiendo un nuevo tipo de dato `BOOL` (que en el fondo es un `int`).

Si junto con esta declaración además declaramos :

o en su defecto (una u otra)

nos permite declarar variables, sus inicializaciones

```
BOOL valor; /* crea una var. */  
Valor = TRUE; /* le asigna TRUE */
```

La declaración de tipo `typedef` además nos permite declaraciones como :

```
typedef int BOOL;
```

```
/* const int FALSE = 0; */  
/* const int TRUE  = 1; */
```

```
#define FALSE    0  
#define TRUE     1
```

Con lo que estamos definiendo un nuevo tipo de dato `BOOL` (que en el fondo es un `int`) .

Si junto con esta declaración además declaramos :

o en su defecto (una u otra)

nos permite declarar variables, sus inicializaciones

```
void main(void)  
{  
    BOOL valor; /* crea una var. */  
    Valor = TRUE; /* le asigna TRUE */  
    if(func() != Valor)  
        puts("La función devolvió falso");  
    /* ... sigue ... */  
}
```

La declaración de tipo `typedef` además nos permite declaraciones como :

```
typedef int BOOL;
```

```
/* const int FALSE = 0; */  
/* const int TRUE  = 1; */
```

```
#define FALSE    0  
#define TRUE     1
```

```
TRUE;  
FALSE;
```

```
void main(void)  
{  
    BOOL valor; /* crea una var. */  
    Valor = TRUE; /* le asigna TRUE */  
    if(func() != Valor)  
        puts("La función devolvió falso");  
    /* ... sigue ... */  
}
```

Con lo que estamos definiendo un nuevo tipo de dato `BOOL` (que en el fondo es un `int`).

Si junto con esta declaración además declaramos :

o en su defecto (una u otra)

nos permite declarar variables, sus inicializaciones, valores

La declaración de tipo `typedef` además nos permite declaraciones como :

```
typedef int BOOL;
```

```
/* const int FALSE = 0; */  
/* const int TRUE  = 1; */
```

```
#define FALSE    0  
#define TRUE     1
```

```
BOOL func()
```

```
TRUE;
```

```
FALSE;
```

```
void main(void)  
{  
    BOOL Valor; /* crea una var. */  
    Valor = TRUE; /* le asigna TRUE */  
    if(func() != Valor)  
        puts("La función devolvió falso");  
    /* ... sigue ... */  
}
```

Con lo que estamos definiendo un nuevo tipo de dato `BOOL` (que en el fondo es un `int`).

Si junto con esta declaración además declaramos :

o en su defecto (una u otra)

nos permite declarar variables, sus inicializaciones, valores y tipos de dato de retorno de funciones

La declaración de tipo `typedef` además nos permite declaraciones como :

```
typedef int BOOL;
```

```
/* const int FALSE = 0; */  
/* const int TRUE  = 1; */
```

```
#define FALSE    0  
#define TRUE     1
```

```
BOOL func()
```

```
{  
    if( /* <alguna condición> */ )  
        return TRUE;  
    return FALSE;  
}
```

```
void main(void)
```

```
{  
    BOOL Valor; /* crea una var. */  
    Valor = TRUE; /* le asigna TRUE */  
    if(func() != Valor)  
        puts("La función devolvió falso");  
    /* ... sigue ... */  
}
```

Con lo que estamos definiendo un nuevo tipo de dato `BOOL` (que en el fondo es un `int`).

Si junto con esta declaración además declaramos :

o en su defecto (una u otra)

nos permite declarar variables, sus inicializaciones, valores y tipos de dato de retorno de funciones

La declaración de tipo `typedef` además nos permite declaraciones como :

```
typedef int BOOL;
```

```
/* const int FALSE = 0; */  
/* const int TRUE  = 1; */
```

```
#define FALSE    0  
#define TRUE     1
```

```
/* funcion booleana que devuelve  
Verdad o Falso*/
```

```
BOOL func()
```

```
{  
    if( /* <alguna condición> */ )  
        return TRUE;  
    return FALSE;  
}
```

```
void main(void)
```

```
{  
    BOOL Valor; /* crea una var. */  
    Valor = TRUE; /* le asigna TRUE */  
    if(func() != Valor)  
        puts("La función devolvió falso");  
    /* ... sigue ... */  
}
```

Con lo que estamos definiendo un nuevo tipo de dato `BOOL` (que en el fondo es un `int`).

Si junto con esta declaración además declaramos :

o en su defecto (una u otra)

nos permite declarar variables, sus inicializaciones, valores y tipos de dato de retorno de funciones con sentido lógico que admitan los valores de verdad o falso .

Mediante las declaraciones de tipos `enum` podemos escribir en una forma mucho más compacta el ejemplo anterior, donde quedan mucho más claro y conciso lo expresado en los tres renglones de declaraciones antes requeridas :

Mediante las declaraciones de tipos `enum` podemos escribir en una forma mucho más compacta el ejemplo anterior, donde quedan mucho más claro y conciso lo expresado en los tres renglones de declaraciones antes requeridas :

```
typedef int BOOL;
```

```
/* const int FALSE = 0; */  
/* const int TRUE  = 1; */
```

```
#define FALSE    0  
#define TRUE     1
```

```
enum BOOL { FALSE, TRUE };
```

```
graph LR; A["enum BOOL { FALSE, TRUE };"] --> B["typedef int BOOL;"]; A --> C["/* const int FALSE = 0; */  
/* const int TRUE  = 1; */"]; A --> D["#define FALSE    0  
#define TRUE     1"];
```

las declaraciones `enum` permiten reemplazar los tipos declarados con `typedef` y las etiquetas (macroreemplazos) declarados con `#define`

Tipos de datos `enum` .

Mediante las declaraciones de tipos `enum` podemos escribir en una forma mucho más compacta el ejemplo anterior, donde quedan mucho más claro y conciso lo expresado en los tres renglones de declaraciones antes requeridas :

```
typedef int BOOL;
```

```
/* const int FALSE = 0; */  
/* const int TRUE  = 1; */
```

```
#define FALSE    0  
#define TRUE     1
```

```
/* funcion booleana que devuelve  
Verdad o Falso*/
```

```
BOOL func()  
{
```

```
    if( /* <alguna condición> */ )  
        return TRUE;  
    return FALSE;  
}
```

```
void main(void)  
{
```

```
    BOOL valor; /* crea una va  
    Valor = TRUE; /* le asigna T  
    if(func() != Valor)  
        puts("La función devolvió  
    /* ... sigue ... */
```

```
enum BOOL { FALSE, TRUE };
```

```
/* funcion booleana que devuelve  
Verdad o Falso*/
```

```
enum BOOL func()  
{
```

```
    if( /* <alguna condición> */ )  
        return TRUE;  
    return FALSE;  
}
```

```
void main(void)  
{
```

```
    enum BOOL Valor; /* crea una var. */  
    Valor = TRUE; /* le asigna TRUE */  
    if(func() != Valor)  
        puts("La función devolvió falso");  
    /* ... sigue ... */
```

pero como las enumeraciones no declaran tipos como lo hace `typedef`, para declarar variables o tipos de valores devueltos por funciones, en C, se debe emplear la palabra clave (reservada) `enum` seguida por el nombre de la enumeración

Tipos de datos `enum` .

Mediante las declaraciones de tipos `enum` podemos escribir en una forma mucho más compacta el ejemplo anterior, donde quedan mucho más claro y conciso lo expresado en los tres renglones de declaraciones antes requeridas :

```
typedef int BOOL;
```

```
/* const int FALSE = 0; */  
/* const int TRUE  = 1; */
```

```
#define FALSE    0  
#define TRUE     1
```

```
/* funcion booleana que devuelve  
Verdad o Falso*/
```

```
BOOL func()
```

```
{  
    if( /* <alguna condición> */ )  
        return TRUE;  
    return FALSE;  
}
```

```
void main(void)
```

```
{  
    BOOL valor; /* crea una var.  
    Valor = TRUE; /* le asigna T  
    if(func() != Valor)  
        puts("La función devolvió  
    /* ... sigue ... */  
}
```

```
enum BOOL { FALSE, TRUE };
```

```
/* funcion booleana que devuelve  
Verdad o Falso*/
```

```
enum BOOL func()
```

```
{  
    if( /* <alguna condición> */ )  
        return TRUE;  
    return FALSE;  
}
```

```
void main(void)
```

```
{  
    enum BOOL Valor; /* crea una var. */  
    Valor = TRUE; /* le asigna TRUE */  
    if(func() != Valor)  
        puts("La función devolvió falso");  
    /* ... sigue ... */  
}
```

las etiquetas (en este caso FALSE y TRUE) declarados por la enum se utilizan igual que antes

Se debe notar que por defecto a la primer etiqueta (**FALSE** en nuestro ejemplo), nuestra declaración **enum** la inicializa con 0, a la segunda (**TRUE**) con 1, y así sucesivamente (si hubiera más) hasta la última .

```
enum BOOL { FALSE, TRUE };
```

Se debe notar que por defecto a la primer etiqueta (**FALSE** en nuestro ejemplo), nuestra declaración **enum** la inicializa con 0, a la segunda (**TRUE**) con 1, y así sucesivamente (si hubiera más) hasta la última .

En el ejemplo sólo teníamos dos etiquetas, pero si para un programa en particular declaráramos la enumeración de **zonas**, y sólo necesitamos considerar algunas de ellas que se corresponderán con los valores 0, 1, 3, 5, 6, 9, 10 y 11, podríamos tratarla del siguiente modo :

```
enum BOOL { FALSE, TRUE };
```

```
enum zonas { CAPITAL,      BSAIRES,  
              CORDOBA = 3, STAFE = 5,  
              LAPAMPA,     ERIOS = 9,  
              CORRTES,     MISIONES };
```

Se debe notar que por defecto a la primer etiqueta (**FALSE** en nuestro ejemplo), nuestra declaración **enum** la inicializa con 0, a la segunda (**TRUE**) con 1, y así sucesivamente (si hubiera más) hasta la última .

En el ejemplo sólo teníamos dos etiquetas, pero si para un programa en particular declaráramos la enumeración de **zonas**, y sólo necesitamos considerar algunas de ellas que se corresponderán con los valores 0, 1, 3, 5, 6, 9, 10 y 11, podríamos tratarla del siguiente modo :

```
enum BOOL { FALSE, TRUE };

enum zonas { CAPITAL,          BSAIRES,
              CORDOBA = 3,     STAFE = 5,
              LAPAMPA,         ERIOS = 9,
              CORRTES,         MISIONES };

enum valores { NEGA = -2147483647,
               POSI = 2147483647,
               CERO = 0 };
```

Se debe tener presente que las enumeraciones permiten definir etiquetas que tendrán asociados únicamente valores enteros, estos pueden tener cualquier valor dentro del rango que admita el compilador para valores **int** .

Las uniones brindan un modo de almacenar un tipo de información u otro en el mismo espacio de almacenamiento, segun las necesidades de un programa en particular

Las uniones brindan un modo de almacenar un tipo de información u otro en el mismo espacio de almacenamiento, segun las necesidades de un programa en particular

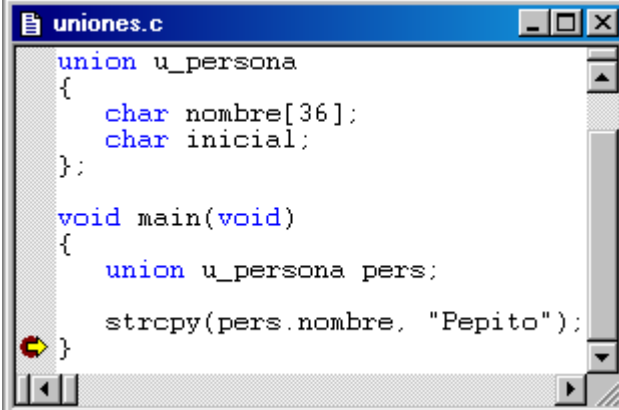
Este es un ejemplo sencillo que nos permitiría comprobar que una vez declarada una variable que responda a esta union si se completa el nombre con p. ej.: "Pedro", la inicial pasa a contener el carácter 'P' y modificando nombre o inicial se altera el otro miembro .

```
union u_persona
{
    char nombre{36};
    char inicial;
};
```

Las uniones brindan un modo de almacenar un tipo de información u otro en el mismo espacio de almacenamiento, según las necesidades de un programa en particular



Este es un ejemplo sencillo que nos permitiría comprobar que una vez declarada una variable que responda a esta union si se completa el nombre con p. ej.: "Pedro", la inicial pasa a contener el carácter 'P' y modificando nombre o inicial se altera el otro miembro .

```
union u_persona
{
    char nombre[36];
    char inicial;
};
```



```
uniones.c
union u_persona
{
    char nombre[36];
    char inicial;
};

void main(void)
{
    union u_persona pers;
    strcpy(pers.nombre, "Pepito");
}
```

Name	Value
 pers	{...}
 nombre	0x0012ff5c "Pepito"
inicial	80 'P'

Las uniones brindan un modo de almacenar un tipo de información u otro en el mismo espacio de almacenamiento, según las necesidades de un programa en particular

Este es un ejemplo sencillo que nos permitiría comprobar que una vez declarada una variable que responda a esta unión si se completa el nombre con p. ej.: "Pedro", la inicial pasa a contener el carácter 'P' y modificando nombre o inicial se altera el otro miembro .

```
union u_persona
{
    char nombre[36];
    char inicial;
};
```

Un ejemplo no tan trivial sería aquel en que necesitemos almacenar por ejemplo las dimensiones de figuras geométricas (p.ej.: círculos, triángulos y rectángulos)

```
enum t_figu { CIRC, TRIA, RECT };
/* las etiquetas de la enum
 * t_figura permiten identificar
 * de qué figura se trata
 */

struct s_figura
{
    float      perim;
    float      superf;
    enum t_figu figura;
    union
    {
        float      radio;
        struct
        {
            float base;
            float altura;
        } rectan;
    } u;
};
```

Las uniones brindan un modo de almacenar un tipo de información u otro en el mismo espacio de almacenamiento, según las necesidades de un programa en particular

C:\...WariosCpp\uniones.c

```
enum t_figu { CIRC, TRIA, RECT };

struct s_figura
{
    float      perim;
    float      superf;
    enum t_figu figura;
    union
    {
        float      radio;
        struct
        {
            float base;
            float altura;
        } rectan;
    } u;
};

void main(void)
{
    struct s_figura figu;
    figu.figura = CIRC;
    figu.u.radio = 5.2f;
    figu.perim = M_PI *
        figu.u.radio * 2;
    figu.superf = M_PI *
        pow(figu.u.radio, 2);
}
```

Un ejemplo no tan trivial sería aquel en que necesitemos almacenar por ejemplo las dimensiones de figuras geométricas (p.ej.: círculos, triángulos y rectángulos)

```
enum t_figu { CIRC, TRIA, RECT };
/* las etiquetas de la enum
 * t_figura permiten identificar
 * de qué figura se trata
 */

struct s_figura
{
    float      perim;
    float      superf;
    enum t_figu figura;
    union
    {
        float      radio;
        struct
        {
            float base;
            float altura;
        } rectan;
    } u;
};
```

Las uniones brindan un modo de almacenar un tipo de información u otro en el mismo espacio de almacenamiento, según las necesidades de un programa en particular

C:\...WariosCpp\uniones.c

```
enum t_figu { CIRC, TRIA, RECT };
```

Name	Value
figu	{...}
perim	32.6726
superf	84.9487
figura	0
u	{...}
radio	5.20000
rectan	{...}
base	5.20000
altura	-1.07374e+008

```
};  
  
void main(void)  
{  
  struct s_figura figu;  
  figu.figura = CIRC;  
  figu.u.radio = 5.2f;  
  figu.perim = M_PI *  
    figu.u.radio * 2;  
  figu.superf = M_PI *  
    pow(figu.u.radio, 2);  
};
```

Un ejemplo no tan trivial sería aquel en que necesitemos almacenar por ejemplo las dimensiones de figuras geométricas (p.ej.: círculos, triángulos y rectángulos)

```
enum t_figu { CIRC, TRIA, RECT };  
/* las etiquetas de la enum  
 * t_figura permiten identificar  
 * de qué figura se trata  
 */
```

```
struct s_figura  
{  
  float        perim;  
  float        superf;  
  enum t_figu  figura;  
  union  
  {  
    float      radio;  
    struct  
    {  
      float  base;  
      float  altura;  
    } rectan;  
  } u;  
};
```


Los operadores del c se pueden agrupar en 16 niveles de precedencia o categorías

Los operadores del C se pueden agrupar en 16 niveles de precedencia o categorías

#	Categoría	Operador
1.	La más alta	() [] -> :: .
2.	Unarios	! ~ + - ++ -- & * sizeof new delete (cast)
3.	Multiplicativos	* / %
4.	Acceso a miembros	.* ->*
5.	Aditivos	+ -
6.	Desplazamiento	<< >>
7.	Relacionales	< <= > >=
8.	Igualdad	== !=
9.	AND binario	&
10.	XOR binario	^
11.	OR binario	
12.	AND lógico	&&
13.	OR lógico	
14.	Condicional	?:
15.	Asignación	= *= /= %= += -= &= ^= = <<= >>=
16.	Coma	,

Los operadores del C se pueden agrupar en 16 niveles de precedencia o categorías

#	Categoría	Operador
1.	La más alta	() [] -> :: .
2.	Unarios	! ~ + - ++ --
3.	Multiplicativos	* / %
4.	Acceso a miembros	.* ->*
5.	Aditivos	+ -
6.	Desplazamiento	<< >>
7.	Relacionales	< <= > >=
8.	Igualdad	== !=
9.	AND binario	&
10.	XOR binario	^
11.	OR binario	
12.	AND lógico	&&
13.	OR lógico	
14.	Condicional	?:
15.	Asignación	= *= /= %= += -= &= ^= = <<= >>=
16.	Coma	,

El nivel 1 tiene la mayor precedencia, le sigue el nivel 2, hasta el nivel 16 que tiene el menor nivel de precedencia

Los operadores del C se pueden agrupar en 16 niveles de precedencia o categorías

#	Categoría	Operador
1.	La más alta	() [] -> :: .
2.	Unarios	! ~ + - ++ --
3.	Multiplicativos	* / %
4.	Acceso a miembros	.* ->*
5.	Aditivos	+ -
6.	Desplazamiento	<< >>
7.	Relacionales	< <= > >=
8.	Igualdad	== !=
9.	AND binario	&
10.	XOR binario	^
11.	OR binario	
12.	AND lógico	&&
13.	OR lógico	
14.	Condicional	?:
15.	Asignación	= *= /= %= += -= &= ^= = <<= >>=
16.	Coma	,

El nivel 1 tiene la mayor precedencia, le sigue el nivel 2, hasta el nivel 16 que tiene el menor nivel de precedencia

Los operadores del mismo nivel tienen la misma precedencia

Los operadores del C se pueden agrupar en 16 niveles de precedencia o categorías

#	Categoría	Operador
1.	La más alta	() [] -> :: .
2.	Unarios	! ~ + - ++ --
3.	Multiplicativos	* / %
4.	Acceso a miembros	.* ->*
5.	Aditivos	+ -
6.	Desplazamiento	<< >>
7.	Relacionales	< <= > >=
8.	Igualdad	== !=
9.	AND binario	&
10.	XOR binario	^
11.	OR binario	
12.	AND lógico	&&
13.	OR lógico	
14.	Condicional	?:
15.	Asignación	= *= /= %= += -= &= ^= = <<= >>=
16.	Coma	,

El nivel 1 tiene la mayor precedencia, le sigue el nivel 2, hasta el nivel 16 que tiene el menor nivel de precedencia

Los operadores del mismo nivel tienen la misma precedencia

Los operadores Unarios (#2), el Condicional (#14) y los de Asignación (#15) se evalúan de derecha a izquierda, en tanto que los demás se evalúan de izquierda a derecha

Los operadores del C se pueden agrupar en 16 niveles de precedencia o categorías

#	Categoría	Operador
1.	La más alta	() [] -> :: .
2.	Unarios	! ~ + - ++ --
3.	Multiplicativos	* / %
4.	Acceso a miembros	. * -> *
5.	Aditivos	+ -
6.	Desplazamiento	<< >>
7.	Relacionales	< <= > >=
8.	Igualdad	== !=
9.	AND binario	&
10.	XOR binario	^
11.	OR binario	
12.	AND lógico	&&
13.	OR lógico	
14.	Condicional	?:
15.	Asignación	= *= /= %= +=
16.	Coma	,

El nivel 1 tiene la mayor precedencia, le sigue el nivel 2, hasta el nivel 16 que tiene el menor nivel de precedencia

Los operadores del mismo nivel tienen la misma precedencia

Los operadores Unarios (#2), el Condicional (#14) y los de Asignación (#15) se evalúan de derecha a izquierda, en tanto que los demás se evalúan de izquierda a derecha

En C++ se pueden sobrecargar todos los operadores salvo por
 (::) Operador de alcance
 (.) Acceso a miembros
 (.*) Acceso a miembros C++
 (?:) Condicional

Los operadores del C se pueden agrupar en 16 niveles de precedencia o categorías

#	Categoría	Operador
1.	La más alta	() [] -> :: .
2.	Unarios	! ~ + - ++ -- & * sizeof new delete (cast)
3.	Multiplicativos	* / %
4.	Acceso a miembros	.* ->*
5.	Aditivos	+ -
6.	Desplazamiento	<< >>
7.	Relacionales	< <= > >=
8.	Igualdad	== !=
9.	AND binario	&
10.	XOR binario	^
11.	OR binario	
12.	AND lógico	&&
13.	OR lógico	
14.	Condicional	?:
15.	Asignación	= *= /= %= += -= &= ^= = <<= >>=
16.	Coma	,

Los operadores del C se pueden agrupar en 16 niveles de precedencia o categorías

#	Categoría	Operador
1.	La más alta	() [] -> :: .

#	Categoría	Operador	Qué es o qué hace
1.	La más alta	()	Invocación a función/agrupa operaciones
		[]	Posición en un array
		->	Acceso a miembros mediante puntero a estructura
		::	Operador de C++/C de alcance
		.	Acceso a miembros de estructuras

11.	OR binario	
12.	AND lógico	&&
13.	OR lógico	
14.	Condicional	?:
15.	Asignación	= *= /= %= += -= &= ^= = <<= >>=
16.	Coma	,

Los operadores del C se pueden agrupar en 16 niveles de precedencia o categorías

#	Categoría	Operador
1.	La más alta	() [] -> :: .
2.	Unarios	! ~ + - ++ -- & * sizeof new delete (cast)

#	Categoría	Operador	Qué es o qué hace
2.	Operadores unarios	!	Negación lógica (NOT)
		~	Complemento a 1 binario
		+	Más unario
		-	Menos unario
		++	Pre/pos incremento
		--	Pre/pos decremento
		&	Dirección de
		*	Indirección
		sizeof	Tamaño de (cantidad de bytes)
		new	Asigna memoria dinámica (C++)
		delete	Libera memoria dinámica (C++)
		(cast)	Conversión de tipo

16.	Coma	,
-----	------	---

Los operadores del C se pueden agrupar en 16 niveles de precedencia o categorías

#	Categoría	Operador
1.	La más alta	() [] -> :: .
2.	Unarios	! ~ + - ++ -- & * sizeof new delete (cast)
3.	Multiplicativos	* / %
4.	Aditivos	+ -
5.	Desplazamiento	< >
6.	Comparación	< <= > >=
7.	Relacionales	< <= > >=
8.	Relacionales	< <= > >=
9.	Relacionales	< <= > >=
10.	XOR binario	^
11.	OR binario	
12.	AND lógico	&&
13.	OR lógico	
14.	Condicional	?:
15.	Asignación	= *= /= %= += -= &= ^= = <<= >>=
16.	Coma	,

#	Categoría	Operador	Qué es o qué hace
3.	Multipli-	*	Multiplica
	cativos	/	Divide
		%	Resto de la división entera

Los operadores del C se pueden agrupar en 16 niveles de precedencia o categorías

#	Categoría	Operador
1.	La más alta	() [] -> :: .
2.	Unarios	! ~ + - ++ -- & * sizeof new delete (cast)
3.	Multiplicativos	* / %
4.	Acceso a miembros	.* ->*
5.	Desplazamiento	<< >>
6.	Relacionales	< > <= >=
7.	Relacionales estrictos	<= >= << >>
8.	Relacionales estrictos	<= >= << >>
9.	Relacionales estrictos	<= >= << >>
10.	XOR binario	^
11.	OR binario	
12.	AND lógico	&&
13.	OR lógico	
14.	Condicional	?:
15.	Asignación	= *= /= %= += -= &= ^= = <<= >>=
16.	Coma	,

#	Categoría	Operador	Qué es o qué hace
4.	Acceso a	.*	C++
	miembros	->*	C++

Los operadores del C se pueden agrupar en 16 niveles de precedencia o categorías

#	Categoría	Operador
1.	La más alta	() [] -> :: .
2.	Unarios	! ~ + - ++ -- & * sizeof new delete (cast)
3.	Multiplicativos	* / %
4.	Acceso a miembros	.* ->*
5.	Aditivos	+ -
6.		
7.		
8.		
9.		
10.		
11.	OR binario	
12.	AND lógico	&&
13.	OR lógico	
14.	Condicional	?:
15.	Asignación	= *= /= %= += -= &= ^= = <<= >>=
16.	Coma	,

#	Categoría	Operador	Qué es o qué hace
5.	Additivos	+	Suma
		-	Resta

Los operadores del C se pueden agrupar en 16 niveles de precedencia o categorías

#	Categoría	Operador
1.	La más alta	() [] -> :: .
2.	Unarios	! ~ + - ++ -- & * sizeof new delete (cast)
3.	Multiplicativos	* / %
4.	Acceso a miembros	. * -> *
5.	Aditivos	+ -
6.	Desplazamiento	<< >>
7.	Relacionales	< > <= >=
8.	Relacionales estrictos	<= >= < >
9.	Relacionales estrictos	<= >= < >
10.	Relacionales estrictos	<= >= < >
11.	Relacionales estrictos	<= >= < >
12.	AND lógico	&&
13.	OR lógico	
14.	Condicional	?:
15.	Asignación	= *= /= %= += -= &= ^= = <<= >>=
16.	Coma	,

#	Categoría	Operador	Qué es o qué hace
6.	Desplazamiento	<<	Desplazamiento a izquierda
	to de bits	>>	Desplazamiento a derecha

Los operadores del C se pueden agrupar en 16 niveles de precedencia o categorías

#	Categoría	Operador
1.	La más alta	() [] -> :: .
2.	Unarios	! ~ + - ++ -- & * sizeof new delete (cast)
3.	Multiplicativos	* / %
4.	Acceso a miembros	.* ->*
5.	Aditivos	+ -
6.	Desplazamiento	<< >>
7.	Relacionales	< <= > >=
8.	Bit a bit	& ^ && >> << <<= >>=
9.	Asignación	= *= /= %= += -= &= ^= = <<= >>=
10.	Coma	,
11.	Condicionales	?:
12.	Operadores de punto y coma	;
13.	Operadores de punto y coma	;
14.	Operadores de punto y coma	;
15.	Asignación	= *= /= %= += -= &= ^= = <<= >>=
16.	Coma	,

#	Categoría	Operador	Qué es o qué hace
7.	Relacionales	<	Menor que
		<=	Menor o igual que
		>	Mayor que
		>=	Mayor o igual que

Los operadores del C se pueden agrupar en 16 niveles de precedencia o categorías

#	Categoría	Operador
1.	La más alta	() [] -> :: .
2.	Unarios	! ~ + - ++ -- & * sizeof new delete (cast)
3.	Multiplicativos	* / %
4.	Acceso a miembros	.* ->*
5.	Aditivos	+ -
6.	Desplazamiento	<< >>
7.	Relacionales	< <= > >=
8.	Igualdad	== !=
9.		
10.		
11.		
12.		
13.		
14.	Condicional	?:
15.	Asignación	= *= /= %= += -= &= ^= = <<= >>=
16.	Coma	,

#	Categoría	Operador	Qué es o qué hace
8.	Igualdad	==	Igual a
		!=	Distinto a

Los operadores del C se pueden agrupar en 16 niveles de precedencia o categorías

#	Categoría	Operador
1.	La más alta	() [] -> :: .
2.	Unarios	! ~ + - ++ -- & * sizeof new delete (cast)
3.	Multiplicativos	* / %
4.	Acceso a miembros	. * -> *
5.	Aditivos	+ -
6.	Desplazamiento	<< >>
7.	Relacionales	< <= > >=
8.	Igualdad	== !=
9.	AND binario	&
10.		
11.		
12.		
13.		
14.	Condicional	?:
15.	Asignación	= *= /= %= += -= &= ^= = <<= >>=
16.	Coma	,

#	Categoría	Operador	Qué es o qué hace
9.	AND binario	&	AND binario

Los operadores del C se pueden agrupar en 16 niveles de precedencia o categorías

#	Categoría	Operador
1.	La más alta	() [] -> :: .
2.	Unarios	! ~ + - ++ -- & * sizeof new delete (cast)
3.	Multiplicativos	* / %
4.	Acceso a miembros	. * -> *
5.	Aditivos	+ -
6.	Desplazamiento	<< >>
7.	Relacionales	< <= > >=
8.	Igualdad	== !=
9.	AND binario	&
10.	XOR binario	^
11.	OR binario	
12.	Condicional	?:
13.	Asignación	=
14.	Combinación de asignación	*= /= %= += -= &= ^= = <<= >>=
15.	Asignación	= *= /= %= += -= &= ^= = <<= >>=
16.	Coma	,

#	Categoría	Operador	Qué es o qué hace
10.	XOR binario	^	XOR binario

Los operadores del C se pueden agrupar en 16 niveles de precedencia o categorías

#	Categoría	Operador
1.	La más alta	() [] -> :: .
2.	Unarios	! ~ + - ++ -- & * sizeof new delete (cast)
3.	Multiplicativos	* / %
4.	Acceso a miembros	. * -> *
5.	Aditivos	+ -
6.	Desplazamiento	<< >>
7.	Relacionales	< <= > >=
8.	Igualdad	== !=
9.	AND binario	&
10.	XOR binario	^
11.	OR binario	
12.	Condicional	?:
13.	Asignación	=
14.	Asignación compuesta	&= *= += -= <<= >>=
15.	Coma	,
16.	Coma	,

#	Categoría	Operador	Qué es o qué hace
11.	OR binario		OR binario

Los operadores del C se pueden agrupar en 16 niveles de precedencia o categorías

#	Categoría	Operador								
1.	La más alta	() [] -> :: .								
2.	Unarios	! ~ + - ++ -- & * sizeof new delete (cast)								
3.	Multiplicativos	* / %								
4.	Acceso a miembros	. * -> *								
5.	Aditivos	+ -								
6.	Desplazamiento	<< >>								
7.	Relacionales	< <= > >=								
<table><tr><th>#</th><th>Categoría</th><th>Operador</th><th>Qué es o qué hace</th></tr><tr><td>12.</td><td>AND Lógico</td><td>&&</td><td>AND lógico</td></tr></table>			#	Categoría	Operador	Qué es o qué hace	12.	AND Lógico	&&	AND lógico
#	Categoría	Operador	Qué es o qué hace							
12.	AND Lógico	&&	AND lógico							
12.	AND lógico	&&								
13.	OR lógico									
14.	Condicional	?:								
15.	Asignación	= *= /= %= += -= &= ^= = <<= >>=								
16.	Coma	,								

Los operadores del C se pueden agrupar en 16 niveles de precedencia o categorías

#	Categoría	Operador
1.	La más alta	() [] -> :: .
2.	Unarios	! ~ + - ++ -- & * sizeof new delete (cast)
3.	Multiplicativos	* / %
4.	Acceso a miembros	.* ->*
5.	Aditivos	+ -
6.	Desplazamiento	<< >>
7.	Relacionales	< <= > >=
8.	Igualdad	== !=

#	Categoría	Operador	Qué es o qué hace
13.	OR Lógico		OR lógico

13.	OR lógico	
14.	Condicional	?:
15.	Asignación	= *= /= %= += -= &= ^= = <<= >>=
16.	Coma	,

Los operadores del C se pueden agrupar en 16 niveles de precedencia o categorías

#	Categoría	Operador
1.	La más alta	() [] -> :: .
2.	Unarios	! ~ + - ++ -- & * sizeof new delete (cast)
3.	Multiplicativos	* / %
4.	Acceso a miembros	.* ->*
5.	Aditivos	+ -
6.	Desplazamiento	<< >>
7.	Relacionales	< <= > >=
8.	Igualdad	== !=

#	Categoría	Operador	Qué es o qué hace
14.	Condicional	?:	a ? x : y <=> si a entonces x, si_no y

14.	Condicional	?:
15.	Asignación	= *= /= %= += -= &= ^= = <<= >>=
16.	Coma	,

Los operadores del C se pueden agrupar en 16 niveles de precedencia o categorías

#	Categoría	Operador
1.	La más alta	() [] -> ::

#	Categoría	Operador	Qué es o qué hace
15.	Asignación	=	Asigna
		*=	Asigna el producto
		/=	Asigna el cociente
		%=	Asigna el resto
		+=	Asigna la suma
		-=	Asigna la diferencia
		&=	Asigna el AND binario
		^=	Asigna el XOR binario
		=	Asigna el OR binario
		<<=	Asigna el desplazamiento binario a izquierda
		>>=	Asigna el desplazamiento binario a derecha

15.	Asignación	= *= /= %= += -= &= ^= = <<= >>=
16.	Coma	,

Los operadores del C se pueden agrupar en 16 niveles de precedencia o categorías

#	Categoría	Operador
1.	La más alta	() [] -> :: .
2.	Unarios	! ~ + - ++ -- & * sizeof new delete (cast)
3.	Multiplicativos	* / %
4.	Acceso a miembros	.* ->*
5.	Aditivos	+ -
6.	Desplazamiento	<< >>
7.	Relacionales	< <= > >=
8.	Igualdad	== !=
9.	AND binario	&
10.	XOR binario	^
11.	OR binario	

#	Categoría	Operador	Qué es o qué hace
16.	Coma	,	Evalúa

16. Coma

Recordemos que :

Recordemos que :

**Para el C dónde se requiere una expresión esta puede ser
una expresión vacía (simplemente ; -punto y coma-),
una única expresión válida (siempre terminada con ;) o
un bloque de expresiones (encerrado entre { }) .**

Recordemos que :

**Para el C dónde se requiere una expresión esta puede ser
una expresión vacía (simplemente ; -punto y coma-),
una única expresión válida (siempre terminada con ;) o
un bloque de expresiones (encerrado entre { }) .**

**A su vez este bloque de expresiones puede contener
una expresión,
varias expresiones,
ninguna expresión (bloque vacío),
una o varias expresiones vacías .**

Recordemos que :

Para el C dónde se requiere una expresión esta puede ser una expresión vacía (simplemente ; -punto y coma-), una única expresión válida (siempre terminada con ;) o un bloque de expresiones (encerrado entre { }) .

**A su vez este bloque de expresiones puede contener una expresión,
varias expresiones,
ninguna expresión (bloque vacío),
una o varias expresiones vacías .**

**Cuando en C se habla de <expresión> se refiere a cualquier operación,
comparación,
invocación a función no void,
expresión lógica,
evaluación del contenido de una variable,
etc.,
que se pueda evaluar como un valor entero.**

Recordemos que :

Para el **C** dónde se requiere una expresión esta puede ser una expresión vacía (simplemente ; -punto y coma-), una única expresión válida (siempre terminada con ;) o un bloque de expresiones (encerrado entre { }) .

A su vez este bloque de expresiones puede contener una expresión,
varias expresiones,
ninguna expresión (bloque vacío),
una o varias expresiones vacías .

Cuando en **C** se habla de **<expresión>** se refiere a cualquier operación,
comparación,
invocación a función no void,
expresión lógica,
evaluación del contenido de una variable,
etc.,
que se pueda evaluar como un valor entero.

Si ese valor entero es 0 (cero), para **C** es falso;
cualquier otro valor es verdad

Recordemos que :

Recordemos que :

**En el condicional `if`,
el `else` es opcional .**

```
if (<condición>
    <expresión>;
else
    <expresión>;
```

Recordemos que :

**En el condicional if,
el else es opcional .**

**En el condicional switch,
default y break son op-
cionales .
Es una expresión condicio-
nal de alternativas múlti-
ples .**

```
if(<condición>
    <expresión>;
else
    <expresión>;

switch(<condición>)
{
case <expresión-constante-entera-1> :
    <expresión-1>; /* una, varias o ninguna */
    break;
case <expresión-constante-entera-2> :
    <expresión-2>; /* una, varias o ninguna */
    break;

/* ... tantos case como sean necesarios */

default :
    <expresión-n>; /* una, varias o ninguna */
}
```

Recordemos que :

**En el condicional if,
el else es opcional .**

**En el condicional switch,
default y break son op-
cionales .
Es una expresión condicio-
nal de alternativas múlti-
ples .**

El ciclo while .

```
if(<condición>
    <expresión>;
else
    <expresión>;
```

```
switch(<condición>)
{
case <expresión-constante-entera-1> :
    <expresión-1>; /* una, varias o ninguna */
    break;
case <expresión-constante-entera-2> :
    <expresión-2>; /* una, varias o ninguna */
    break;
```

```
while(<condición>)
    <expresión>;
```

Recordemos que :

En el condicional **if**,
el **else** es opcional .

En el condicional **switch**,
default y **break** son op-
cionales .
Es una expresión condicio-
nal de alternativas múlti-
ples .

El ciclo **while** .

El ciclo **for** .

```
if(<condición>
    <expresión>;
else
    <expresión>;
```

```
switch(<condición>)
{
case <expresión-constante-entera-1> :
    <expresión-1>; /* una, varias o ninguna */
    break;
case <expresión-constante-entera-2> :
    <expresión-2>; /* una, varias o ninguna */
    break;
```

```
while(<condición>)
    <expresión>;
```

```
for(<expresión-1>; <expresión-2>; <expresión-3>)
    <expresión>;
```


Recordemos que :

En el condicional **if**,
el **else** es opcional .

En el condicional **switch**,
default y **break** son op-
cionales .
Es una expresión condicio-
nal de alternativas múlti-
ples .

El ciclo **while** .

El ciclo **for** .

La **<expresión-1>** se ejecutará por úni-
ca vez, habitualmente se emplea para ini-
cialización de variables

```
if(<condición>
    <expresión>;
else
    <expresión>;
```

```
switch(<condición>)
{
case <expresión-constante-entera-1> :
    <expresión-1>; /* una, varias o ninguna */
    break;
case <expresión-constante-entera-2> :
    <expresión-2>; /* una, varias o ninguna */
    break;
```

```
while(<condición>)
    <expresión>;
```

```
for(<expresión-1>; <expresión-2>; <expresión-3>)
    <expresión>;
```

Recordemos que :

En el condicional **if**,
el **else** es opcional .

En el condicional **switch**,
default y **break** son op-
cionales .
Es una expresión condicio-
nal de alternativas múlti-
ples .

El ciclo **while** .

El ciclo **for** .

La **<expresión-1>** se ejecutará por úni-
ca vez, habitualmente se emplea para ini-
cialización de variables


La **<expresión-2>** puede ser cualquier
expresión válida C que se evaluará como
expresión booleana (**falso** = 0,
verdad = ~falso) antes de ejecutar la
<expresión> .

```
if(<condición>)  
    <expresión>;  
else  
    <expresión>;
```

```
switch(<condición>)  
{  
case <expresión-constante-entera-1> :  
    <expresión-1>; /* una, varias o ninguna */  
    break;  
case <expresión-constante-entera-2> :  
    <expresión-2>; /* una, varias o ninguna */  
    break;
```

```
while(<condición>)  
    <expresión>;
```

```
for(<expresión-1>; <expresión-2>; <expresión-3>)  
    <expresión>;
```



Control de flujo de ejecución del c

Recordemos que :

En el condicional **if**,
el **else** es opcional .

En el condicional **switch**,
default y **break** son op-
cionales .
Es una expresión condicio-
nal de alternativas múlti-
ples .

El ciclo **while** .

El ciclo **for** .

La **<expresión-1>** se ejecutará por úni-
ca vez, habitualmente se emplea para ini-
cialización de variables

La **<expresión-2>** puede ser cualquier
expresión válida C que se evaluará como
expresión booleana (**falso** = 0,
verdad = ~falso) antes de ejecutar la
<expresión> .

```
if(<condición>)  
    <expresión>;  
else  
    <expresión>;
```

```
switch(<condición>)  
{  
case <expresión-constante-entera-1> :  
    <expresión-1>; /* una, varias o ninguna */  
    break;  
case <expresión-constante-entera-2> :  
    <expresión-2>; /* una, varias o ninguna */  
    break;
```

```
while(<condición>)  
    <expresión>;
```

```
for(<expresión-1>; <expresión-2>; <expresión-3>)  
    <expresión>;
```

La **<expresión-3>** (habitualmente llama-
da incremento) se ejecutará cada vez que
se haya ejecutado la **<expresión>** y antes
de volver a evaluar la **<expresión-2>** .

Recordemos que :

En el condicional `if`,
el `else` es opcional .

En el condicional `switch`,
`default` y `break` son op-
cionales .
Es una expresión condicio-
nal de alternativas múlti-
ples .

El ciclo `while` .

El ciclo `for` .

```
if(<condición>
    <expresión>;
else
    <expresión>;
```

```
switch(<condición>)
{
case <expresión-constante-entera-1> :
    <expresión-1>; /* una, varias o ninguna */
    break;
case <expresión-constante-entera-2> :
    <expresión-2>; /* una, varias o ninguna */
    break;
```

```
while(<condición>)
    <expresión>;
```

```
for(<expresión-1>; <expresión-2>; <expresión-3>)
    <expresión>;
```

La `<expresión-1>` se ejecutará por úni-
ca vez, habitualmente se emplea para ini-
cialización de variables

La `<expresión-2>` puede ser cualquier
expresión válida C que se evaluará como
expresión booleana (`falso = 0`,
`verdad = ~falso`) antes de ejecutar la
`<expresión>` .

La `<expresión-3>` (habitualmente llama-
da incremento) se ejecutará cada vez que
se haya ejecutado la `<expresión>` y antes
de volver a evaluar la `<expresión-2>` .

De ser necesaria/s más de una
`<expresión-1>/<expresión-3>`
se separan con `(,)` .

Control de flujo de ejecución del c

Recordemos que :

En el condicional **if**,
el **else** es opcional .

En el condicional **switch**,
default y **break** son op-
cionales .
Es una expresión condicio-
nal de alternativas múlti-
ples .

El ciclo **while** .

El ciclo **for** .

El ciclo **do...while** .

```
if(<condición>
    <expresión>;
else
    <expresión>;
```

```
switch(<condición>)
{
case <expresión-constante-entera-1> :
    <expresión-1>; /* una, varias o ninguna */
    break;
case <expresión-constante-entera-2> :
    <expresión-2>; /* una, varias o ninguna */
    break;
```

```
while(<condición>)
    <expresión>;
```

```
for(<expresión-1>; <expresión-2>; <expresión-3>)
    <expresión>;
```

```
do
{
    <expresión>; /* una o varias */
} while(<condición>;
```

Recordemos que :

En el condicional **if**,
el **else** es opcional .

En el condicional **switch**,
default y **break** son op-
cionales .
Es una expresión condicio-
nal de alternativas múlti-
ples .

El ciclo **while** .

El ciclo **for** .

El ciclo **do...while** .

El empleo de las llaves ({ })
es necesario cuando hay más
de una <expresión> (en la
mayoría de los compiladores)

```
if(<condición>)  
    <expresión>;  
else  
    <expresión>;
```

```
switch(<condición>)  
{  
case <expresión-constante-entera-1> :  
    <expresión-1>; /* una, varias o ninguna */  
    break;  
case <expresión-constante-entera-2> :  
    <expresión-2>; /* una, varias o ninguna */  
    break;
```

```
while(<condición>)  
    <expresión>;
```

```
for(<expresión-1>; <expresión-2>; <expresión-3>)  
    <expresión>;
```

```
do  
{  
    <expresión>; /* una o varias */  
} while(<condición>;
```

Recordemos que :

En el condicional **if**,
el **else** es opcional .

En el condicional **switch**,
default y **break** son op-
cionales .
Es una expresión condicio-
nal de alternativas múlti-
ples .

El ciclo **while** .

El ciclo **for** .

El ciclo **do...while** .

El empleo de las llaves ({ })
es necesario cuando hay más
de una **<expresión>** (en la
mayoría de los compiladores)

```
if(<condición>)  
    <expresión>;  
else  
    <expresión>;
```

```
switch(<condición>)  
{  
case <expresión-constante-entera-1> :  
    <expresión-1>; /* una, varias o ninguna */  
    break;  
case <expresión-constante-entera-2> :  
    <expresión-2>; /* una, varias o ninguna */  
    break;  
}
```

```
while(<condición>)  
    <expresión>;
```

```
for(<expresión-1>; <expresión-2>; <expresión-3>)  
    <expresión>;
```

```
do  
{  
    <expresión>; /* una o varias */  
} while(<condición>;
```

La **<expresión>** siem-
pre se ejecuta al menos
una vez, antes de con-
trolar la **<condición>**

Recordemos que :

En el condicional **if**,
el **else** es opcional .

En el condicional **switch**,
default y **break** son op-
cionales .
Es una expresión condicio-
nal de alternativas múlti-
ples .

El ciclo **while** .

El ciclo **for** .

El ciclo **do...while** .

```
if(<condición>
    <expresión>;
else
    <expresión>;
```

```
switch(<condición>)
{
case <expresión-constante-entera-1> :
    <expresión-1>; /* una, varias o ninguna */
    break;
case <expresión-constante-entera-2> :
    <expresión-2>; /* una, varias o ninguna */
    break;
```

```
while(<condición>)
    <expresión>;
```

```
for(<expresión-1>; <expresión-2>; <expresión-3>)
    <expresión>;
```

```
do
{
    <expresión>; /* una o varias */
} while(<condición>;
```

El empleo de las llaves ({})
es necesario cuando hay más
de una <expresión> (en la
mayoría de los compiladores)

Por legibilidad, siempre
utilizaremos ({}) en los
ciclos **do...while**

La <expresión> siem-
pre se ejecuta al menos
una vez, antes de con-
trolar la <condición>

Recordemos que :

En el condicional **if**,
el **else** es opcional .

En el condicional **switch**,
default y **break** son op-
cionales .
Es una expresión condicio-
nal de alternativas múlti-
ples .

El ciclo **while** .

El ciclo **for** .

El ciclo **do...while** .

La expresión o sentencia
goto produce un salto
incondicional a la etiqueta
indicada .

```
if(<condición>
    <expresión>;
else
    <expresión>;
```

```
switch(<condición>)
{
case <expresión-constante-entera-1> :
    <expresión-1>; /* una, varias o ninguna */
    break;
case <expresión-constante-entera-2> :
    <expresión-2>; /* una, varias o ninguna */
    break;
```

```
while(<condición>)
    <expresión>;
```

```
for(<expresión-1>; <expresión-2>; <expresión-3>)
    <expresión>;
```

```
do
{
```

```
    /* ... */
    goto <etiqueta>;
    /* ... más expresiones ... */
<etiqueta> :
    /* ... */
```

Recordemos que :

En el condicional **if**,
el **else** es opcional .

En el condicional **switch**
default y **break** son op-
cionales .
Es una expresión condicio-
nal de alternativas múlti-
ples .

El ciclo **while** .

El ciclo **for** .

El ciclo **do...while** .

La expresión o sentencia
goto produce un salto
incondicional a la etiqueta
indicada .

```
if(<condición>
    <expresión>;
else
    <expresión>;
```

```
switch(<condición>
{
    c
```

La expresión o sentencia **break** produce la inte-
rrupción de las estructuras de control **switch**,
while, **do...while** o **for** y que se continúe
con la siguiente expresión fuera de la estructu-
ra de control .

```
*/
```

```
*/
```

```
while(<condición>
    <expresión>;
```

```
for(<expresión-1>; <expresión-2>; <expresión-3>)
    <expresión>;
```

```
do
{
```

```
    /* ... */
    goto <etiqueta>;
    /* ... más expresiones ... */
<etiqueta> :
    /* ... */
```

Recordemos que :

En el condicional **if**,
el **else** es opcional .

En el condicional **switch**,
default y **break** son op-
cionales .
Es una expresión condicio-
nal de alternativas múlti-
ples .

El ciclo **while**.

El ciclo **for**.

El ciclo **do...while**.

La expresión o sentencia
goto produce un salto
incondicional a la etiqueta
indicada .

```
if(<condición>)  
    <expresión>;  
else  
    <expresión>;
```

```
switch(<condición>)
```

```
{  
c  
c
```

La expresión o sentencia **break** produce la inte-
rrupción de las estructuras de control **switch**,
while, **do...while** o **for** y que se continúe
con la siguiente expresión fuera de la estructu-
ra de control .

```
*/
```

```
*/
```

```
w
```

La expresión o sentencia **continue** produce la
siguiente iteración de las estructuras de control
while, **do...while** o **for** .

```
(>)
```

```
<expresión>;
```

```
do  
{
```

```
/* ... */
```

```
goto <etiqueta>;
```

```
/* ... más expresiones ... */
```

```
<etiqueta> :
```

```
/* ... */
```


Recordemos que un puntero es simplemente una variable que puede contener una dirección de memoria (de otra variable, de una función, o una dirección 'void') .

Recordemos que un puntero es simplemente una variable que puede contener una dirección de memoria (de otra variable, de una función, o una dirección 'void') .

Puntero a una variable .

Recordemos que un puntero es simplemente una variable que puede contener una dirección de memoria (de otra variable, de una función, o una dirección 'void') .

Puntero a una variable .

```
void main(void)
{
```

`main`

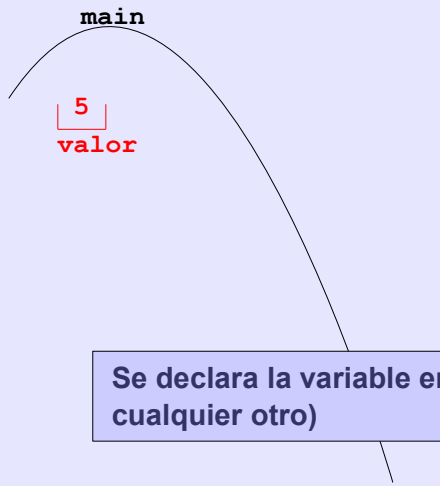


Para visualizar nuestra explicación imaginemos que el dibujo de la izquierda es un gráfico de la memoria de variables de nuestra función `main`

Recordemos que un puntero es simplemente una variable que puede contener una dirección de memoria (de otra variable, de una función, o una dirección 'void') .

Puntero a una variable .

```
void main(void)
{
    int valor = 5;
```

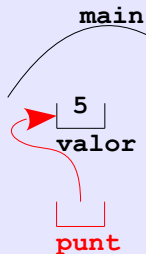


Se declara la variable entera `valor`, inicializada con 5 (podría haber sido cualquier otro)

Recordemos que un puntero es simplemente una variable que puede contener una dirección de memoria (de otra variable, de una función, o una dirección 'void') .

Puntero a una variable .

```
void main(void)
{
    int valor = 5;
    int *punt = &valor;
```



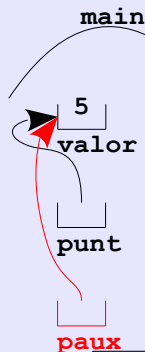
Se declara la variable entera `valor`, inicializada con 5 (podría haber sido

Se declara la variable puntero a entero `punt`, inicializada con la dirección de memoria en que se encuentra la variable `valor`

Recordemos que un puntero es simplemente una variable que puede contener una dirección de memoria (de otra variable, de una función, o una dirección 'void') .

Puntero a una variable .

```
void main(void)
{
    int valor = 5;
    int *punt = &valor;
    void *paux = (void *)punt;
```



Se declara la variable entera `valor`, inicializada con 5 (podría haber sido

Se declara la variable puntero a entero `punt`, inicializada con la dirección de memoria en que se encuentra la variable `valor`

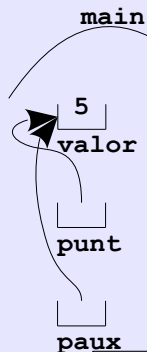
Se declara la variable puntero `paux`, inicializada también con la dirección de memoria en que se encuentra la variable `valor` (o se podría hacer con `void *paux = (void *)&valor;`). Note que para hacerlo, hay que convertir la dirección de memoria que tiene `punt` para que sea una dirección `void` (mediante la conversión de tipo `(void *)`)

Recordemos que un puntero es simplemente una variable que puede contener una dirección de memoria (de otra variable, de una función, o una dirección 'void') .

Puntero a una variable .

```
void main(void)
{
    int valor = 5;
    int *punt = &valor;
    void *paux = (void *)punt;

    printf("Valor contiene %d\n", valor);
}
```



Se declara la variable entera **valor**, inicializada con 5 (podría haber sido

Se declara la variable puntero a entero **punt**, inicializada con la dirección de memoria en que se encuentra la variable **valor**

Se declara la variable puntero **paux**, inicializada también con la dirección de memoria en que se encuentra la variable **valor** (o se podría hacer con

Valiéndonos de la función **printf**, podemos mostrar la variable **valor**

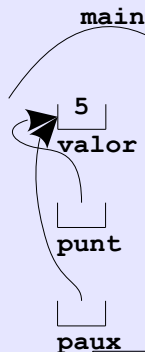
convertir la dirección de memoria que tiene **punt** para que sea una dirección **void** (mediante la conversión de tipo **(void *)**)

Recordemos que un puntero es simplemente una variable que puede contener una dirección de memoria (de otra variable, de una función, o una dirección 'void') .

Puntero a una variable .

```
void main(void)
{
    int valor = 5;
    int *punt = &valor;
    void *paux = (void *)punt;

    printf("Valor contiene %d\n", valor);
    printf("Punt apunta a %d\n", *punt);
}
```



Se declara la variable entera **valor**, inicializada con 5 (podría haber sido

Se declara la variable puntero a entero **punt**, inicializada con la dirección de memoria en que se encuentra la variable **valor**

Se declara la variable puntero **paux**, inicializada también con la dirección de memoria en que se encuentra la variable **valor** (o se podría hacer con

Valiéndonos de la función **printf**, podemos mostrar la variable **valor**

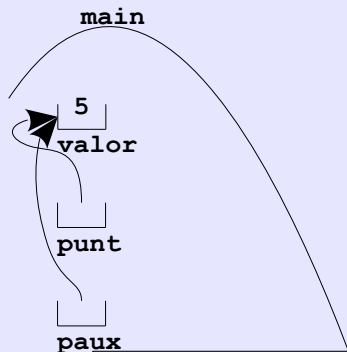
O mostrarla utilizando el puntero **punt** (note el uso del *)

Recordemos que un puntero es simplemente una variable que puede contener una dirección de memoria (de otra variable, de una función, o una dirección 'void') .

Puntero a una variable .

```
void main(void)
{
    int valor = 5;
    int *punt = &valor;
    void *paux = (void *)punt;

    printf("Valor contiene %d\n", valor);
    printf("Punt apunta a %d\n", *punt);
    printf("Paux apunta a %d\n", *(int *)paux);
}
```



Se declara la variable entera **valor**, inicializada con 5 (podría haber sido

Se declara la variable puntero a entero **punt**, inicializada con la dirección de memoria en que se encuentra la variable **valor**

Se declara la variable puntero **paux**, inicializada también con la dirección de memoria en que se encuentra la variable **valor** (o se podría hacer con

Valiéndonos de la función **printf**, podemos mostrar la variable **valor**

O mostrarla utilizando el puntero **punt** (note el uso del *****)

O la variable **valor** usando el puntero **paux** (note el uso del ***(int *)**)

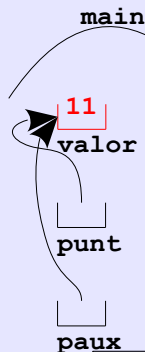
Punteros

Recordemos que un puntero es simplemente una variable que puede contener una dirección de memoria (de otra variable, de una función, o una dirección 'void') .

Puntero a una variable .

```
void main(void)
{
    int valor = 5;
    int *punt = &valor;
    void *paux = (void *)punt;

    printf("Valor contiene %d\n", valor);
    printf("Punt apunta a %d\n", *punt);
    printf("Paux apunta a %d\n", *(int *)paux);
    valor = 11;
```



Se declara la variable entera `valor`, inicializada con 5 (podría haber sido cualquier otro valor).

Se declara la variable puntero `punt`, inicializada con la dirección de memoria en que se encuentra la variable `valor`.

Se declara la variable puntero `paux`, inicializada también con la dirección de memoria en que se encuentra la variable `valor` (o se podría hacer con la dirección de memoria en que se encuentra la variable `punt`).

Valiéndonos de la función `printf`, podemos mostrar la variable `valor`

O mostrarla utilizando el puntero `punt` (note el uso del `*`)

O la variable `valor` usando el puntero `paux` (note el uso del `*(int *)`)

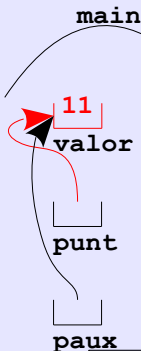
Punteros

Recordemos que un puntero es simplemente una variable que puede contener una dirección de memoria (de otra variable, de una función, o una dirección 'void') .

Puntero a una variable .

```
void main(void)
{
    int valor = 5;
    int *punt = &valor;
    void *paux = (void *)punt;

    printf("Valor contiene %d\n", valor);
    printf("Punt apunta a  %d\n", *punt);
    printf("Paux apunta a  %d\n", *(int *)paux);
    valor = 11;           /* equivale a : */
    *punt = 11;
}
```



Si queremos modificar la variable `valor` valiéndonos del identificador, resulta

Lo podríamos haber hecho mediante el puntero `punt` (note la necesidad del uso del `*`)

Valiéndonos de la función `printf`, podemos mostrar la variable `valor`

O mostrarla utilizando el puntero `punt` (note el uso del `*`)

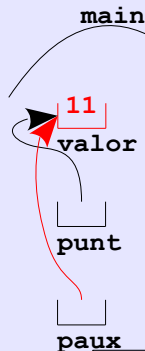
O la variable `valor` usando el puntero `paux` (note el uso del `*(int *)`)

Recordemos que un puntero es simplemente una variable que puede contener una dirección de memoria (de otra variable, de una función, o una dirección 'void') .

Puntero a una variable .

```
void main(void)
{
    int valor = 5;
    int *punt = &valor;
    void *paux = (void *)punt;

    printf("Valor contiene %d\n", valor);
    printf("Punt apunta a  %d\n", *punt);
    printf("Paux apunta a  %d\n", *(int *)paux);
    valor = 11;           /* equivale a : */
    *punt = 11;           /* y tambien a : */
    *(int *)paux = 11;
```



Si queremos modificar la variable `valor` valiéndonos del identificador, resulta

Lo podríamos haber hecho mediante el puntero `punt` (note la necesidad del uso del `*`)

O también con el puntero `paux` (note la necesidad del uso de la conversión de tipo o cast `(int *)`)

O mostrara utilizando el puntero `punt` (note el uso del `*`)

O la variable `valor` usando el puntero `paux` (note el uso del `*(int *)`)

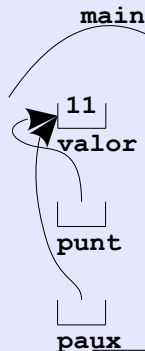
Punteros

Recordemos que un puntero es simplemente una variable que puede contener una dirección de memoria (de otra variable, de una función, o una dirección 'void') .

Puntero a una variable .

```
void main(void)
{
    int valor = 5;
    int *punt = &valor;
    void *paux = (void *)punt;

    printf("Valor contiene %d\n", valor);
    printf("Punt apunta a  %d\n", *punt);
    printf("Paux apunta a  %d\n", *(int *)paux);
    valor = 11;          /* equivale a : */
    *punt = 11;          /* y tambien a : */
    *(int *)paux = 11;
    /* las tres dan el mismo resultado */
}
```



Si queremos modificar la variable **valor** valiéndonos del identificador, resulta

Lo podríamos haber hecho mediante el puntero **punt** (note la necesidad del uso del *****)

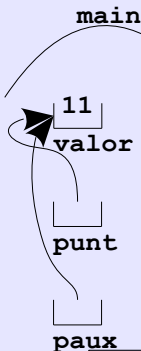
O también con el puntero **paux** (note la necesidad del uso de la conversión de tipo o cast **(int *)**)

¿Nota que las tres asignaciones dan por resultado la misma asignación de 11 a la variable **valor**?

Punteros

Recordemos que un puntero es simplemente una variable que puede contener una dirección de memoria (de otra variable, de una función, o una dirección 'void') .

Puntero a una variable .



```
void main(void)
{
    int valor = 5;
    int *punt = &valor;
    void *paux = (void *)punt;

    printf("Valor contiene %d\n", valor);
    printf("Punt apunta a  %d\n", *punt);
    printf("Paux apunta a  %d\n", *(int *)paux);
    valor = 11;           /* equivale a : */
    *punt = 11;           /* y tambien a : */
    *(int *)paux = 11;
                          /* las tres dan el mismo resultado */
}
```

Si queremos modificar la variable `valor` valiéndonos del identificador, resulta

Tenga en cuenta que este sólo es un ejemplo didáctico acerca del uso de punteros .

Es ilógico que alguien declare una variable y un puntero (o más) a esa variable para tratarla .

O mostraria utilizando el puntero `punt` (note el uso del `*`)

O la variable `valor` usando el puntero `paux` (note el uso del `*(int *)`)

Punteros

Recordemos que un puntero es simplemente una variable que puede contener una dirección de memoria (de otra variable, de una función, o una dirección 'void') .

Puntero a un array .

```
void main(void)
{
```

main



Un array no es otra cosa que un conjunto de valores de un tipo de dato arbitrario, para este ejemplo, consideraremos un array de enteros `char` .

Recordemos que un puntero es simplemente una variable que puede contener una dirección de memoria (de otra variable, de una función, o una dirección 'void') .

Puntero a un array .

main

'H' 'o' 'l' 'a' '\0'

texto

```
void main(void)
{
    char texto[] = { "Hola" };
```

Un array no es otra cosa que un conjunto de valores de un tipo de dato arbitrario, para este ejemplo, consideraremos un array de enteros `char` .

Al declarar el array `texto` preinicializado con la cadena de caracteres "Hola" el compilador determina que debe generar un array de 5 posiciones en las que almacenará los números de ASCII de esas letras, y en la última posición almacenará el carácter nulo ('`\0`')

Recordemos que un puntero es simplemente una variable que puede contener una dirección de memoria (de otra variable, de una función, o una dirección 'void') .

Puntero a un array .

main

'H' 'o' 'l' 'a' '\0'

texto

?

p

```
void main(void)
{
    char texto[] = { "Hola" };
    char *p;
```

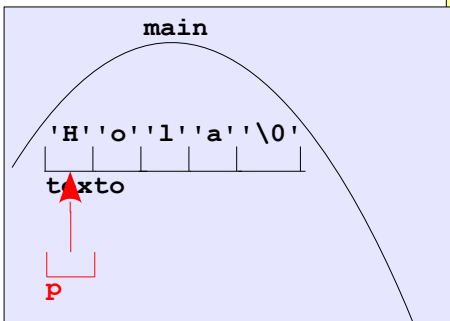
Un array no es otra cosa que un conjunto de valores de un tipo de dato arbitrario, para este ejemplo, consideraremos un array de enteros `char` .

Al declarar el array `texto` preinicializado con la cadena de caracteres "Hola" el compilador determina que debe generar un array de 5 posiciones en las que almacenará los números de ASCII de esas letras, y en la última posición almacenará el carácter nulo (`'\0'`)

Se declara además la variable puntero `p` la que por no estar inicializada contendrá alguna dirección de memoria no determinada

Recordemos que un puntero es simplemente una variable que puede contener una dirección de memoria (de otra variable, de una función, o una dirección 'void') .

Puntero a un array .



```
void main(void)
{
    char texto[] = { "Hola" };
    char *p;

    p = texto;
```

Un array no es otra cosa que un conjunto de valores de un tipo de dato arbitrario, para este ejemplo, consideraremos un array de enteros `char` .

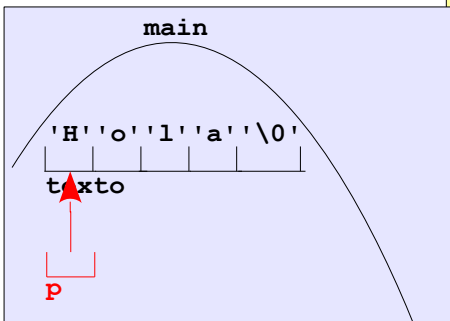
Al declarar el array `texto` preinicializado con la cadena de caracteres `"Hola"` el compilador determina que debe generar un array de 5 posiciones en las que almacenará los números de ASCII de esas letras, y en la última posición almacenará el carácter nulo (`'\0'`)

Se declara además la variable puntero `p` la que por no estar inicializada contendrá alguna dirección de memoria no determinada

Al inicializar la variable `p` con el identificador `texto`, o sea, con la dirección de comienzo del array (`p` apunta a `texto`)

Recordemos que un puntero es simplemente una variable que puede contener una dirección de memoria (de otra variable, de una función, o una dirección 'void') .

Puntero a un array .



```
void main(void)
{
    char texto[] = { "Hola" };
    char *p;

    p = texto;
    while(*p != '\0')
    {

    }
}
```

Comienza un ciclo repetitivo en que se evalúa si no se terminó la cadena (mientras no se apunta al carácter nulo)

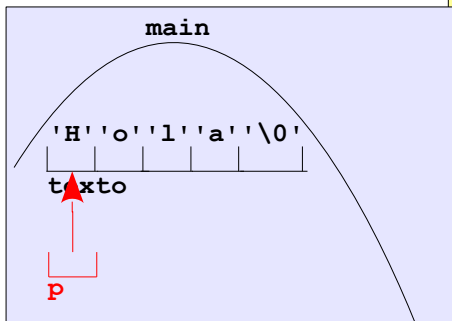
Al declarar el array `texto` preinicializado con la cadena de caracteres "Hola" el compilador determina que debe generar un array de 5 posiciones en las que almacenará los números de ASCII de esas letras, y en la última posición almacenará el carácter nulo (`'\0'`)

Se declara además la variable puntero `p` la que por no estar inicializada contendrá alguna dirección de memoria no determinada

Al inicializar la variable `p` con el identificador `texto`, o sea, con la dirección de comienzo del array (`p` apunta a `texto`)

Recordemos que un puntero es simplemente una variable que puede contener una dirección de memoria (de otra variable, de una función, o una dirección 'void') .

Puntero a un array .



```
void main(void)
{
    char texto[] = { "Hola" };
    char *p;

    p = texto;
    while(*p != '\0')
    {
        printf("%c", *p);
    }
}
```

Comienza un ciclo repetitivo en que se evalúa si no se terminó la cadena (mientras no se apunta al carácter nulo)

Muestra el carácter apuntado por `p`

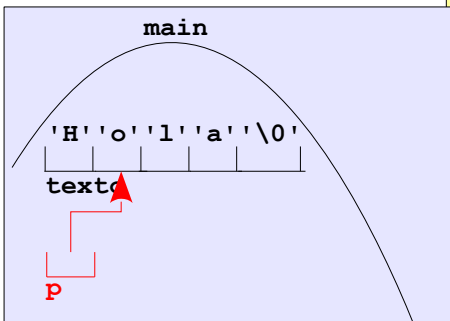
Se incrementa el puntero `p` para que apunte a la siguiente posición en la que almacenará los números de ASCII de esas letras, y en la última posición almacenará el carácter nulo (`'\0'`)

Se declara además la variable puntero `p` la que por no estar inicializada contendrá alguna dirección de memoria no determinada

Al inicializar la variable `p` con el identificador `texto`, o sea, con la dirección de comienzo del array (`p` apunta a `texto`)

Recordemos que un puntero es simplemente una variable que puede contener una dirección de memoria (de otra variable, de una función, o una dirección 'void') .

Puntero a un array .



```
void main(void)
{
    char texto[] = { "Hola" };
    char *p;

    p = texto;
    while(*p != '\0')
    {
        printf("%c", *p);
        p++;
    }
}
```

Comienza un ciclo repetitivo en que se evalúa si no se terminó la cadena (mientras no se apunta al carácter nulo)

Muestra el carácter apuntado por `p`, es decir, la 'H'

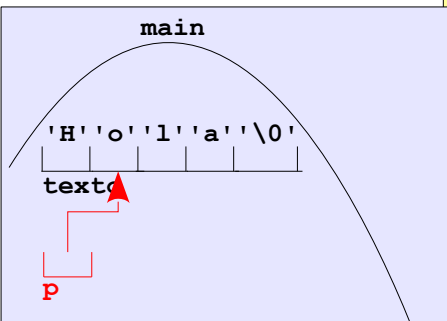
Incrementa la variable `p` para que apunte al siguiente carácter ('o') de `texto`
posición almacenará el carácter nulo ('\0')

Se declara además la variable puntero `p` la que por no estar inicializada contendrá alguna dirección de memoria no determinada

Al inicializar la variable `p` con el identificador `texto`, o sea, con la dirección de comienzo del array (`p` apunta a `texto`)

Recordemos que un puntero es simplemente una variable que puede contener una dirección de memoria (de otra variable, de una función, o una dirección 'void') .

Puntero a un array .



```
void main(void)
{
    char texto[] = { "Hola" };
    char *p;

    p = texto;
    while(*p != '\0')
    {
        printf("%c", *p);
        p++;
    }
}
```

Comienza un ciclo repetitivo en que se evalúa si no se terminó la cadena (mientras no se apunta al carácter nulo)

Muestra el carácter apuntado por `p`, es decir, la 'H'

Incrementa la variable `p` para que apunte al siguiente carácter ('o') de `texto`

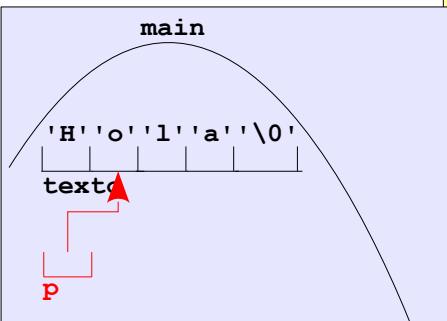
Evalúa nuevamente si no se terminó la cadena (mientras no se apunta al carácter nulo)

contendrá alguna dirección de memoria no determinada

Al inicializar la variable `p` con el identificador `texto`, o sea, con la dirección de comienzo del array (`p` apunta a `texto`)

Recordemos que un puntero es simplemente una variable que puede contener una dirección de memoria (de otra variable, de una función, o una dirección 'void') .

Puntero a un array .



```
void main(void)
{
    char texto[] = { "Hola" };
    char *p;

    p = texto;
    while(*p != '\0')
    {
        printf("%c", *p);
        p++;
    }
}
```

Comienza un ciclo repetitivo en que se evalúa si no se terminó la cadena (mientras no se apunta al carácter nulo)

Muestra el carácter apuntado por `p`, es decir, la `'H'`

Incrementa la variable `p` para que apunte al siguiente carácter (`'o'`) de `texto`

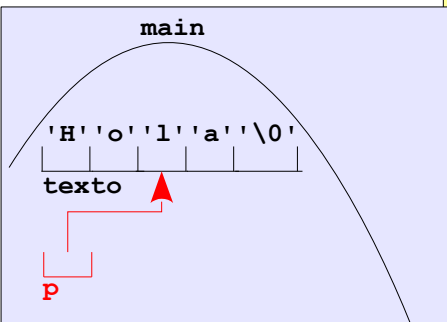
Evalúa nuevamente si no se terminó la cadena (mientras no se apunta al carácter nulo)

Muestra el carácter apuntado por `p`, es decir, la `'o'`

Al inicializar la variable `p` con el identificador `texto`, o sea, con la dirección de comienzo del array (`p` apunta a `texto`)

Recordemos que un puntero es simplemente una variable que puede contener una dirección de memoria (de otra variable, de una función, o una dirección 'void') .

Puntero a un array .



```
void main(void)
{
    char texto[] = { "Hola" };
    char *p;

    p = texto;
    while(*p != '\0')
    {
        printf("%c", *p);
        p++;
    }
}
```

Comienza un ciclo repetitivo en que se evalúa si no se terminó la cadena (mientras no se apunta al carácter nulo)

Muestra el carácter apuntado por `p`, es decir, la `'H'`

Incrementa la variable `p` para que apunte al siguiente carácter (`'o'`) de `texto`

Evalúa nuevamente si no se terminó la cadena (mientras no se apunta al carácter nulo)

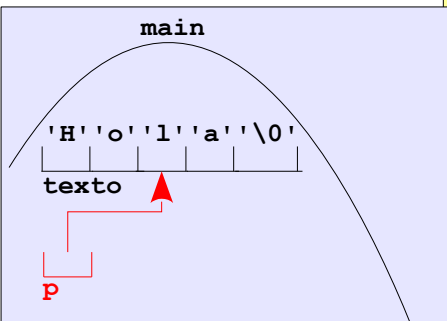
Muestra el carácter apuntado por `p`, es decir, la `'o'`

Incrementa la variable `p` para que apunte al siguiente carácter (`'l'`) de `texto`

de comienzo del array (`p` apunta a `texto`)

Recordemos que un puntero es simplemente una variable que puede contener una dirección de memoria (de otra variable, de una función, o una dirección 'void') .

Puntero a un array .



```
void main(void)
{
    char texto[] = { "Hola" };
    char *p;

    p = texto;
    while(*p != '\0')
    {
        printf("%c", *p);
        p++;
    }
}
```

Comienza un ciclo repetitivo en que se evalúa si no se terminó la cadena (mientras no se apunta al carácter nulo)

Muestra el carácter apuntado por `p`, es decir, la `'H'`

Incrementa la variable `p` para que apunte al siguiente carácter (`'o'`) de `texto`

Evalúa nuevamente si no se terminó la cadena (mientras no se apunta al carácter nulo)

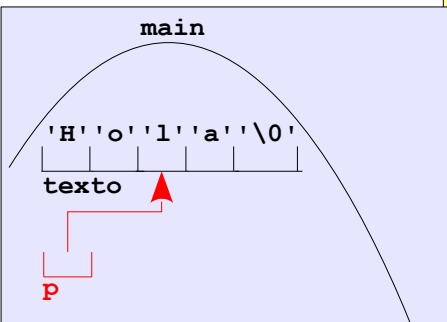
Muestra el carácter apuntado por `p`, es decir, la `'o'`

Incrementa la variable `p` para que apunte al siguiente carácter (`'l'`) de `texto`

Evalúa nuevamente si no se terminó la cadena

Recordemos que un puntero es simplemente una variable que puede contener una dirección de memoria (de otra variable, de una función, o una dirección 'void') .

Puntero a un array .



```
void main(void)
{
    char texto[] = { "Hola" };
    char *p;

    p = texto;
    while(*p != '\0')
    {
        printf("%c", *p);
        p++;
    }
}
```

Muestra el carácter apuntado por `p`, es decir, la 'l'

Muestra el carácter apuntado por `p`, es decir, la 'H'

Incrementa la variable `p` para que apunte al siguiente carácter ('o') de `texto`

Evalúa nuevamente si no se terminó la cadena (mientras no se apunta al carácter nulo)

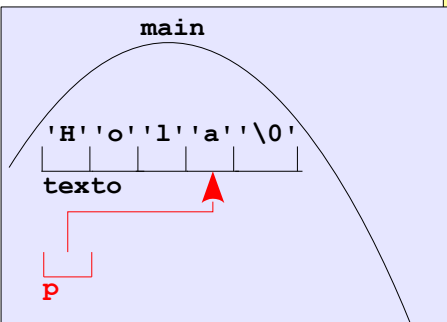
Muestra el carácter apuntado por `p`, es decir, la 'o'

Incrementa la variable `p` para que apunte al siguiente carácter ('l') de `texto`

Evalúa nuevamente si no se terminó la cadena

Recordemos que un puntero es simplemente una variable que puede contener una dirección de memoria (de otra variable, de una función, o una dirección 'void') .

Puntero a un array .



```
void main(void)
{
    char texto[] = { "Hola" };
    char *p;

    p = texto;
    while(*p != '\0')
    {
        printf("%c", *p);
        p++;
    }
}
```

Muestra el carácter apuntado por `p`, es decir, la `'l'`

Incrementa la variable `p` para que apunte al siguiente carácter (`'a'`) de `texto`

Incrementa la variable `p` para que apunte al siguiente carácter (`'o'`) de `texto`

Evalúa nuevamente si no se terminó la cadena (mientras no se apunta al carácter nulo)

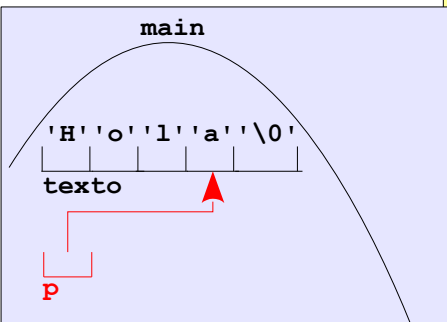
Muestra el carácter apuntado por `p`, es decir, la `'o'`

Incrementa la variable `p` para que apunte al siguiente carácter (`'l'`) de `texto`

Evalúa nuevamente si no se terminó la cadena

Recordemos que un puntero es simplemente una variable que puede contener una dirección de memoria (de otra variable, de una función, o una dirección 'void') .

Puntero a un array .



```
void main(void)
{
    char texto[] = { "Hola" };
    char *p;

    p = texto;
    while(*p != '\0')
    {
        printf("%c", *p);
        p++;
    }
}
```

Muestra el carácter apuntado por **p**, es decir, la 'H'

Incrementa la variable **p** para que apunte al siguiente carácter ('o') de **texto**

Evalúa nuevamente si no se terminó la cadena

Evalúa nuevamente si no se terminó la cadena (mientras no se apunta al carácter nulo)

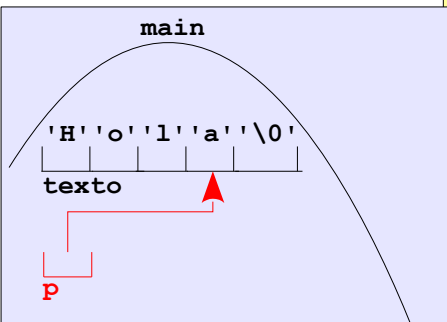
Muestra el carácter apuntado por **p**, es decir, la 'o'

Incrementa la variable **p** para que apunte al siguiente carácter ('l') de **texto**

Evalúa nuevamente si no se terminó la cadena

Recordemos que un puntero es simplemente una variable que puede contener una dirección de memoria (de otra variable, de una función, o una dirección 'void') .

Puntero a un array .



```
void main(void)
{
    char texto[] = { "Hola" };
    char *p;

    p = texto;
    while(*p != '\0')
    {
        printf("%c", *p);
        p++;
    }
}
```

Muestra el carácter apuntado por `p`, es decir, la `'l'`

Incrementa la variable `p` para que apunte al siguiente carácter (`'a'`) de `texto`

Evalúa nuevamente si no se terminó la cadena

Muestra el carácter apuntado por `p`, es decir, la `'a'`

carácter nulo)

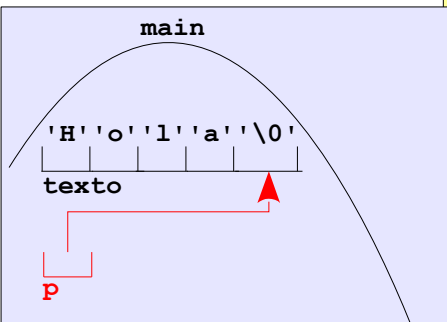
Muestra el carácter apuntado por `p`, es decir, la `'o'`

Incrementa la variable `p` para que apunte al siguiente carácter (`'l'`) de `texto`

Evalúa nuevamente si no se terminó la cadena

Recordemos que un puntero es simplemente una variable que puede contener una dirección de memoria (de otra variable, de una función, o una dirección 'void') .

Puntero a un array .



```
void main(void)
{
    char texto[] = { "Hola" };
    char *p;

    p = texto;
    while(*p != '\0')
    {
        printf("%c", *p);
        p++;
    }
}
```

Muestra el carácter apuntado por **p**, es decir, la 'H'

Incrementa la variable **p** para que apunte al siguiente carácter ('o') de **texto**

Evalúa nuevamente si no se terminó la cadena

Muestra el carácter apuntado por **p**, es decir, la 'o'

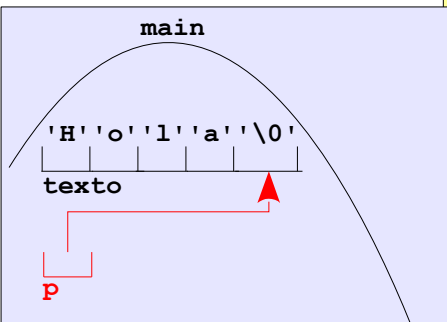
Incrementa la variable **p** para que apunte al siguiente carácter ('\0') de **texto**

Incrementa la variable **p** para que apunte al siguiente carácter ('l') de **texto**

Evalúa nuevamente si no se terminó la cadena

Recordemos que un puntero es simplemente una variable que puede contener una dirección de memoria (de otra variable, de una función, o una dirección 'void') .

Puntero a un array .



```
void main(void)
{
    char texto[] = { "Hola" };
    char *p;

    p = texto;
    while(*p != '\0')
    {
        printf("%c", *p);
        p++;
    }
}
```

Muestra el carácter apuntado por `p`, es decir, la 'l'

Incrementa la variable `p` para que apunte al siguiente carácter ('a') de `texto`

Evalúa nuevamente si no se terminó la cadena

Muestra el carácter apuntado por `p`, es decir, la 'a'

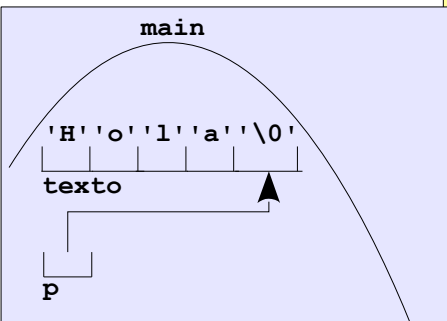
Incrementa la variable `p` para que apunte al siguiente carácter ('\0') de `texto`

Evalúa nuevamente si no se terminó la cadena

Evalúa nuevamente si no se terminó la cadena

Recordemos que un puntero es simplemente una variable que puede contener una dirección de memoria (de otra variable, de una función, o una dirección 'void') .

Puntero a un array .



```
void main(void)
{
    char texto[] = { "Hola" };
    char *p;

    p = texto;
    while(*p != '\0')
    {
        printf("%c", *p);
        p++;
    }
}
```

Muestra el carácter apuntado por `p`, es decir, la `'l'`

Incrementa la variable `p` para que apunte al siguiente carácter (`'a'`) de `texto`

Evalúa nuevamente si no se terminó la cadena

Muestra el carácter apuntado por `p`, es decir, la `'a'`

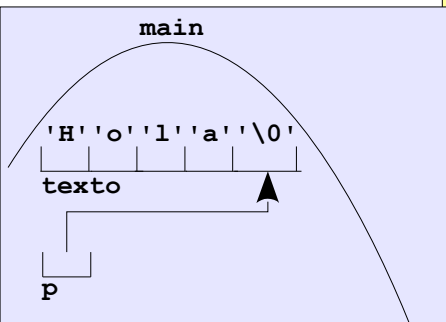
Incrementa la variable `p` para que apunte al siguiente carácter (`'\0'`) de `texto`

Evalúa nuevamente si no se terminó la cadena

Como se alcanzó el carácter nulo, sale del ciclo, terminando este ejemplo

Recordemos que un puntero es simplemente una variable que puede contener una dirección de memoria (de otra variable, de una función, o una dirección 'void') .

Puntero a un array .



```
void main(void)
{
    char texto[] = { "Hola" };
    char *p;

    p = texto;
    while(*p != '\0')
    {
        printf("%c", *p);
        p++;
    }
}
```

Muestra el carácter apuntado por **p**, es decir, la 'l'

Incrementa la variable **p** para que apunte al siguiente carácter ('a') de **texto**

Tenga en cuenta que este sólo es un ejemplo didáctico acerca del uso de punteros .

Es ilógico que alguien declare una variable y un puntero (o más) a esa variable para tratarla .

Evalúa nuevamente si no se terminó la cadena

Como se alcanzó el carácter nulo, sale del ciclo, terminando este ejemplo

Recordemos que un puntero es simplemente una variable que puede contener una dirección de memoria (de otra variable, de una función, o una dirección 'void') .

Arrays de punteros .

Recordemos que un puntero es simplemente una variable que puede contener una dirección de memoria (de otra variable, de una función, o una dirección 'void') .

Arrays de punteros .

Tal y como usted ya sabe, el **Lenguaje C**, le permite declarar y tratar arrays de cualquiera de los tipos de información que sean necesarios .

Recordemos que un puntero es simplemente una variable que puede contener una dirección de memoria (de otra variable, de una función, o una dirección 'void') .

Arrays de punteros .

Tal y como usted ya sabe, el **Lenguaje C**, le permite declarar y tratar arrays de cualquiera de los tipos de información que sean necesarios .

Se puede tratar con arrays de enteros o flotantes, arrays de estructuras, arrays de punteros, arrays de punteros a funciones, etc. .

Recordemos que un puntero es simplemente una variable que puede contener una dirección de memoria (de otra variable, de una función, o una dirección 'void') .

Arrays de punteros .

Tal y como usted ya sabe, el **Lenguaje C**, le permite declarar y tratar arrays de cualquiera de los tipos de información que sean necesarios .

Se puede tratar con arrays de enteros o flotantes, arrays de estructuras, arrays de punteros, arrays de punteros a funciones, etc. .

Veremos a continuación un ejemplo en que se muestra el uso de un array, preinicializado, de punteros a **char** .

Recordemos que un puntero es simplemente una variable que puede contener una dirección de memoria (de otra variable, de una función, o una dirección 'void') .

Arrays de punteros .

```
void main(void)
{
    char *nombres[3];
}
```

main

? ? ?
nombres

Esta es la declaración de un array de 3 posiciones, donde cada una de ellas es un puntero a char

Recordemos que un puntero es simplemente una variable que puede contener una dirección de memoria (de otra variable, de una función, o una dirección 'void') .

Arrays de punteros .

```
void main(void)
{
    char *nombres[3];
}
```

main



Esta es la declaración de un array de 3 posiciones, donde cada una de ellas es un puntero a **char**

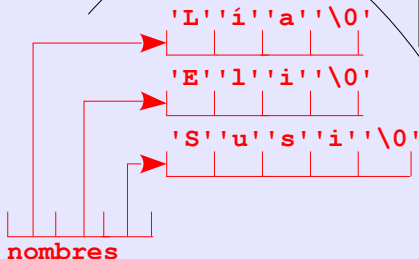
Por no estar preinicializado, cada una de sus posiciones contendrá alguna dirección indeterminada

Recordemos que un puntero es simplemente una variable que puede contener una dirección de memoria (de otra variable, de una función, o una dirección 'void') .

Arrays de punteros .

```
void main(void)
{
    char *nombres[3] = { { "Lia" }, { "Eli" }, { "Susi" } };
}
```

main



Esta es la declaración de un array de 3 posiciones, donde cada una de ellas es un puntero a **char**

Por no estar preinicializado, cada una de sus posiciones contendrá alguna dirección indeterminada

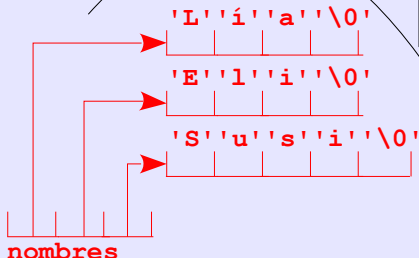
Si lo declaramos preinicializado, lo que sucede es que a cada posición del array de punteros se le asigna en qué dirección de memoria se encuentra cada cadena

Recordemos que un puntero es simplemente una variable que puede contener una dirección de memoria (de otra variable, de una función, o una dirección 'void') .

Arrays de punteros .

```
void main(void)
{
    char *nombres[ ] = { { "Lia" }, { "Eli" }, { "Susi" } };
}
```

main



Esta es la declaración de un array de 3 posiciones, donde cada una de ellas es un puntero a **char**

Por no estar preinicializado, cada una de sus posiciones contendrá alguna dirección indeterminada

Si lo declaramos preinicializado, lo que sucede es que a cada posición del array de punteros se le asigna en qué dirección de memoria se encuentra cada cadena

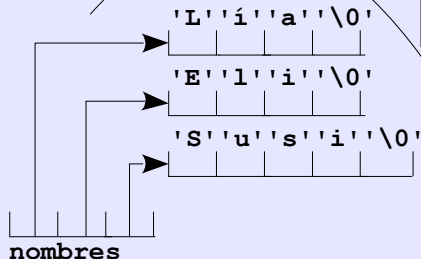
Como siempre, si no se indica la cantidad de posiciones del array, el compilador lo resolverá, generando un array, en este caso, de tres punteros a **char**, porque se lo preinicializa con las direcciones de comienzo de tres array de **char**

Recordemos que un puntero es simplemente una variable que puede contener una dirección de memoria (de otra variable, de una función, o una dirección 'void') .

Arrays de punteros .

```
void main(void)
{
    char *nombres[ ] = { { "Lia" }, { "Eli" }, { "Susi" } };
    int ciclo;
```

main



?
ciclo

Se declara una variable entera (**ciclo**) para poder recorrer las distintas posiciones del array, la que por no estar inicializada, contendrá algún valor entero

Si no declaramos preinicializado, lo que sucede es que a cada posición del array de punteros se le asigna en qué dirección de memoria se encuentra cada cadena

Como siempre, si no se indica la cantidad de posiciones del array, el compilador lo resolverá, generando un array, en este caso, de tres punteros a **char**, porque se lo preinicializa con las direcciones de comienzo de tres array de **char**

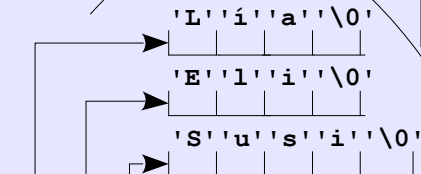
Recordemos que un puntero es simplemente una variable que puede contener una dirección de memoria (de otra variable, de una función, o una dirección 'void') .

Arrays de punteros .

```
void main(void)
{
    char *nombres[ ] = { { "Lia" }, { "Eli" }, { "Susi" } };
    int  ciclo;

    for(ciclo = 0; ciclo < 3; ciclo++)
```

main



nombres

0
ciclo

Se declara una variable entera (**ciclo**) para poder recorrer las distintas posiciones del array, la que por no estar inicializada, contendrá algún valor entero

Al ejecutarse el **for**, se inicializa la variable **ciclo** con el valor **0** . y se evalúa la condición que resulta ser Verdad ('**ciclo < 3**'), con lo cual ...

Como siempre, si no se indica la cantidad de posiciones del array, el compilador lo resolverá, generando un array, en este caso, de tres punteros a **char**, porque se lo preinicializa con las direcciones de comienzo de tres array de **char**

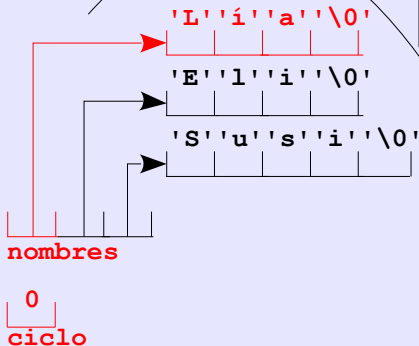
Recordemos que un puntero es simplemente una variable que puede contener una dirección de memoria (de otra variable, de una función, o una dirección 'void') .

Arrays de punteros .

```
void main(void)
{
    char *nombres[ ] = { { "Lia" }, { "Eli" }, { "Susi" } };
    int  ciclo;

    for(ciclo = 0; ciclo < 3; ciclo++)
        puts(nombres[ciclo]); /* printf("%s\n", nombres[ciclo]); */
}
```

main



Se declara una variable entera (`ciclo`) para poder recorrer las distintas posiciones del array, la que por no estar inicializada, contendrá algún valor entero

Al ejecutarse el `for`, se inicializa la variable `ciclo` con el valor 0 . y se evalúa la condición que resulta ser Verdad (`'ciclo < 3'`), con lo cual ...

... se muestra la posición 0 del array `nombres` que contiene la dirección de memoria de la primer cadena ("Lía"), por lo tanto se muestra esa cadena

`char`, porque se lo preinicializa con las direcciones de comienzo de tres array de `char`

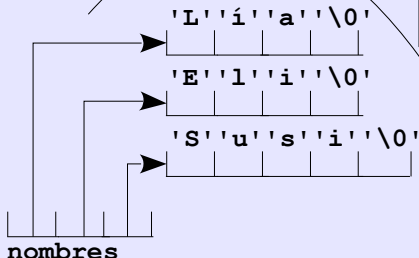
Recordemos que un puntero es simplemente una variable que puede contener una dirección de memoria (de otra variable, de una función, o una dirección 'void') .

Arrays de punteros .

```
void main(void)
{
    char *nombres[ ] = { { "Lia" }, { "Eli" }, { "Susi" } };
    int  ciclo;

    for(ciclo = 0; ciclo < 3; ciclo++)
        puts(nombres[ciclo]); /* printf("%s\n", nombres[ciclo]); */
}
```

main



1
ciclo

Se declara una variable entera (**ciclo**) para poder recorrer las distintas posiciones del array, la que por no estar inicializada, contendrá algún valor entero

Al ejecutarse el **for**, se inicializa la variable **ciclo** con el valor **0** . y se evalúa la condición que resulta ser Verdad ('**ciclo < 3**'), con lo cual ...

... se muestra la posición **0** del array **nombres** que contiene la dirección de memoria de la primer cadena ("Lía"), por lo tanto se muestra esa cadena

Se incrementa la variable **ciclo**, que pasa a contener **1**, y se evalúa la condición ('**ciclo < 3**')...

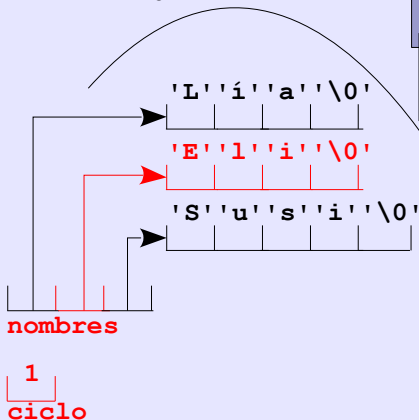
Recordemos que un puntero es simplemente una variable que puede contener una dirección de memoria (de otra variable, de una función, o una dirección 'void') .

Arrays de punteros .

```
void main(void)
{
    char *nombres[ ] = { { "Lia" }, { "Eli" }, { "Susi" } };
    int  ciclo;

    for(ciclo = 0; ciclo < 3; ciclo++)
        puts(nombres[ciclo]); /* printf("%s\n", nombres[ciclo]); */
}
```

main



... nuevamente muestra por pantalla, pero esta vez, la posición 1 de `nombres`, que como contiene la dirección de memoria en que se encuentra la cadena "Eli", esto es lo que se muestra

Al ejecutarse el `for`, se inicializa la variable `ciclo` con el valor 0 . y se evalúa la condición que resulta ser Verdad (`'ciclo < 3'`), con lo cual ...

... se muestra la posición 0 del array `nombres` que contiene la dirección de memoria de la primer cadena ("Lía"), por lo tanto se muestra esa cadena

Se incrementa la variable `ciclo`, que pasa a contener 1, y se evalúa la condición (`'ciclo < 3'`)...

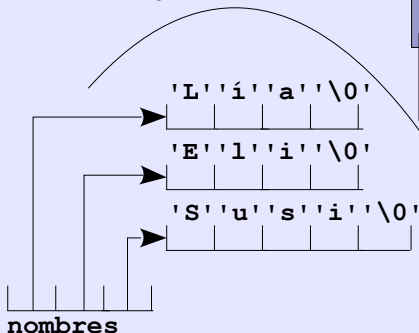
Recordemos que un puntero es simplemente una variable que puede contener una dirección de memoria (de otra variable, de una función, o una dirección 'void') .

Arrays de punteros .

```
void main(void)
{
    char *nombres[ ] = { { "Lia" }, { "Eli" }, { "Susi" } };
    int  ciclo;

    for(ciclo = 0; ciclo < 3; ciclo++)
        puts(nombres[ciclo]); /* printf("%s\n", nombres[ciclo]); */
}
```

main



2
ciclo

... nuevamente muestra por pantalla, pero esta vez, la posición 1 de **nombres**, que como contiene la dirección de memoria en que se encuentra la cadena "Eli", esto es lo que se muestra

Nuevamente ejecuta **ciclo++**, que pasa a contener 2, y se evalúa la condición ('**ciclo < 3**')...

... se muestra la posición 0 del array **nombres** que contiene la dirección de memoria de la primer cadena ("Lía"), por lo tanto se muestra esa cadena

Se incrementa la variable **ciclo**, que pasa a contener 1, y se evalúa la condición ('**ciclo < 3**')...

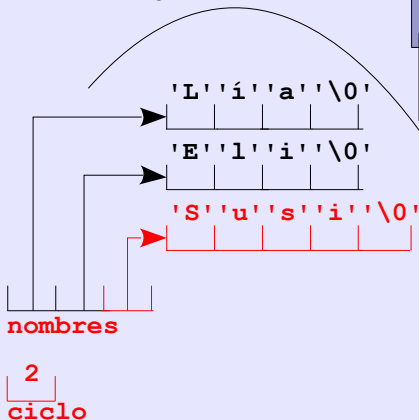
Recordemos que un puntero es simplemente una variable que puede contener una dirección de memoria (de otra variable, de una función, o una dirección 'void') .

Arrays de punteros .

```
void main(void)
{
    char *nombres[ ] = { { "Lia" }, { "Eli" }, { "Susi" } };
    int  ciclo;

    for(ciclo = 0; ciclo < 3; ciclo++)
        puts(nombres[ciclo]); /* printf("%s\n", nombres[ciclo]); */
}
```

main



... nuevamente muestra por pantalla, pero esta vez, la posición 1 de **nombres**, que como contiene la dirección de memoria en que se encuentra la cadena "Eli", esto es lo que se muestra

Nuevamente ejecuta **ciclo++**, que pasa a contener 2, y se evalúa la condición ('**ciclo < 3**')...

... nuevamente muestra por pantalla, pero esta vez, la posición 2 de **nombres**, que contiene la dirección de memoria en que se encuentra la cadena "Susi"

Se incrementa la variable **ciclo**, que pasa a contener 1, y se evalúa la condición ('**ciclo < 3**')...

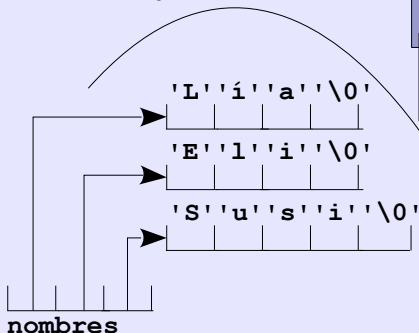
Recordemos que un puntero es simplemente una variable que puede contener una dirección de memoria (de otra variable, de una función, o una dirección 'void') .

Arrays de punteros .

```
void main(void)
{
    char *nombres[ ] = { { "Lia" }, { "Eli" }, { "Susi" } };
    int  ciclo;

    for(ciclo = 0; ciclo < 3; ciclo++)
        puts(nombres[ciclo]); /* printf("%s\n", nombres[ciclo]); */
}
```

main



... nuevamente muestra por pantalla, pero esta vez, la posición 1 de **nombres**, que como contiene la dirección de memoria en que se encuentra la cadena "Eli", esto es lo que se muestra

Nuevamente ejecuta **ciclo++**, que pasa a contener 2, y se evalúa la condición ('**ciclo < 3**')...

... nuevamente muestra por pantalla, pero esta vez, la posición 2 de **nombres**, que contiene la dirección de memoria en que se encuentra la cadena "Susi"

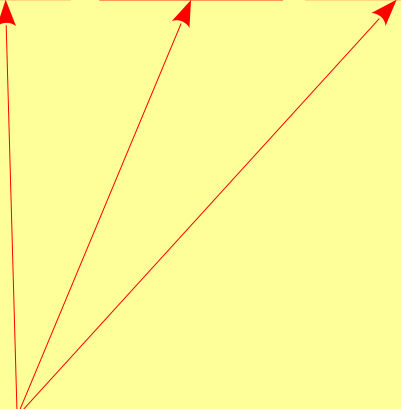
Se incrementa la variable **ciclo**, que pasa a contener 3, por lo cual termina el **for**, y el programa

Argumentos de `main`.

Argumentos de main .

main



```
void main(    )  
{  
  
}  
  

```

Igual que cualquier otra función escrita en C, la función **main** puede recibir argumentos

Argumentos de `main`.

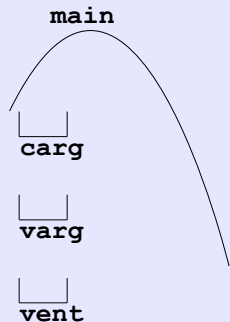
```
void main(int carg )
{

```

Igual que cualquier otra función escrita en C, la función main puede recibir argumentos

Pero tales argumentos sólo pueden ser a lo sumo tres, el primero sólo puede ser entero . . .

Argumentos de main .



```
void main(int carg, char **varg, char **vent)
{
    }
}
```

Igual que cualquier otra función escrita en C, la función **main** puede recibir argumentos

Pero tales argumentos sólo pueden ser a lo sumo tres, el primero sólo puede ser entero . . .

. . . los dos restantes, sólo pueden ser punteros a punteros a char

Argumentos de main .

```
void main(int carg, char **varg, char **vent)
{
    }
}
```

main

carg

varg

vent

Igual que cualquier otra función escrita en C, la función **main** puede recibir argumentos

Pero tales argumentos sólo pueden ser a lo sumo tres, el primero sólo puede ser entero ...

... los dos restantes, sólo pueden ser punteros a punteros a char

Supongamos que hemos escrito el programa **p1.c** y que con él hemos generado el ejecutable **copiar.exe**

Argumentos de main.

```
c:\>copiar c:\fuentes\p1.c e:\p1.c /v[Enter]
```

```
void main(int carg, char **varg, char **vent)
{
}

}
```

main

carg

varg

vent

Igual que cualquier otra función escrita en C, la función **main** puede recibir argumentos

Pero tales argumentos sólo pueden ser a lo sumo tres, el primero sólo puede ser entero ...

... los dos restantes, sólo pueden ser punteros a punteros a char

Supongamos que hemos escrito el programa **p1.c** y que con él hemos generado el ejecutable **copiar.exe**

... y que lo ejecutamos desde la línea de comando del DOS

Punteros

Argumentos de main.

```
c:\>copiar c:\fuentes\p1.c e:\p1.c /v[Enter]
```

```
void main(int carg, char **varg, char **vent)
{
}

}
```

main

4

carg

varg

vent

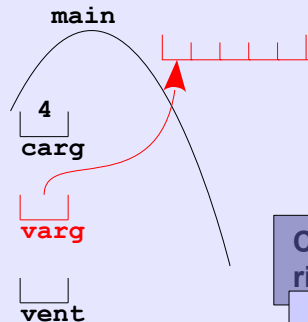
Cuando el programa comience a ejecutarse recibe en la variable `carg` de `main` el valor entero 4 (por las 4 cadenas)

Argumentos de main.

```
c:\>copiar c:\fuentes\p1.c e:\p1.c /v[Enter]
```

```
void main(int carg, char **varg, char **vent)
{

}
```



Cuando el programa comienza a ejecutarse recibe en la variable `carg` de `main` el valor entero 4 (por las 4 cadenas)

En el segundo argumento (`varg`) recibe la dirección de comienzo de un array de 5 punteros a `char`

Argumentos de main .

```
c:\>copiar c:\fuentes\p1.c e:\p1.c /v[Enter]
```

"C:\Utiles\COPIAR.EXE"

```
void main(int carg, char **varg, char **vent)
{
}

}
```

main

4

carg

varg

vent

Cuando el programa comience a ejecutarse recibe en la variable `carg` de `main` el valor entero 4 (por las 4 cadenas)

En el segundo argumento (`varg`) recibe la dirección de comienzo de un array de 5 punteros a `char`

en su primer posición, contendrá la dirección de comienzo de la primer cadena escrita en la línea de comando

Argumentos de main.

```
c:\>copiar c:\fuentes\p1.c e:\p1.c /v[Enter]
```

"C:\Utiles\COPIAR.EXE"

```
void main(int carg, char **varg, char **vent)
{
}
```

main

4

carg

varg

vent

Cuando el programa comience a ejecutarse recibe en la variable `carg` de `main` el valor entero 4 (por las 4 cadenas)

En el segundo argumento (`varg`) recibe la dirección de comienzo de un array de 5 punteros a `char`

en su primer posición, contendrá la dirección de comienzo de la primera cadena escrita en la línea de comando

Muchos compiladores C completan el nombre del ejecutable con la unidad (C:) y path (\Utiles\)...

Argumentos de main.

```
c:\>copiar c:\fuentes\p1.c e:\p1.c /v[Enter]
```

"C:\Utiles\COPIAR.EXE"

```
void main(int carg, char **varg, char **vent)
{
}
```

main

4

carg

varg

vent

Cuando el programa comience a ejecutarse recibe en la variable `carg` de `main` el valor entero 4 (por las 4 cadenas)

En el segundo argumento (`varg`) recibe la dirección de comienzo de un array de 5 punteros a `char`

en su primer posición, contendrá la dirección de comienzo de la primera cadena escrita en la línea de comando

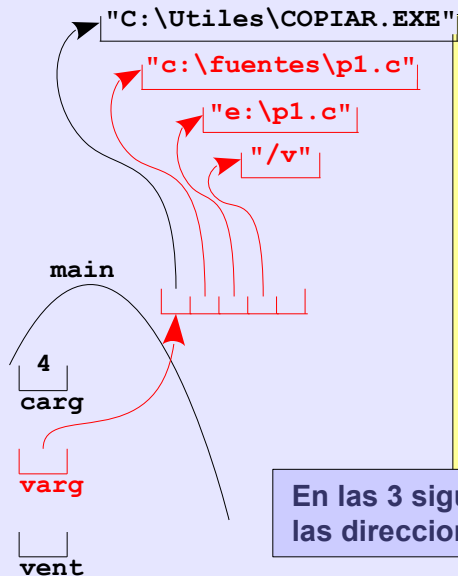
Muchos compiladores C completan el nombre del ejecutable con la unidad (`c:`) y path (`\Utiles\`)...

... agregando la extensión y dejándolo en mayúsculas

Punteros

Argumentos de main .

```
c:\>copiar c:\fuentes\p1.c e:\p1.c /v[Enter]
```

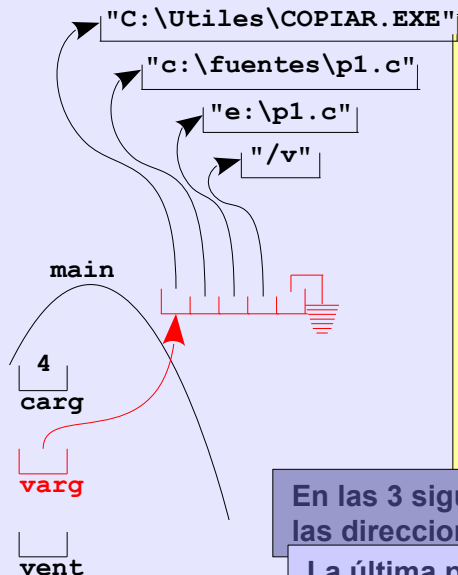


```
void main(int carg, char **varg, char **vent)
{
    ...
}
```

En las 3 siguientes posiciones, el array de punteros, tendrá las direcciones de los argumentos de la línea de comando

Argumentos de main .

```
c:\>copiar c:\fuentes\p1.c e:\p1.c /v[Enter]
```



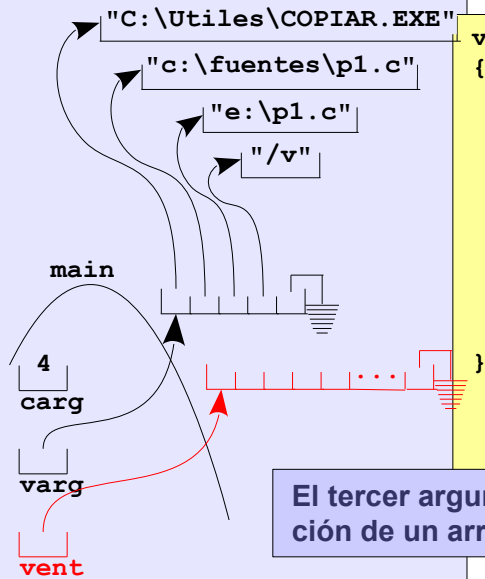
```
void main(int carg, char **varg, char **vent)
{
    ...
}
```

En las 3 siguientes posiciones, el array de punteros, tendrá las direcciones de los argumentos de la línea de comando

La última posición del array de punteros tiene la dirección de memoria `NULL`, indicando que ahí termina

Argumentos de main .

```
c:\>copiar c:\fuentes\p1.c e:\p1.c /v[Enter]
```

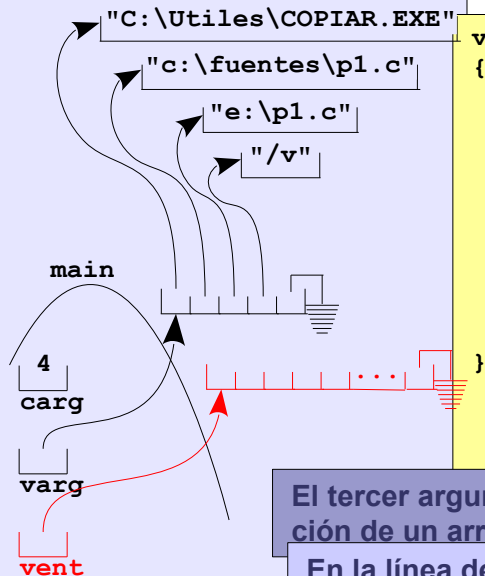


```
void main(int carg, char **varg, char **vent)
{
}
```

El tercer argumento es similar al segundo, tiene la dirección de un array de punteros a las variables de entorno

Argumentos de main .

```
c:\>copiar c:\fuentes\p1.c e:\p1.c /v[Enter]
```



```
void main(int carg, char **varg, char **vent)
{
    ...
}
```

El tercer argumento es similar al segundo, tiene la dirección de un array de punteros a las variables de entorno

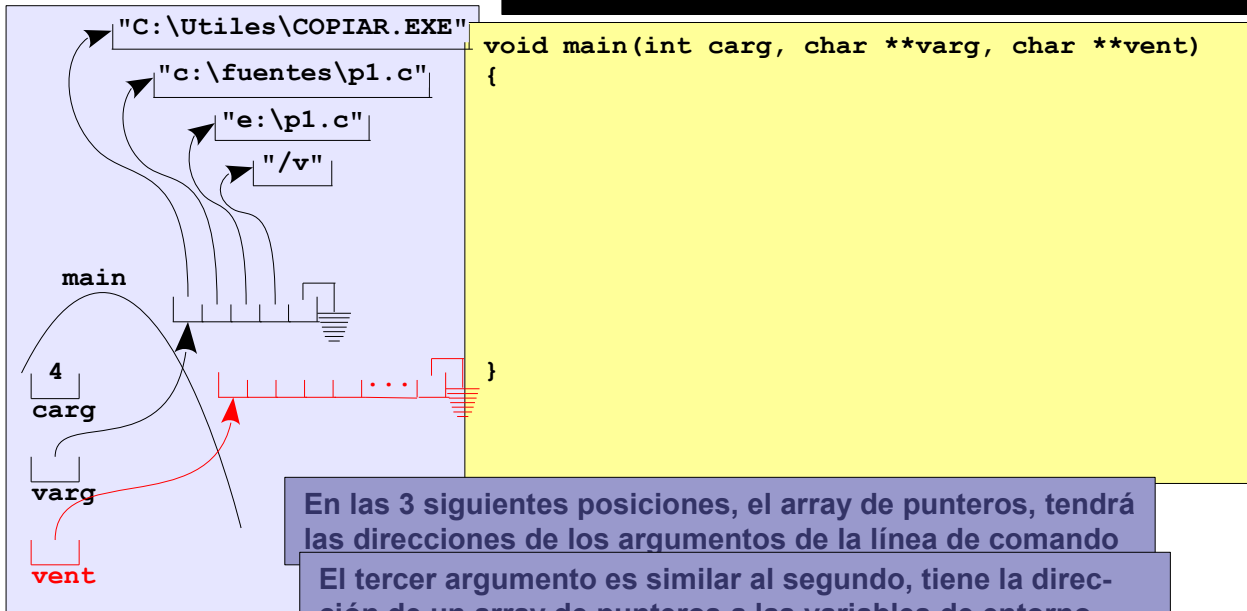
En la línea de comando, ejecute el comando `set` para ver las variables de su entorno

```
C:\ C:\WINDOWS\system32\cmd.exe
C:\>set |more
ALLUSERSPROFILE=C:\Documents and Settings\All Users
APPDATA=C:\Documents and Settings\LuisAJLopez\Application Data
CLIENTNAME=Console
CommonProgramFiles=C:\Program Files\Common Files
COMPUTERNAME=LUISLOPEZ
ComSpec=C:\WINDOWS\system32\cmd.exe
FP_NO_HOST_CHECK=NO
GETMODEL=Satellite A105
HOMEDRIVE=C:
HOMEPATH=\Documents and Settings\LuisAJLopez
include=C:\Program Files\Microsoft Visual Studio\VC98\atl\include;C:\Program Files\Microsoft Visual Studio\VC98\mfc\include;C:\Program Files\Microsoft Visual Studio\VC98\include
lib=C:\Program Files\Microsoft Visual Studio\VC98\mfc\lib;C:\Program Files\Microsoft Visual Studio\VC98\lib
LOGONSERVER=\\LUISLOPEZ
MSDevDir=C:\Program Files\Microsoft Visual Studio\Common\MSDev98
NUMBER_OF_PROCESSORS=1
OS=Windows_NT
Path=C:\WINDOWS\system32;C:\WINDOWS;C:\WINDOWS\System32\Wbem;C:\Program Files\ATI Technologies\ATI Control Panel;C:\Program Files\doxygen\bin;C:\PROGRAM*1\ATT\Graphviz\bin;C:\Program Files\Microsoft Visual Studio\Common\Tools\WinNT;C:\Program Files\Microsoft Visual Studio\Common\MSDev98\Bin;C:\Program Files\Microsoft Visual Studio\Common\Tools;C:\Program Files\Microsoft Visual Studio\VC98\bin;c:\tc
-- More --
```

En la línea de comando, ejecute el comando `set` para ver las variables de su entorno

Argumentos de main.

```
c:\>copiar c:\fuentes\p1.c e:\p1.c /v[Enter]
```



```
void main(int carg, char **varg, char **vent)
{
    ...
}
```

En las 3 siguientes posiciones, el array de punteros, tendrá las direcciones de los argumentos de la línea de comando

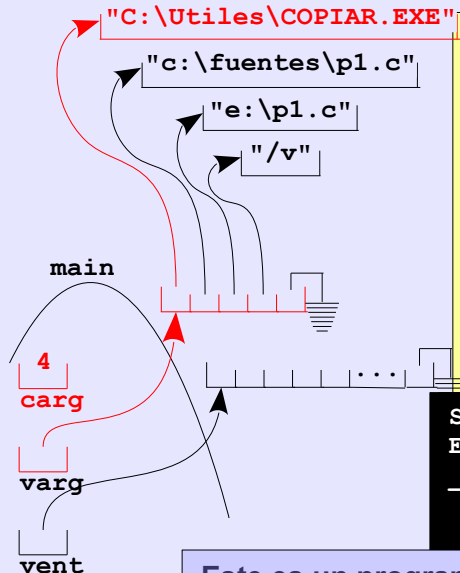
El tercer argumento es similar al segundo, tiene la dirección de un array de punteros a las variables de entorno

En la línea de comando, ejecute el comando `set` para ver las variables de su entorno

No ejemplificaremos mayormente acerca de este tercer argumento ya que excede los alcances de la materia y el uso que le podemos dar en ella

Argumentos de main .

```
c:\>copiar c:\fuentes\p1.c e:\p1.c /v[Enter]
```



```
void main(int carg, char **varg, char **vent)
{
    printf("Se recibieron %d argumentos en la"
           " línea de comando\n",
           carg - 1);

    printf("El nombre del programa es %s\n",
           *varg);
}
```

```
Se recibieron 3 argumentos en la línea de comando
El nombre del programa es C:\Utiles\COPIAR.EXE
```

—

Este es un programa muy sencillo que informa cuántos argumentos recibe, y muestre el nombre del programa . . .

Argumentos de main.

```
c:\>copiar c:\fuentes\p1.c e:\p1.c /v[Enter]
```

"C:\Utiles\COPIAR.EXE"

"c:\fuentes\p1.c"

"e:\p1.c"

"/v"

main

4

carg

varg

vent

```
void main(int carg, char **varg, char **vent)
{
    /* ... */
}
```

Este es un programa muy sencillo que informa cuántos argumentos recibe y muestre el nombre del programa.

La salida por pantalla de la continuación del mismo (o de otro programa), es

Argumentos de main.

```
c:\>copiar c:\fuentes\p1.c e:\p1.c /v[Enter]
```

"C:\Utiles\COPIAR.EXE"

"c:\fuentes\p1.c"

"e:\p1.c"

"/v"

main

4

carg

varg

vent

```
void main(int carg, char **varg, char **vent)
{
    /* ... */
    puts("Se recibieron los siguientes"
        " argumentos");
}
```

Se recibieron los siguientes argumentos :

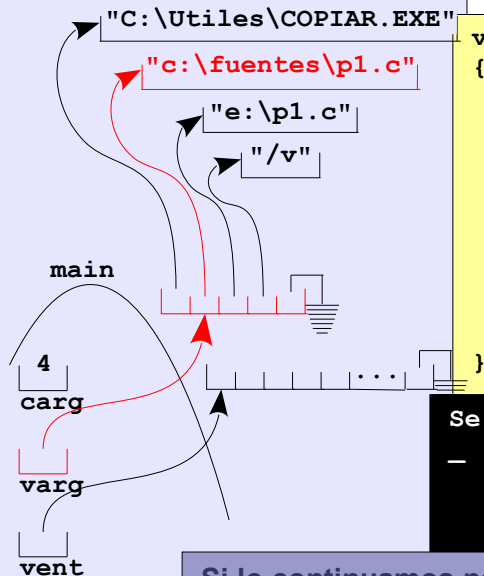
—

Si lo continuamos para que muestre los argumentos que recibe, comienza con un mensaje por pantalla

Argumentos de main.

```
c:\>copiar c:\fuentes\p1.c e:\p1.c /v[Enter]
```

```
void main(int carg, char **varg, char **vent)
{
    /* ... */
    puts("Se recibieron los siguientes"
        " argumentos");
    varg++;
}
```



Se recibieron los siguientes argumentos :

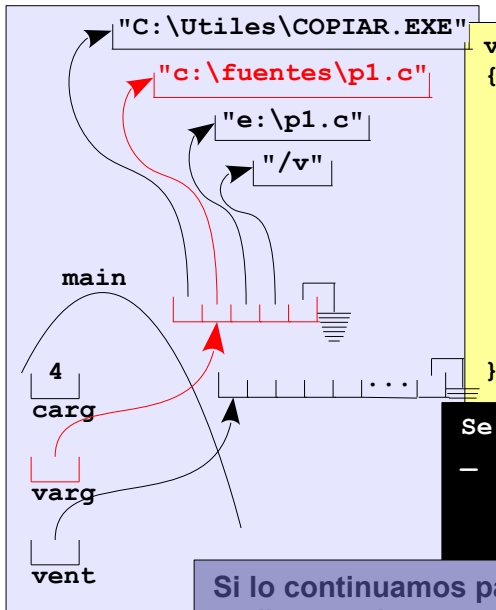
—

Si lo continuamos para que muestre los argumentos que recibe, comienza con un mensaje por pantalla

al incrementar `varg` deja de tener la posición del primer puntero del array para apuntar al segundo

Argumentos de main.

```
c:\>copiar c:\fuentes\p1.c e:\p1.c /v[Enter]
```



```
void main(int carg, char **varg, char **vent)
{
    /* ... */
    puts("Se recibieron los siguientes"
        " argumentos");
    varg++;
    while(*varg)
    {

    }
}
```

Se recibieron los siguientes argumentos :

—

Si lo continuamos para que muestre los argumentos que recibe, comienza con un mensaje por pantalla

al incrementar `varg` deja de tener la posición del primer puntero del array para apuntar al segundo

evalúa la condición del `while` que le da que es verdad, con lo que entra al ciclo repetitivo

Argumentos de main.

```
c:\>copiar c:\fuentes\p1.c e:\p1.c /v[Enter]
```

```
void main(int carg, char **varg, char **vent)
{
    /* ... */
    puts("Se recibieron los siguientes"
        " argumentos");
    varg++;
    while(*varg)
    {
        puts(*varg);
    }
}
```

Se recibieron los siguientes argumentos :
c:\fuentes\p1.c

—

Si muestra lo que apunta el puntero, que es la segunda posición del array en la que está la dirección del 1er argumento al incrementar **varg** deja de tener la posición del primer puntero del array para apuntar al segundo

evalúa la condición del **while** que le da que es verdad, con lo que entra al ciclo repetitivo

Argumentos de main.

```
c:\>copiar c:\fuentes\p1.c e:\p1.c /v[Enter]
```

```
void main(int carg, char **varg, char **vent)
{
    /* ... */
    puts("Se recibieron los siguientes"
        " argumentos");
    varg++;
    while(*varg)
    {
        puts(*varg);
        varg++;
    }
}
```

Se recibieron los siguientes argumentos :
c:\fuentes\p1.c

—

Si Muestra lo que apunta el puntero, que es la segunda posición del array en la que está la dirección del 1er argumento

al Incrementa el puntero, con lo que pasa a apuntar a la tercer posición del array de punteros

evalúa la condición del `while` que le da que es verdad, con lo que entra al ciclo repetitivo

Argumentos de main.

```
c:\>copiar c:\fuentes\p1.c e:\p1.c /v[Enter]
```

```
void main(int carg, char **varg, char **vent)
{
    /* ... */
    puts("Se recibieron los siguientes"
        " argumentos");
    varg++;
    while(*varg)
    {
        puts(*varg);
        varg++;
    }
}
```

Se recibieron los siguientes argumentos :
c:\fuentes\p1.c

—

Si Muestra lo que apunta el puntero, que es la segunda posición del array en la que está la dirección del 1er argumento

al Incrementa el puntero, con lo que pasa a apuntar a la tercera posición del array de punteros

evalúa nuevamente la condición del `while` que le da que lo es verdad, con lo que permanece en el ciclo repetitivo

Argumentos de main.

```
c:\>copiar c:\fuentes\p1.c e:\p1.c /v[Enter]
```

```
void main(int carg, char **varg, char **vent)
{
    /* ... */
    puts("Se recibieron los siguientes"
        " argumentos");
    varg++;
    while(*varg)
    {
        puts(*varg);
        varg++;
    }
}
```

Se recibieron los siguientes argumentos :
c:\fuentes\p1.c
e:\p1.c
—

Si M Muestra lo que apunta el puntero, que es la segunda posi-
re ci ción del array en la que está la dirección del 2º argumento
al Incrementa el puntero, con lo que pasa a apuntar a la
pu tercer posición del array de punteros
ev evalúa nuevamente la condición del `while` que le da que
lo es verdad, con lo que permanece en el ciclo repetitivo

Argumentos de main.

```
c:\>copiar c:\fuentes\p1.c e:\p1.c /v[Enter]
```

```
void main(int carg, char **varg, char **vent)
{
    /* ... */
    puts("Se recibieron los siguientes"
        " argumentos");
    varg++;
    while(*varg)
    {
        puts(*varg);
        varg++;
    }
}
```

```
Se recibieron los siguientes argumentos :
c:\fuentes\p1.c
e:\p1.c
-
```

Si M Muestra lo que apunta el puntero, que es la segunda posición del array en la que está la dirección del 2º argumento

al In Incrementa el puntero, con lo que pasa a apuntar a la cuarta posición del array de punteros

ev evalúa nuevamente la condición del while que le da que lo es verdad, con lo que permanece en el ciclo repetitivo

Argumentos de main.

```
c:\>copiar c:\fuentes\p1.c e:\p1.c /v[Enter]
```

```
void main(int carg, char **varg, char **vent)
{
    /* ... */
    puts("Se recibieron los siguientes"
        " argumentos");
    varg++;
    while(*varg)
    {
        puts(*varg);
        varg++;
    }
}
```

```
Se recibieron los siguientes argumentos :
c:\fuentes\p1.c
e:\p1.c
-
```

Si M Muestra lo que apunta el puntero, que es la segunda posición del array en la que está la dirección del 2º argumento

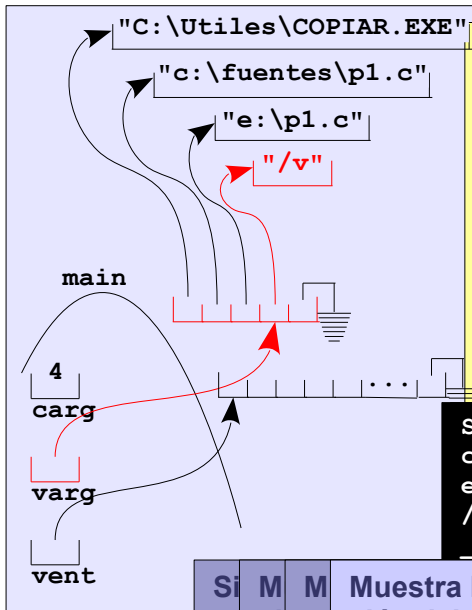
al In Incrementa el puntero, con lo que pasa a apuntar a la cuarta posición del array de punteros

ev ev evalúa nuevamente la condición del while que le da que

lo es es verdad, con lo que permanece en el ciclo repetitivo

Argumentos de main.

```
c:\>copiar c:\fuentes\p1.c e:\p1.c /v[Enter]
```



```
void main(int carg, char **varg, char **vent)
{
    /* ... */
    puts("Se recibieron los siguientes"
        " argumentos");
    varg++;
    while(*varg)
    {
        puts(*varg);
        varg++;
    }
}
```

Se recibieron los siguientes argumentos :

```
c:\fuentes\p1.c
e:\p1.c
/v
```

Si M M
re ci ci

al In
pu te

ev ev
lo es

Muestra lo que apunta el puntero, que es la segunda posición del array en la que está la dirección del 3er argumento

Incrementa el puntero, con lo que pasa a apuntar a la cuarta posición del array de punteros

evalúa nuevamente la condición del `while` que le da que es verdad, con lo que permanece en el ciclo repetitivo

Argumentos de main.

```
c:\>copiar c:\fuentes\p1.c e:\p1.c /v[Enter]
```

```
void main(int carg, char **varg, char **vent)
{
    /* ... */
    puts("Se recibieron los siguientes"
        " argumentos");
    varg++;
    while(*varg)
    {
        puts(*varg);
        varg++;
    }
}
```

Se recibieron los siguientes argumentos :

```
c:\fuentes\p1.c
e:\p1.c
/v
```

Si M M Muestra lo que apunta el puntero, que es la segunda posición del array en la que está la dirección del 3er argumento

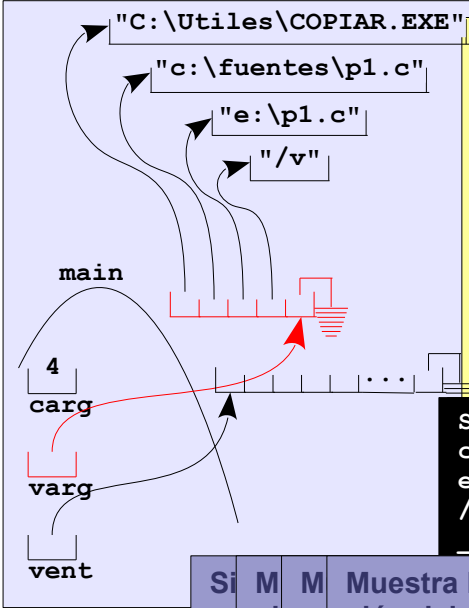
re ci ci Incrementa el puntero, con lo que pasa a apuntar a la quinta posición del array de punteros

al In In evalúa nuevamente la condición del while que le da que

pu te ta lo es es verdad, con lo que permanece en el ciclo repetitivo

Argumentos de `main`.

```
c:\>copiar c:\fuentes\p1.c e:\p1.c /v[Enter]
```



```
void main(int carg, char **varg, char **vent)
{
    /* ... */
    puts("Se recibieron los siguientes"
        " argumentos");
    varg++;
    while(*varg)
    {
        puts(*varg);
        varg++;
    }
}
```

```
Se recibieron los siguientes argumentos :
c:\fuentes\p1.c
e:\p1.c
/v
_
```

Si	M	M	Muestra lo que apunta el puntero, que es la segunda posición del array en la que está la dirección del 3er argumento
----	---	---	--

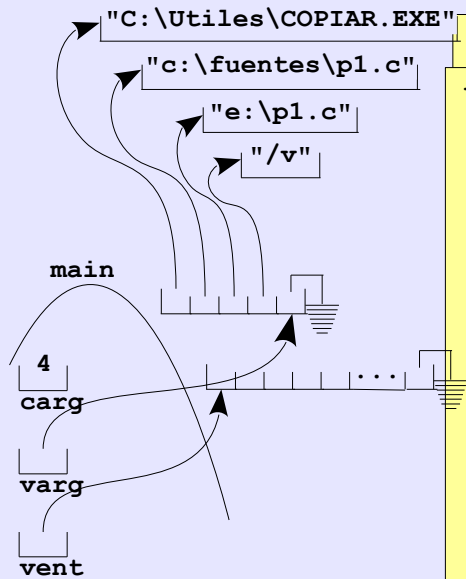
al	In	In	Incrementa el puntero, con lo que pasa a apuntar a la quinta posición del array de punteros
pu	te	ta	

evalúa nuevamente la condición del **while** que le da que es Falsa, porque apunta a **NULL** y sale del ciclo repetitivo

Argumentos de `main`.

Argumentos de main .

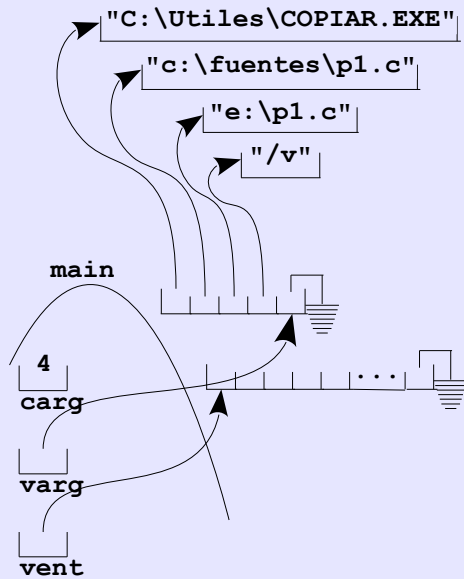
```
c:\>copiar c:\fuentes\p1.c e:\p1.c /v[Enter]
```



```
void main(int carg, char **varg, char **vent)  
{  
void main(int carg, char *varg[], char *vent[])  
{
```

El Lenguaje C permite de todas formas, que el manejo de los punteros a arrays se pueda hacer con subíndices, con lo que usted podría usarlos con subíndice los haya o no recibido con corchetes ([])

Argumentos de main.



```
c:\>copiar c:\fuentes\p1.c e:\p1.c /v[Enter]
```

```
void main(int carg, char *varg[], char *vent[])
{
    int posi;

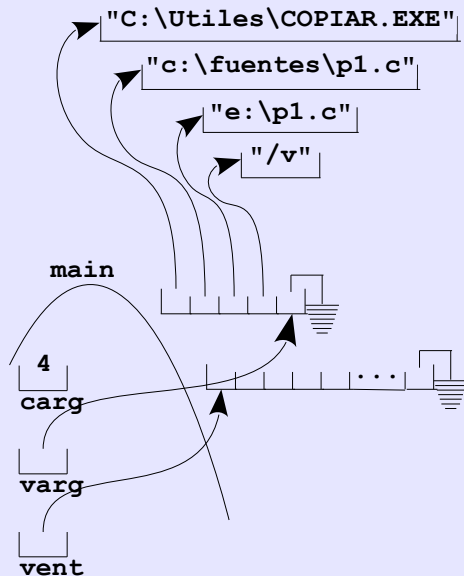
    printf("Se recibieron %d argumentos en la"
           " línea de comando\n",
           carg - 1);

    printf("El nombre del programa es %s\n",
           varg[0]);
    /* ... */
    puts("Se recibieron los siguientes"
         " argumentos");
    for(posi = 1; posi < carg; posi++)
    /* for(posi = 1; varg[posi]; posi++) */
        puts(varg[posi]);
    /* ... */
    for(posi = 0; vent[posi]; posi++)
        puts(vent[posi]);
}
```

El Lenguaje C permite de todas formas, que el manejo de los punteros a arrays se pueda hacer con subíndices, con lo podría usar subíndice los haya o no recibido con corchetes [i]

Esta versión del programa es (casi) equivalente al anterior, requiere la declaración de una variable entera para usarla de subíndice de varg (y de vent)

Argumentos de main.



```
c:\>copiar c:\fuentes\p1.c e:\p1.c /v[Enter]
```

```
void main(int carg, char *varg[], char *vent[])
{
    int posi;

    printf("Se recibieron %d argumentos en la"
           " línea de comando\n",
           carg - 1);

    printf("El nombre del programa es %s\n",
           varg[0]);
    /* ... */
    puts("Se recibieron los siguientes"
         " argumentos");
    for(posi = 1; posi < carg; posi++)
    /* for(posi = 1; varg[posi]; posi++) */
        puts(varg[posi]);
    /* ... */
    for(posi = 0; vent[posi]; posi++)
        puts(vent[posi]);
}
```

El Lenguaje C permite de todas formas, que el manejo de los punteros a arrays se pueda hacer con subíndices, con lo podría usar subíndice los haya o no recibido con corchetes []

Esta versión del programa es (casi) equivalente al anterior, requiere la declaración de una variable entera para usarla de subíndice de `varg` (o de `vent`)

Pero es muy importante que entienda el uso de punteros y su manejo.

Punteros

Argumentos de funciones (por valor) .

En c los argumentos a funciones, siempre se reciben por valor

Argumentos de funciones (por valor) .

En c los argumentos a funciones, siempre se reciben por valor

En este ejemplo, en `main` se declaran dos variables preinicializadas

```
void main(void)
{
    int a = 4,
        b = 5;

}
```

`main`

4

a

5

b

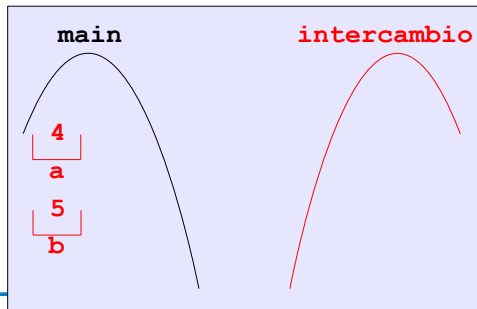
Punteros

Argumentos de funciones (por valor) .

En c los argumentos a funciones, siempre se reciben por valor

En este ejemplo, en `main` se declaran dos variables preinicializadas

Al invocar a la función `intercambio` con las variables ...



```
void intercambio(int v1, int v2);

void main(void)
{
    int a = 4,
        b = 5;

    intercambio(a, b);
}
```

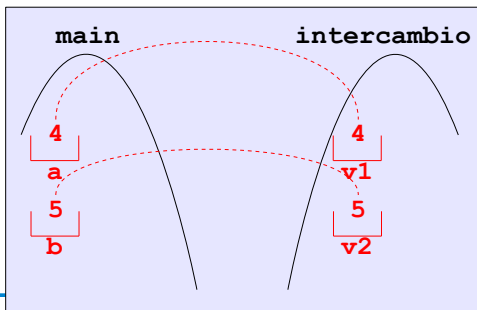
Argumentos de funciones (por valor) .

En c los argumentos a funciones, siempre se reciben por valor

En este ejemplo, en `main` se declaran dos variables preinicializadas

Al invocar a la función `intercambio` con las variables ...

... al comenzar a ejecutarse, se generan copias de las variables originales



```
void intercambio(int v1, int v2);

void main(void)
{
    int a = 4,
        b = 5;

    intercambio(a, b);
}

void intercambio(int v1, int v2)
{
}
```

Argumentos de funciones (por valor) .

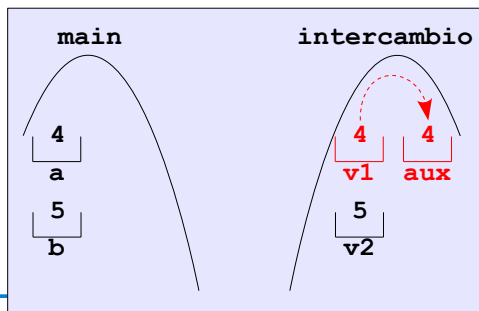
En c los argumentos a funciones, siempre se reciben por valor

En este ejemplo, en `main` se declaran dos variables preinicializadas

Al invocar a la función `intercambio` con las variables ...

... al comenzar a ejecutarse, se generan copias de las variables originales

Aunque en la función se modifiquen sus argumentos



```
void intercambio(int v1, int v2);
```

```
void main(void)
```

```
{
```

```
    int a = 4,
```

```
        b = 5;
```

```
    intercambio(a, b);
```

```
}
```

```
void intercambio(int v1, int v2)
```

```
{
```

```
    int aux = v1;
```

```
}
```

Punteros

Argumentos de funciones (por valor) .

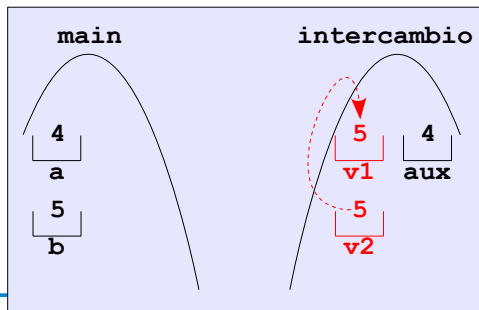
En c los argumentos a funciones, siempre se reciben por valor

En este ejemplo, en `main` se declaran dos variables preinicializadas

Al invocar a la función `intercambio` con las variables ...

... al comenzar a ejecutarse, se generan copias de las variables originales

Aunque en la función se modifiquen sus argumentos ...



```
void intercambio(int v1, int v2);
```

```
void main(void)
```

```
{
```

```
    int a = 4,
```

```
        b = 5;
```

```
    intercambio(a, b);
```

```
}
```

```
void intercambio(int v1, int v2)
```

```
{
```

```
    int aux = v1;
```

```
    v1 = v2;
```

```
}
```

Punteros

Argumentos de funciones (por valor) .

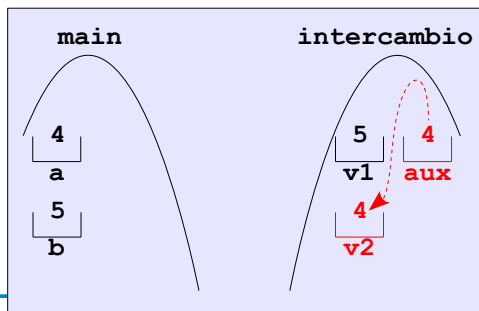
En c los argumentos a funciones, siempre se reciben por valor

En este ejemplo, en `main` se declaran dos variables preinicializadas

Al invocar a la función `intercambio` con las variables ...

... al comenzar a ejecutarse, se generan copias de las variables originales

Aunque en la función se modifiquen sus argumentos



```
void intercambio(int v1, int v2);
```

```
void main(void)
```

```
{
```

```
    int a = 4,
```

```
        b = 5;
```

```
    intercambio(a, b);
```

```
}
```

```
void intercambio(int v1, int v2)
```

```
{
```

```
    int aux = v1;
```

```
    v1 = v2;
```

```
    v2 = aux;
```

```
}
```

Punteros

Argumentos de funciones (por valor) .

En c los argumentos a funciones, siempre se reciben por valor

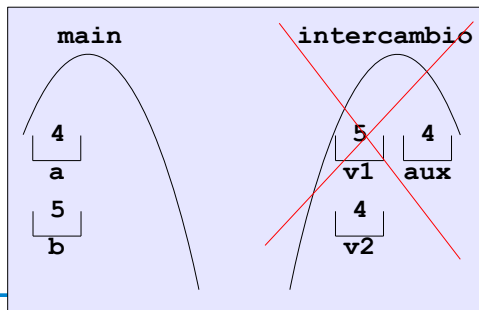
En este ejemplo, en `main` se declaran dos variables preinicializadas

Al invocar a la función `intercambio` con las variables ...

... al comenzar a ejecutarse, se generan copias de las variables originales

Aunque en la función se modifiquen sus argumentos

... al terminar de ejecutarse ...



```
void intercambio(int v1, int v2);
```

```
void main(void)
```

```
{
```

```
    int a = 4,
```

```
        b = 5;
```

```
    intercambio(a, b);
```

```
}
```

```
void intercambio(int v1, int v2)
```

```
{
```

```
    int aux = v1;
```

```
    v1 = v2;
```

```
    v2 = aux;
```

```
}
```

Punteros

Argumentos de funciones (por valor) .

En C los argumentos a funciones, siempre se reciben por valor

En este ejemplo, en `main` se declaran dos variables preinicializadas

Al invocar a la función `intercambio` con las variables ...

... al comenzar a ejecutarse, se generan copias de las variables originales

Aunque en la función se modifiquen sus argumentos

... al terminar de ejecutarse ...

... las variables de `main` mantienen sus valores originales .

```
void intercambio(int v1, int v2);
```

```
void main(void)
```

```
{
```

```
    int a = 4,
```

```
        b = 5;
```

```
    intercambio(a, b);
```

```
    /* a y b mantienen sus valores */
```

```
}
```

```
void intercambio(int v1, int v2)
```

```
{
```

```
    int aux = v1;
```

```
    v1 = v2;
```

```
    v2 = aux;
```

```
}
```

`main`

4

a

5

b

Argumentos de funciones (por '*referencia*' o puntero) .

En c los argumentos a funciones, siempre se reciben por valor

Argumentos de funciones (por '*referencia*' o puntero) .

En C los argumentos a funciones, siempre se reciben por valor

En la bibliografía se suele decir argumentos por referencia, lo cual es correcto a medias .

Argumentos de funciones (por '*referencia*' o puntero) .

En c los argumentos a funciones, siempre se reciben por valor

En la bibliografía se suele decir argumentos por referencia, lo cual es correcto a medias .

En c no hay referencias como en C++ .

Argumentos de funciones (por '*referencia*' o puntero) .

En C los argumentos a funciones, siempre se reciben por valor

En la bibliografía se suele decir argumentos por referencia, lo cual es correcto a medias .

En C no hay referencias como en C++ .

Cuando habla de referencias en C se quiere significar que se hace referencia a la/s variable/s original/es mediante el uso de puntero/s .

Argumentos de funciones (por '*referencia*' o puntero) .

En c los argumentos a funciones, siempre se reciben por valor

En este ejemplo, en `main` se declaran dos variables preinicializadas

```
void main(void)
{
    int a = 4,
        b = 5;

}
```

`main`

4

a

5

b

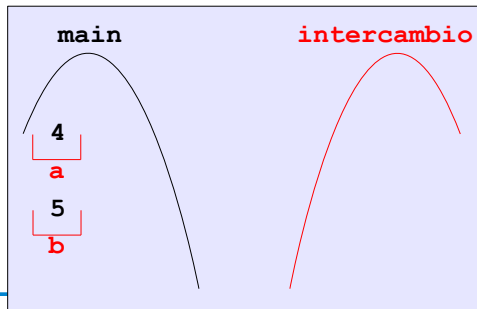
Punteros

Argumentos de funciones (por '*referencia*' o puntero) .

En c los argumentos a funciones, siempre se reciben por valor

En este ejemplo, en `main` se declaran dos variables preinicializadas

Al invocar a la función `intercambio` con las direcciones de las variables ...



```
void intercambio(int *v1, int *v2);

void main(void)
{
    int a = 4,
        b = 5;

    intercambio(&a, &b);
}
```

Punteros

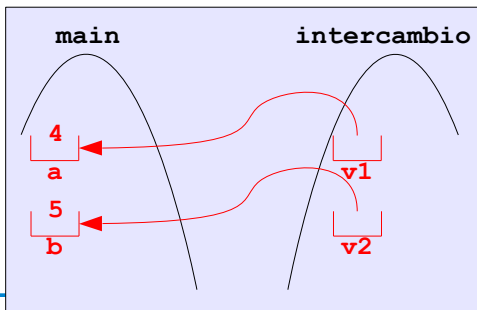
Argumentos de funciones (por '*referencia*' o puntero) .

En c los argumentos a funciones, siempre se reciben por valor

En este ejemplo, en `main` se declaran dos variables preinicializadas

Al invocar a la función `intercambio` con las direcciones de las variables ...

... al comenzar a ejecutarse, se generan punteros a las variables originales



```
void intercambio(int *v1, int *v2);

void main(void)
{
    int a = 4,
        b = 5;

    intercambio(&a, &b);
}

void intercambio(int *v1, int *v2)
{
}
```

Punteros

Argumentos de funciones (por '*referencia*' o puntero) .

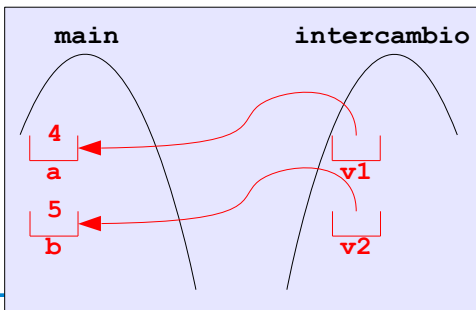
En c los argumentos a funciones, siempre se reciben por valor

En este ejemplo, en `main` se declaran dos variables preinicializadas

Al invocar a la función `intercambio` con las direcciones de las variables . . .

. . . al comenzar a ejecutarse, se generan punteros a las variables originales

Recuerde que los punteros son variables que reciben valores que son las direcciones de memoria de las variables con que se invocó a la función .



```
void intercambio(int *v1, int *v2);

void main(void)
{
    int a = 4,
        b = 5;

    intercambio(&a, &b);
}

void intercambio(int *v1, int *v2)
{
}
```

Punteros

Argumentos de funciones (por '*referencia*' o puntero) .

En c los argumentos a funciones, siempre se reciben por valor

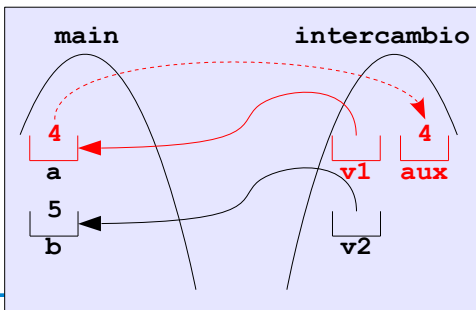
En este ejemplo, en `main` se declaran dos variables preinicializadas

Al invocar a la función `intercambio` con las direcciones de las variables . . .

. . . al comenzar a ejecutarse, se generan punteros a las variables originales

Recuerde que los punteros son variables que reciben valores que son las direcciones de memoria de las variables con que se invocó a la función .

Ahora sí, mediante los punteros intercambiamos las variables originales



```
void intercambio(int *v1, int *v2);

void main(void)
{
    int a = 4,
        b = 5;

    intercambio(&a, &b);
}

void intercambio(int *v1, int *v2)
{
    int aux = *v1;
}
```

Punteros

Argumentos de funciones (por '*referencia*' o puntero) .

En c los argumentos a funciones, siempre se reciben por valor

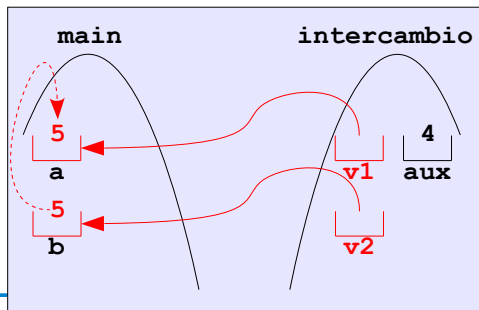
En este ejemplo, en `main` se declaran dos variables preinicializadas

Al invocar a la función `intercambio` con las direcciones de las variables ...

... al comenzar a ejecutarse, se generan punteros a las variables originales

Recuerde que los punteros son variables que reciben valores que son las direcciones de memoria de las variables con que se invocó a la función .

Ahora sí, mediante los punteros intercambiamos las variables originales



```
void intercambio(int *v1, int *v2);

void main(void)
{
    int a = 4,
        b = 5;

    intercambio(&a, &b);
}

void intercambio(int *v1, int *v2)
{
    int aux = *v1;
    *v1 = *v2;
}
```

Punteros

Argumentos de funciones (por '*referencia*' o puntero) .

En c los argumentos a funciones, siempre se reciben por valor

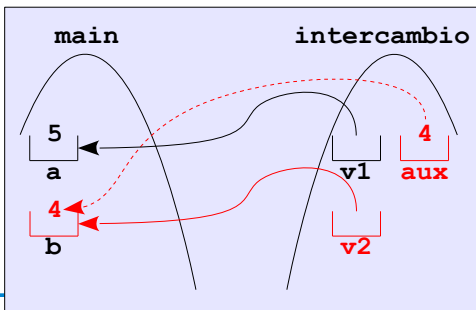
En este ejemplo, en `main` se declaran dos variables preinicializadas

Al invocar a la función `intercambio` con las direcciones de las variables . . .

. . . al comenzar a ejecutarse, se generan punteros a las variables originales

Recuerde que los punteros son variables que reciben valores que son las direcciones de memoria de las variables con que se invocó a la función .

Ahora sí, mediante los punteros intercambiamos las variables originales



```
void intercambio(int *v1, int *v2);

void main(void)
{
    int a = 4,
        b = 5;

    intercambio(&a, &b);
}

void intercambio(int *v1, int *v2)
{
    int aux = *v1;
    *v1 = *v2;
    *v2 = aux;
}
```

Punteros

Argumentos de funciones (por '*referencia*' o puntero) .

En c los argumentos a funciones, siempre se reciben por valor

En este ejemplo, en `main` se declaran dos variables preinicializadas

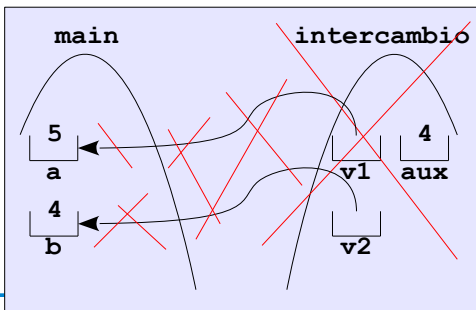
Al invocar a la función `intercambio` con las direcciones de las variables ...

... al comenzar a ejecutarse, se generan punteros a las variables originales

Recuerde que los punteros son variables que reciben valores que son las direcciones de memoria de las variables con que se invocó a la función .

Ahora sí, mediante los punteros intercambiamos las variables originales

... al terminar de ejecutarse ...



```
void intercambio(int *v1, int *v2);

void main(void)
{
    int a = 4,
        b = 5;

    intercambio(&a, &b);
}

void intercambio(int *v1, int *v2)
{
    int aux = *v1;
    *v1 = *v2;
    *v2 = aux;
}
```

Punteros

Argumentos de funciones (por '*referencia*' o puntero) .

En c los argumentos a funciones, siempre se reciben por valor

En este ejemplo, en `main` se declaran dos variables preinicializadas

Al invocar a la función `intercambio` con las direcciones de las variables ...

... al comenzar a ejecutarse, se generan punteros a las variables originales

Recuerde que los punteros son variables que reciben valores que son las direcciones de memoria de las variables con que se invocó a la función .

Ahora sí, mediante los punteros intercambiamos las variables originales

... al terminar de ejecutarse ...

... las variables de `main` intercambiaron sus valores originales .

```
void intercambio(int *v1, int *v2);
```

```
void main(void)
```

```
{  
    int a = 4,  
        b = 5;
```

```
    intercambio(&a, &b);
```

```
    /* intercambiaron sus valores */  
}
```

```
void intercambio(int *v1, int *v2)
```

```
{  
    int aux = *v1;  
    *v1 = *v2;  
    *v2 = aux;  
}
```

main

5

a

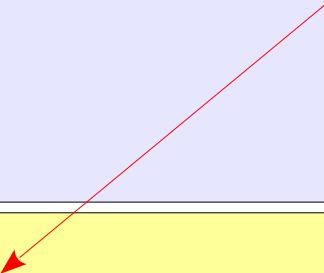
4

b

Punteros

Argumentos de funciones - Punteros a estructuras .

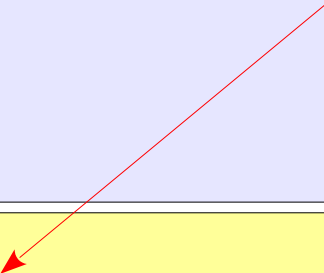
Ya hemos hablado de qué son las estructuras, se puede hacer uso de las declaraciones `struct`



```
struct t_info
{
    long dni;
    char apyn[31];
};
```

Argumentos de funciones - Punteros a estructuras .

Ya hemos hablado de qué son las estructuras, se puede hacer uso de las declaraciones `struct` o mejor aún con `typedef struct` .



```
typedef struct
{
    long dni;
    char apyn[31];
} t_info;
```

Argumentos de funciones - Punteros a estructuras .

Ya hemos hablado de qué son las estructuras, se puede hacer uso de las declaraciones `struct` o mejor aún con `typedef struct`.

Lo que se logra en ambos casos es declarar el 'molde'

```
typedef struct
{
    long dni;
    char apyn[31];
} t_info;
```

[illegible]

Argumentos de funciones - Punteros a estructuras .

The diagram shows a light blue rectangular area labeled "main" at the top left. A black arc originates from the left side of the "main" area and points to a red-bordered box. This box contains three entries: "dni" followed by a single empty slot, "apyn" followed by a row of 11 slots (the 4th slot contains three dots "..."), and "info" followed by no slots. To the right of the "main" area, a red line extends diagonally upwards and to the right, ending at a vertical line that represents the edge of the memory segment.

The diagram shows a light blue rectangular area labeled "main" at the top left. A black arc originates from the left side of the "main" area and points to a red-bordered box. This box contains three entries: "dni" followed by a single empty slot, "apyn" followed by a row of 11 slots (the 4th slot contains three dots "..."), and "info" followed by no slots. To the right of the "main" area, a red line extends diagonally upwards and to the right, ending at a vertical line that represents the edge of the memory segment.

Ya hemos hablado de qué son las estructuras, se puede hacer uso de las declaraciones `struct` o mejor aún con `typedef struct`.

Lo que se logra en ambos casos es declarar el 'molde' con el cual se podrán crear variables con esos miembros de información .

```
typedef struct
{
    long dni;
    char apyn[31];
} t_info;

void main(void)
{
    t_info info;
}
```

[illegible]

```
typedef struct
{
    long dni;
    char apyn[31];
} t_info;

void main(void)
{
    t_info info;
}
```

```
typedef struct
{
    long dni;
    char apyn[31];
} t_info;

void main(void)
{
    t_info info;
}
```

Argumentos de funciones - Punteros a estructuras .

main

dni []
apyn []
info

Ya hemos hablado de qué son las estructuras, se puede hacer uso de las declaraciones `struct` o mejor aún con `typedef struct` .

Lo que se logra en ambos casos es declarar el 'molde' con el cual se podrán crear variables con esos miembros de información .

```
typedef struct  
{  
    long dni;  
    char apyn[31];  
} t_info;
```

dni []
apyn []
t_info

Si lo que se necesita es `ingresar` información de ese tipo

```
void main(void)  
{  
    t_info info;  
    if (cargar(&info))  
        mostrar(&info);  
}
```

Argumentos de funciones - Punteros a estructuras .

main

dni []
apyn []
info

Ya hemos hablado de qué son las estructuras, se puede hacer uso de las declaraciones `struct` o mejor aún con `typedef struct` .

Lo que se logra en ambos casos es declarar el 'molde' con el cual se podrán crear variables con esos miembros de información .

```
typedef struct  
{  
    long dni;  
    char apyn[31];  
} t_info;
```

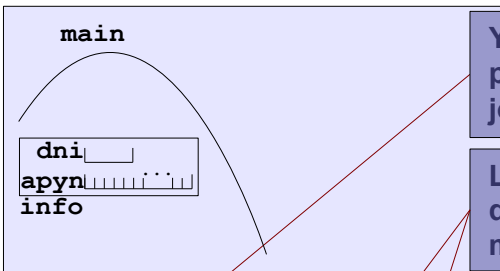
dni []
apyn []
t_info

Si lo que se necesita es `ingresar` información de ese tipo, y si se la ingresó, `mostrarla`, podríamos hacerlo mediante funciones que cumplan con tal cometido .

```
void main(void)  
{  
    t_info info;  
    if (cargar(&info))  
        mostrar(&info);  
}
```

Punteros

Argumentos de funciones - Punteros a estructuras .



Ya hemos hablado de qué son las estructuras, se puede hacer uso de las declaraciones `struct` o mejor aún con `typedef struct` .

Lo que se logra en ambos casos es declarar el 'molde' con el cual se podrán crear variables con esos miembros de información .

```
typedef struct
{
    long dni;
    char apyn[31];
} t_info;
```

The diagram shows a variable of type `t_info` named `info`. It has members `dni`, `apyn`, and `t_info`.

Si lo que se necesita es `ingresar` información de ese tipo, y si se la ingresó, `mostrarla`, podríamos hacerlo mediante funciones que cumplan con tal cometido .

Si la función que ingresa la información no logra su cometido, devolverá un valor que lo indique para mostrar un mensaje adecuado (podría estar intentando leerla de un archivo, recibirla por un socket, etc.) .

```
void main(void)
{
    t_info info;

    if (cargar(&info))
        mostrar(&info)
    else
        puts("No se cargó información");
}
```


Argumentos de funciones - Punteros a estructuras .

main

dni
apyn
info

Ya hemos hablado de qué son las estructuras, se puede hacer uso de las declaraciones `struct` o mejor aún con `typedef struct` .

Lo que se logra en ambos casos es declarar el 'molde' con el cual se podrán crear variables con esos miembros de información .

```
typedef struct  
{  
    long dni;  
    char apyn[31];  
} t_info;
```

dni
apyn
t_info

Si lo que se necesita es `ingresar` información de ese tipo, y si se la ingresó, `mostrarla`, podríamos hacerlo mediante funciones que cumplan con tal cometido .

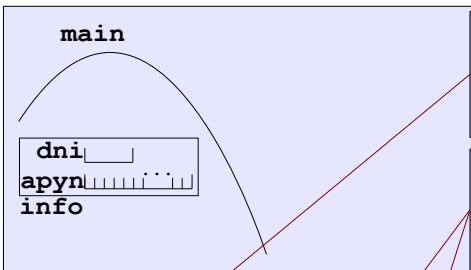
```
int cargar(t_info *p);  
void mostrar(const t_info *p);
```

Si la función que ingresa la información no logra su cometido, devolverá un valor que lo indique para mostrar un mensaje adecuado (podría estar intentando leerla de un archivo, recibirla por un socket, etc.) .

```
void main(void)  
{  
    t_info info;  
  
    if(cargar(&info))  
        mostrar(&info)  
    else  
        puts("No se cargó información");  
}
```

Completamos nuestro programa con los prototipos de las funciones utilizadas (se invocan pasándoles la dirección de memoria de la variable `info` de `main`, por lo que reciben `t_info *`) .

Argumentos de funciones - Punteros a estructuras .



Ya hemos hablado de qué son las estructuras, se puede hacer uso de las declaraciones `struct` o mejor aún con `typedef struct` .

Lo que se logra en ambos casos es declarar el 'molde' con el cual se podrán crear variables con esos miembros de información .

```
#include <stdio.h>

typedef struct
{
    long dni;
    char apyn[31];
} t_info;

int cargar(t_info *p);
void mostrar(const t_info *p);

void main(void)
{
    t_info info;

    if(cargar(&info))
        mostrar(&info)
    else
        puts("No se cargó informa");
}
```

Si lo que se necesita es `ingresar` información de ese tipo, y si se la ingresó, `mostrarla`, podríamos hacerlo mediante funciones que cumplan con tal cometido .

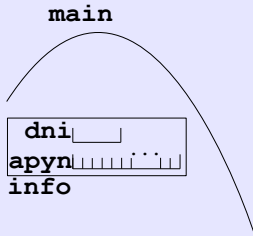
Si la función que ingresa la información no logra su cometido, devolverá un valor que lo indique para mostrar un mensaje adecuado (podría estar intentando leerla de un archivo, recibirla por un socket, etc.) .

Completamos nuestro programa con los prototipos de las funciones utilizadas (se invocan pasándoles la dirección de memoria de la variable `info` de `main`, por lo que reciben `t_info *`) .

Sólo faltan los archivos cabecera de las funciones de biblioteca estándar necesarias para el programa .

Argumentos de funciones - Punteros a estructuras .

¿Queda clara la estructuración del programa, invocando funciones que resuelvan el problema del ingreso y egreso de la información que nos hemos planteado para este ejemplo?



```
#include <stdio.h>

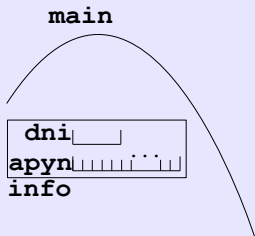
typedef struct
{
    long dni;
    char apyn[31];
} t_info;

int cargar(t_info *p);
void mostrar(const t_info *p);

void main(void)
{
    t_info info;

    if(cargar(&info))
        mostrar(&info);
    else
        puts("No se cargó información");
}
```

Argumentos de funciones - Punteros a estructuras .



¿Queda clara la estructuración del programa, invocando funciones que resuelvan el problema del ingreso y egreso de la información que nos hemos planteado para este ejemplo?

Pasemos a ver cómo se invocan y ejecutan las funciones que estamos invocando .

```
#include <stdio.h>

typedef struct
{
    long dni;
    char apyn[31];
} t_info;

int cargar(t_info *p);
void mostrar(const t_info *p);

void main(void)
{
    t_info info;

    if(cargar(&info))
        mostrar(&info);
    else
        puts("No se cargó información");
}
```

Argumentos de funciones - Punteros a estructuras .

main

dni []
apyn []
info

Se evalúa el operador &

```
#include <stdio.h>
```

```
typedef struct
```

```
{
```

```
    long dni;
```

```
    char apyn[31];
```

```
} t_info;
```

```
int cargar(t_info *p);
```

```
void mostrar(const t_info *p);
```

```
void main(void)
```

```
{
```

```
    t_info info;
```

```
    if(cargar(&info))
```

```
        mostrar(&info);
```

```
    else
```

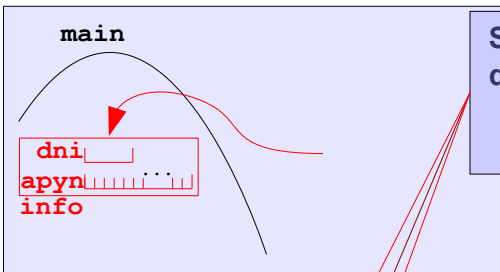
```
        puts("No se cargó información");
```

```
}
```

dni []
apyn []
t_info

Punteros

Argumentos de funciones - Punteros a estructuras .



Se evalúa el operador & que obtiene la dirección en que se encuentra la variable `info`

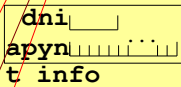
```
#include <stdio.h>

typedef struct
{
    long dni;
    char apyn[31];
} t_info;

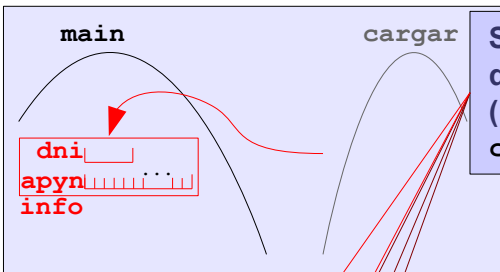
int cargar(t_info *p);
void mostrar(const t_info *p);

void main(void)
{
    t_info info;

    if(cargar(&info))
        mostrar(&info);
    else
        puts("No se cargó información");
}
```



Argumentos de funciones - Punteros a estructuras .



Se evalúa el operador `&` que obtiene la dirección en que se encuentra la variable `info`, y con este valor (la dirección de memoria) se invoca a la función `cargar` .

```
#include <stdio.h>

typedef struct
{
    long dni;
    char apyn[31];
} t_info;

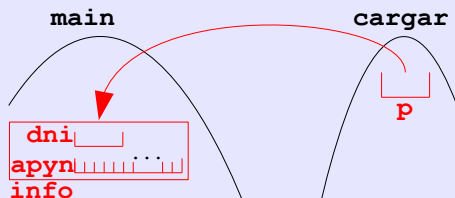
int cargar(t_info *p);
void mostrar(const t_info *p);

void main(void)
{
    t_info info;

    if(cargar(&info))
        mostrar(&info);
    else
        puts("No se cargó información");
}
```

```
int cargar(t_info *p)
{
    // ...
}
```

Argumentos de funciones - Punteros a estructuras .

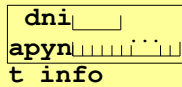


Al comenzar a ejecutarse la función **cargar**, recibe en la variable **p** el valor con que se la invocó, es decir la dirección de memoria en que se encuentra la variable **info** de **main** .

```
#include <stdio.h>
```

```
typedef struct
```

```
{  
    long dni;  
    char apyn[31];  
} t_info;
```



```
int cargar(t_info *p);  
void mostrar(const t_info *p);
```

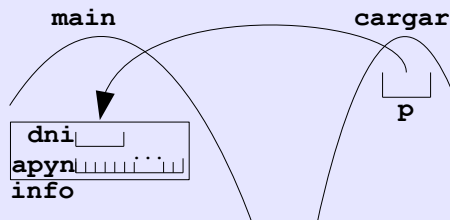
```
void main(void)
```

```
{  
    t_info info;  
  
    if(cargar(&info))  
        mostrar(&info);  
    else  
        puts("No se cargó información");  
}
```

```
int cargar(t_info *p)  
{
```

```
}
```


Argumentos de funciones - Punteros a estructuras .



Al comenzar a ejecutarse la función **cargar**, recibe en la variable **p** el valor con que se la invocó, es decir la dirección de memoria en que se encuentra la variable **info** de **main** .

Comienza por mostrar un mensaje y liberar la stream de teclado .

```
typedef struct
{
    long dni;
    char apyn[31];
} t_info;

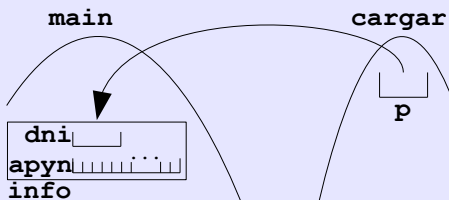
int cargar(t_info *p);
void mostrar(const t_info *p);

void main(void)
{
    t_info info;

    if(cargar(&info))
        mostrar(&info);
    else
        puts("No se cargó información");
}
```

```
int cargar(t_info *p)
{
    printf("Cero o negativo, sale\n"
           "D. N. I. : ");
    fflush(stdin);
}
```

Argumentos de funciones - Punteros a estructuras .



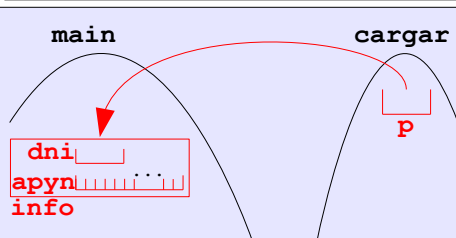
Al comenzar a ejecutarse la función **cargar**, recibe en la variable **p** el valor con que se la invocó, es decir la dirección de memoria en que se encuentra la variable **info** de **main** .

Comienza por mostrar un mensaje y liberar la stream de teclado .

Note que **p** es un puntero, que contiene la dirección de memoria en que se encuentra una variable del tipo **t_info** (en este caso la variable **info** de **main**) .

```
int cargar(t_info *p)
{
    printf("Cero o negativo, sale\n"
           "D. N. I. : ");
    fflush(stdin);
    scanf("%ld", &p->dni);
}
```

Argumentos de funciones - Punteros a estructuras .



Al comenzar a ejecutarse la función **cargar**, recibe en la variable **p** el valor con que se la invocó, es decir la dirección de memoria en que se encuentra la variable **info** de **main** .

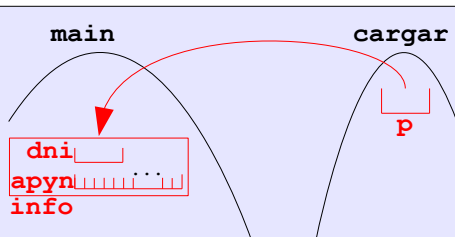
Comienza por mostrar un mensaje y liberar la stream de teclado .

Note que **p** es un puntero, que contiene la dirección de memoria en que se encuentra una variable del tipo **t_info** (en este caso la variable **info** de **main**) .

Si utilizamos ***p** estamos direccionando toda la variable **info** de **main** .

```
int cargar(t_info *p)
{
    printf("Cero o negativo, sale\n"
           "D. N. I. : ");
    fflush(stdin);
    scanf("%ld", &p->dni);
}
```

Argumentos de funciones - Punteros a estructuras .



Al comenzar a ejecutarse la función **cargar**, recibe en la variable **p** el valor con que se la invocó, es decir la dirección de memoria en que se encuentra la variable **info** de **main** .

Comienza por mostrar un mensaje y liberar la stream de teclado .

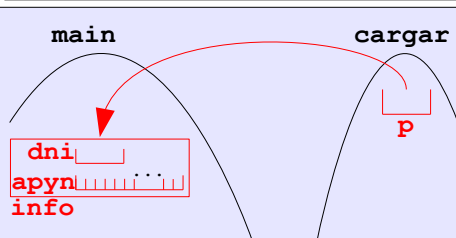
Note que **p** es un puntero, que contiene la dirección de memoria en que se encuentra una variable del tipo **t_info** (en este caso la variable **info** de **main**) .

Si utilizamos ***p** estamos direccionando toda la variable **info** de **main** .

La notación **(*p).dni** es equivalente (en **cargar**) a **info.dni** (en **main**) .

```
int cargar(t_info *p)
{
    printf("Cero o negativo, sale\n"
           "D. N. I. : ");
    fflush(stdin);
    scanf("%ld", &p->dni);
}
```

Argumentos de funciones - Punteros a estructuras .



Al comenzar a ejecutarse la función **cargar**, recibe en la variable **p** el valor con que se la invocó, es decir la dirección de memoria en que se encuentra la variable **info** de **main** .

Comienza por mostrar un mensaje y liberar la stream de teclado .

Note que **p** es un puntero, que contiene la dirección de memoria en que se encuentra una variable del tipo **t_info** (en este caso la variable **info** de **main**) .

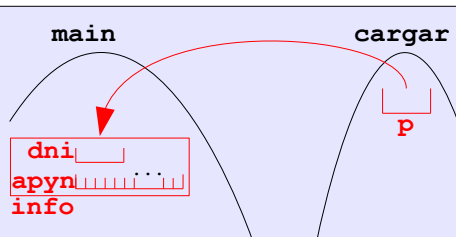
Si utilizamos ***p** estamos direccionando toda la variable **info** de **main** .

La notación **(*p).dni** es equivalente (en **cargar**) a **info.dni** (en **main**) .

Note que es necesario el uso de **()** dado que el **'*'** tiene menor precedencia que el **'.'** .

```
int cargar(t_info *p)
{
    printf("Cero o negativo, sale\n"
           "D. N. I. : ");
    fflush(stdin);
    scanf("%ld", &p->dni);
}
```

Argumentos de funciones - Punteros a estructuras .



Al comenzar a ejecutarse la función **cargar**, recibe en la variable **p** el valor con que se la invocó, es decir la dirección de memoria en que se encuentra la variable **info** de **main** .

Comienza por mostrar un mensaje y liberar la stream de teclado .

Note que **p** es un puntero, que contiene la dirección de memoria en que se encuentra una variable del tipo **t_info** (en este caso la variable **info** de **main**) .

Si utilizamos ***p** estamos direccionando toda la variable **info** de **main** .

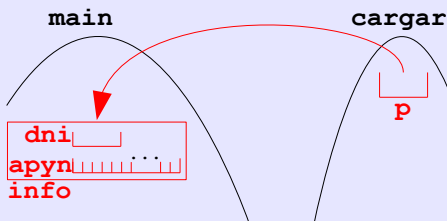
La notación **(*p) .dni** es equivalente (en **cargar**) a **info .dni** (en **main**) .

Note que es necesario el uso de **()** dado que el **'*'** tiene menor precedencia que el **'.'** .

La notación **p->dni** es una forma más conveniente que **(*p) .dni** y es el modo que utilizaremos en nuestros programas .

```
int cargar(t_info *p)
{
    printf("Cero o negativo, sale\n"
           "D. N. I. : ");
    fflush(stdin);
    scanf("%ld", &p->dni);
}
```

Argumentos de funciones - Punteros a estructuras .



Al comenzar a ejecutarse la función `cargar`, recibe en la variable `p` el valor con que se la invocó, es decir la dirección de memoria en que se encuentra la variable `info` de `main`.

Comienza por mostrar un mensaje y liberar la stream de teclado .

Se ejecuta entonces el ingreso (mediante el `scanf("%ld", &p->dni);`) del miembro `dni` de `info` de `main`.

```
} t_info;
```

```
int cargar(t_info *p);  
void mostrar(const t_info *p);
```

```
void main(void)  
{  
    t_info info;  
  
    if(cargar(&info))  
        mostrar(&info);  
    else  
        puts("No se cargó información");  
}
```

```
int cargar(t_info *p)
```

```
{
```

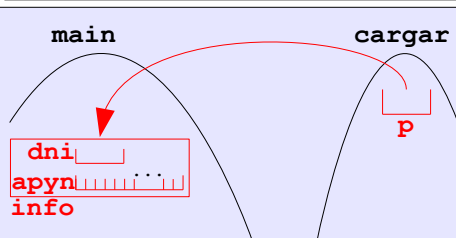
```
    printf("Cero o negativo, sale\n"  
           "D. N. I. : ");
```

```
    fflush(stdin);
```

```
    scanf("%ld", &p->dni);
```

```
}
```

Argumentos de funciones - Punteros a estructuras .



Al comenzar a ejecutarse la función `cargar`, recibe en la variable `p` el valor con que se la invocó, es decir la dirección de memoria en que se encuentra la variable `info` de `main`.

Comienza por mostrar un mensaje y liberar la stream de teclado .

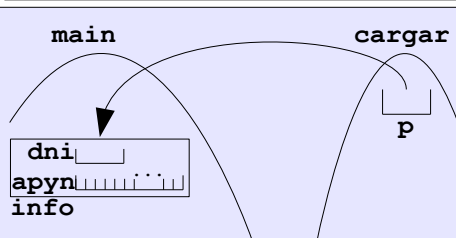
Se ejecuta entonces el ingreso (mediante el `scanf("%ld", &p->dni);`) del miembro `dni` de `info` de `main`.

`t_info:`

Si se ingresó un DNI mayor que cero

```
int cargar(t_info *p)
{
    printf("Cero o negativo, sale\n"
           "D. N. I. : ");
    fflush(stdin);
    scanf("%ld", &p->dni);
    if(p->dni > 0L)
    {
    }
}
```


Argumentos de funciones - Punteros a estructuras .



Al comenzar a ejecutarse la función `cargar`, recibe en la variable `p` el valor con que se la invocó, es decir la dirección de memoria en que se encuentra la variable `info` de `main`.

Comienza por mostrar un mensaje y liberar la stream de teclado .

Se ejecuta entonces el ingreso (mediante el `scanf("%ld", &p->dni);`) del miembro `dni` de `info` de `main`.

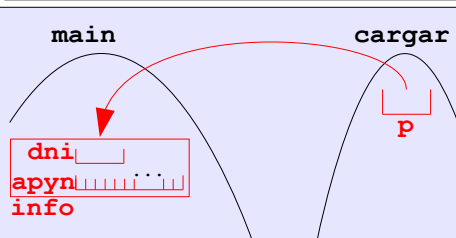
`t_info:`

Si se ingresó un DNI mayor que cero, entonces

muestra un nuevo mensaje,
libera la stream de teclado

```
int cargar(t_info *p)
{
    printf("Cero o negativo, sale\n"
           "D. N. I. : ");
    fflush(stdin);
    scanf("%ld", &p->dni);
    if(p->dni > 0L)
    {
        printf("Apellido y Nombre : ");
        fflush(stdin);
    }
}
```

Argumentos de funciones - Punteros a estructuras .



Al comenzar a ejecutarse la función cargar, recibe en la variable **p** el valor con que se la invocó, es decir la dirección de memoria en que se encuentra la variable **info** de **main** .

Comienza por mostrar un mensaje y liberar la stream de teclado .

Se ejecuta entonces el ingreso (mediante el `scanf("%ld", &p->dni);`) del miembro **dni** de **info** de **main** .

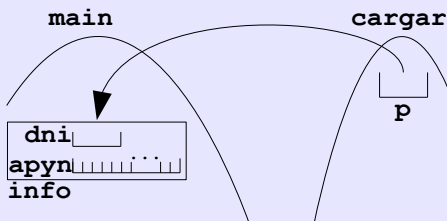
`t_info:`

Si se ingresó un DNI mayor que cero, entonces

muestra un nuevo mensaje,
libera la **stream** de teclado,
ingresa el apellido y nombre

```
int cargar(t_info *p)
{
    printf("Cero o negativo, sale\n"
           "D. N. I. : ");
    fflush(stdin);
    scanf("%ld", &p->dni);
    if(p->dni > 0L)
    {
        printf("Apellido y Nombre : ");
        fflush(stdin);
        gets(p->apyn);
    }
}
```

Argumentos de funciones - Punteros a estructuras .



Al comenzar a ejecutarse la función cargar, recibe en la variable **p** el valor con que se la invocó, es decir la dirección de memoria en que se encuentra la variable **info** de **main** .

Comienza por mostrar un mensaje y liberar la stream de teclado .

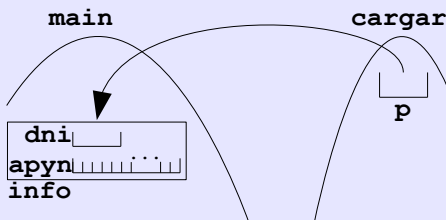
Se ejecuta entonces el ingreso (mediante el `scanf("%ld", &p->dni);`) del miembro **dni** de **info** de **main** .

`t_info:`

Si se ingresó un DNI mayor que cero, entonces
muestra un nuevo mensaje,
libera la stream de teclado,
ingresa el apellido y nombre, y
devuelve 1 a quien haya invocado (**main**)
fin-si

```
int cargar(t_info *p)
{
    printf("Cero o negativo, sale\n"
           "D. N. I. : ");
    fflush(stdin);
    scanf("%ld", &p->dni);
    if(p->dni > 0L)
    {
        printf("Apellido y Nombre : ");
        fflush(stdin);
        gets(p->apyn);
        return 1; /* cargó */
    }
}
```

Argumentos de funciones - Punteros a estructuras .



Al comenzar a ejecutarse la función cargar, recibe en la variable **p** el valor con que se la invocó, es decir la dirección de memoria en que se encuentra la variable **info** de **main** .

Comienza por mostrar un mensaje y liberar la stream de teclado .

Se ejecuta entonces el ingreso (mediante el `scanf("%ld", &p->dni);`) del miembro **dni** de **info** de **main** .

t_info:

Si se ingresó un DNI mayor que cero, entonces

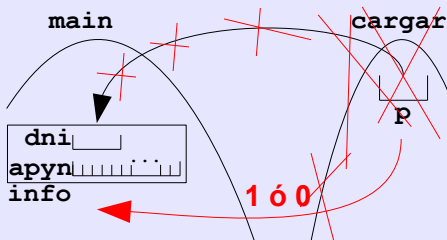
muestra un nuevo mensaje,
libera la stream de teclado,
ingresa el apellido y nombre, y
devuelve 1 a quien haya invocado (main)

fin-si

devuelve 0 a quien haya invocado (main)

```
int cargar(t_info *p)
{
    printf("Cero o negativo, sale\n"
           "D. N. I. : ");
    fflush(stdin);
    scanf("%ld", &p->dni);
    if(p->dni > 0L)
    {
        printf("Apellido y Nombre : ");
        fflush(stdin);
        gets(p->apyn);
        return 1; /* cargó */
    }
    return 0; /* no cargó */
}
```

Argumentos de funciones - Punteros a estructuras .



Al terminar su ejecución la función **cargar**, devuelve **1 ó 0** a quién la invocó

```
#include <stdio.h>
```

```
typedef struct
```

```
{  
    long dni;  
    char apyn[31];  
} t_info;
```

```
int cargar(t_info *p);  
void mostrar(const t_info *p);
```

```
void main(void)
```

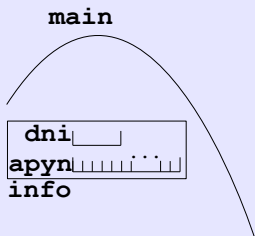
```
{  
    t_info info;  
  
    if(cargar(&info))  
        mostrar(&info);  
    else  
        puts("No se cargó información");  
}
```

```
    dni_____  
    apyn_____  
    t_info
```

```
int cargar(t_info *p)
```

```
{  
    printf("Cero o negativo, sale\n"  
          "D. N. I. : ");  
    fflush(stdin);  
    scanf("%ld", &p->dni);  
    if(p->dni > 0L)  
    {  
        printf("Apellido y Nombre : ");  
        fflush(stdin);  
        gets(p->apyn);  
        return 1; /* cargó */  
    }  
    return 0; /* no cargó */  
}
```

Argumentos de funciones - Punteros a estructuras .



Al terminar su ejecución la función `cargar`, devuelve 1 ó 0 a quién la invocó

Note que la función `cargar`, tiene dos `return`, hubiera sido mejor ...

```
#include <stdio.h>
```

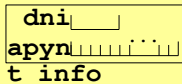
```
typedef struct
```

```
{  
    long dni;  
    char apyn[31];  
} t_info;
```

```
int cargar(t_info *p);  
void mostrar(const t_info *p);
```

```
void main(void)
```

```
{  
    t_info info;  
  
    if(cargar(&info))  
        mostrar(&info);  
    else  
        puts("No se cargó información");  
}
```



```
int cargar(t_info *p)  
{  
    printf("Cero o negativo, sale\n"  
          "D. N. I. : ");  
    fflush(stdin);  
    scanf("%ld", &p->dni);  
    if(p->dni > 0L)  
    {  
        printf("Apellido y Nombre : ");  
        fflush(stdin);  
        gets(p->apyn);  
        /* return 1; */  
    }  
    /* return 0; */  
}
```

Argumentos de funciones - Punteros a estructuras .

main

dni
apyn
info

Al terminar su ejecución la función **cargar**, devuelve **1** ó **0** a quién la invocó

Note que la función **cargar**, tiene dos **return**, hubiera sido mejor con un único punto de salida .

```
#include <stdio.h>
```

```
typedef struct
```

```
{  
    long dni;  
    char apyn[31];  
} t_info;
```

```
int cargar(t_info *p);  
void mostrar(const t_info *p);
```

```
void main(void)
```

```
{  
    t_info info;  
  
    if(cargar(&info))  
        mostrar(&info);  
    else  
        puts("No se cargó información");  
}
```

dni
apyn
t_info

```
int cargar(t_info *p)  
{  
    printf("Cero o negativo, sale\n"  
        "D. N. I. : ");  
    fflush(stdin);  
    scanf("%ld", &p->dni);  
    if(p->dni > 0L)  
    {  
        printf("Apellido y Nombre : ");  
        fflush(stdin);  
        gets(p->apyn);  
    }  
    return p->dni > 0L;  
}
```

Argumentos de funciones - Punteros a estructuras .

main

dni_____
apyn_____
info

Al terminar de ejecutarse cargar con el control de ejecución en main, si devolviera un valor distinto de cero, el if evalúa que la condición es Falsa

```
#include <stdio.h>
```

```
typedef struct
```

```
{  
    long dni;  
    char apyn[31];  
} t_info;
```

```
int cargar(t_info *p);  
void mostrar(const t_info *p);
```

```
void main(void)
```

```
{  
    t_info info;  
    if(cargar(&info))  
        mostrar(&info);  
    else  
        puts("No se cargó información");  
}
```

dni_____
apyn_____
t_info

```
int cargar(t_info *p)  
{  
    printf("Cero o negativo, sale\n"  
        "D. N. I. : ");  
    fflush(stdin);  
    scanf("%ld", &p->dni);  
    if(p->dni > 0L)  
    {  
        printf("Apellido y Nombre : ");  
        fflush(stdin);  
        gets(p->apyn);  
    }  
    return p->dni > 0L;  
}
```


Argumentos de funciones - Punteros a estructuras .

main

dni
apyn
info

Al terminar de ejecutarse cargar con el control de ejecución en main, si devolviera un valor distinto de cero, el if evalúa que la condición es Falsa con lo cual se muestra el mensaje de error

```
#include <stdio.h>
```

```
typedef struct
```

```
{  
    long dni;  
    char apyn[31];  
} t_info;
```

```
int cargar(t_info *p);  
void mostrar(const t_info *p);
```

```
void main(void)
```

```
{  
    t_info info;  
  
    if(cargar(&info))  
        mostrar(&info);  
    else  
        puts("No se cargó información");  
}
```

dni
apyn
t_info

```
int cargar(t_info *p)  
{  
    printf("Cero o negativo, sale\n"  
        "D. N. I. : ");  
    fflush(stdin);  
    scanf("%ld", &p->dni);  
    if(p->dni > 0L)  
    {  
        printf("Apellido y Nombre : ");  
        fflush(stdin);  
        gets(p->apyn);  
    }  
    return p->dni > 0L;  
}
```

Argumentos de funciones - Punteros a estructuras .

main

dni _____
apyn _____
info

Al terminar de ejecutarse cargar con el control de ejecución en main, si devolviera un valor distinto de cero, el if evalúa que la condición es Falsa con lo cual se muestra el mensaje de error, de ser Verdad (si cargar no devuelve cero) se invoca a mostrar .

```
#include <stdio.h>
```

```
typedef struct
```

```
{  
    long dni;  
    char apyn[31];  
} t_info;
```

```
int cargar(t_info *p);  
void mostrar(const t_info *p);
```

```
void main(void)
```

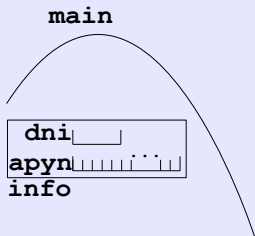
```
{  
    t_info info;  
  
    if(cargar(&info))  
        mostrar(&info);  
    else  
        puts("No se cargó información");  
}
```

dni _____
apyn _____
t_info

```
int cargar(t_info *p)  
{  
    printf("Cero o negativo, sale\n"  
        "D. N. I. : ");  
    fflush(stdin);  
    scanf("%ld", &p->dni);  
    if(p->dni > 0L)  
    {  
        printf("Apellido y Nombre : ");  
        fflush(stdin);  
        gets(p->apyn);  
    }  
    return p->dni > 0L;  
}
```

Punteros

Argumentos de funciones - Punteros a estructuras .



Si se invoca a mostrar, en primer lugar se evalúa el operador &

```
#include <stdio.h>
```

```
typedef struct
```

```
{
    long dni;
    char apyn[31];
} t_info;
```

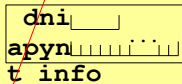
```
int cargar(t_info *p);
```

```
void mostrar(const t_info *p);
```

```
void main(void)
```

```
{
    t_info info;

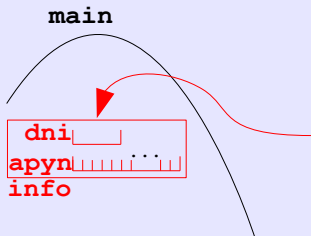
    if(cargar(&info))
        mostrar(&info);
    else
        puts("No se cargó información");
}
```



```
int cargar(t_info *p)
```

```
{
    printf("Cero o negativo, sale\n"
           "D. N. I. : ");
    fflush(stdin);
    scanf("%ld", &p->dni);
    if(p->dni > 0L)
    {
        printf("Apellido y Nombre : ");
        fflush(stdin);
        gets(p->apyn);
    }
    return p->dni > 0L;
}
```

Argumentos de funciones - Punteros a estructuras .



Si se invoca a mostrar, en primer lugar se evalúa el operador & que obtiene en qué dirección de memoria se encuentra la variable info

```
#include <stdio.h>
```

```
typedef struct
```

```
{
    long dni;
    char apyn[31];
} t_info;
```

```
int cargar(t_info *p);
void mostrar(const t_info *p);
```

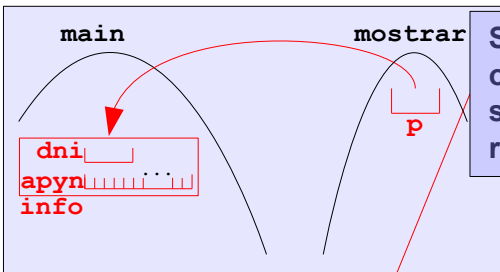
```
void main(void)
```

```
{
    t_info info;

    if(cargar(&info))
        mostrar(&info);
    else
        puts("No se cargó información");
}
```

```
int cargar(t_info *p)
{
    printf("Cero o negativo, sale\n"
           "D. N. I. : ");
    fflush(stdin);
    scanf("%ld", &p->dni);
    if(p->dni > 0L)
    {
        printf("Apellido y Nombre : ");
        fflush(stdin);
        gets(p->apyn);
    }
    return p->dni > 0L;
}
```

Argumentos de funciones - Punteros a estructuras .



Si se invoca a mostrar, en primer lugar se evalúa el operador & que obtiene en qué dirección de memoria se encuentra la variable info, y con este valor (la dirección de la variable info). se invoca a la función .

```
#include <stdio.h>
```

```
typedef struct
```

```
{
    long dni;
    char apyn[31];
} t_info;
```

```
int cargar(t_info *p);
```

```
void mostrar(const t_info *p);
```

```
void main(void)
```

```
{
    t_info info;

    if(cargar(&info))
        mostrar(&info);
    else
        puts("No se cargó información");
}
```

```
    dni____
    apyn____
t_info
```

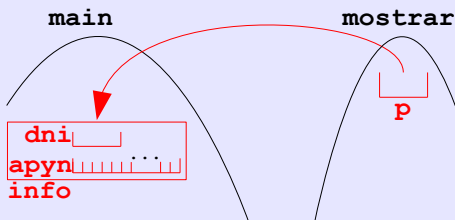
```
int cargar(t_info *p)
```

```
{
    printf("Cero o negativo, sale\n"
           "D. N. I. : ");
    fflush(stdin);
    scanf("%ld", &p->dni);
    if(p->dni > 0L)
    {
        printf("Apellido y Nombre : ");
        fflush(stdin);
        gets(p->apyn);
    }
    return p->dni > 0L;
}
```

```
void mostrar(const t_info *p)
```

```
{
    }
}
```

Argumentos de funciones - Punteros a estructuras .



Si se invoca a `mostrar`, en primer lugar se evalúa el operador `&` que obtiene en qué dirección de memoria se encuentra la variable `info`, y con este valor (la dirección de la variable `info`). se invoca a la función .

Lo que debe hacer la función es muy simple, tan sólo debe mostrar el contenido de los miembros de información de la variable del tipo `t_info` cuya dirección de memoria recibió en la variable `p` junto con los mensajes aclaratorios de qué es cada uno .

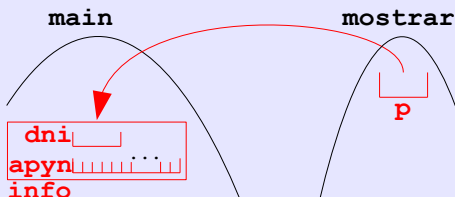
```
t_info,
```

```
int cargar(t_info *p);  
void mostrar(const t_info *p);
```

```
void main(void)  
{  
    t_info info;  
  
    if(cargar(&info))  
        mostrar(&info);  
    else  
        puts("No se cargó información");  
}
```

```
int cargar(t_info *p)  
{  
    printf("Cero o negativo, sale\n"  
          "D. N. I. : ");  
    fflush(stdin);  
    scanf("%ld", &p->dni);  
    if(p->dni > 0L)  
    {  
        printf("Apellido y Nombre : ");  
        fflush(stdin);  
        gets(p->apyn);  
    }  
    return p->dni > 0L;  
}  
  
void mostrar(const t_info *p)  
{  
    printf("Apellido y Nombre : %s\n"  
          "D. N. I. : %ld\n",  
          p->apyn, p->dni);  
}
```

Argumentos de funciones - Punteros a estructuras .



Si se invoca a `mostrar`, en primer lugar se evalúa el operador `&` que obtiene en qué dirección de memoria se encuentra la variable `info`, y con este valor (la dirección de la variable `info`). se invoca a la función .

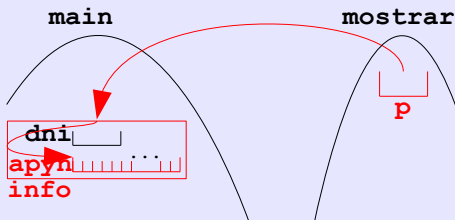
Lo que debe hacer la función es muy simple, tan sólo debe mostrar el contenido de los miembros de información de la variable del tipo `t_info` cuya dirección de memoria recibió en la variable `p` junto con los mensajes aclaratorios de qué es cada uno .

Cuando se ejecute el `printf`, antes se evaluarán sus argumentos, con lo que el primero es la dirección de memoria de la cadena de control de impresión

```
int cargar(t_info *p)
{
    printf("Cero o negativo, sale\n"
           "D. N. I. : ");
    fflush(stdin);
    scanf("%ld", &p->dni);
    if(p->dni > 0L)
    {
        printf("Apellido y Nombre : ");
        fflush(stdin);
        gets(p->apyn);
    }
    return p->dni > 0L;
}

void mostrar(const t_info *p)
{
    printf("Apellido y Nombre : %s\n"
           "D. N. I. : %ld\n",
           p->apyn, p->dni);
}
```

Argumentos de funciones - Punteros a estructuras .



Si se invoca a `mostrar`, en primer lugar se evalúa el operador `&` que obtiene en qué dirección de memoria se encuentra la variable `info`, y con este valor (la dirección de la variable `info`). se invoca a la función .

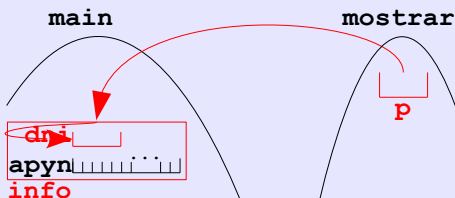
Lo que debe hacer la función es muy simple, tan sólo debe mostrar el contenido de los miembros de información de la variable del tipo `t_info` cuya dirección de memoria recibió en la variable `p` junto con los mensajes aclaratorios de qué es cada uno .

Cuando se ejecute el `printf`, antes se evaluarán sus argumentos, con lo que el primero es la dirección de memoria de la cadena de control de impresión, el segundo la dirección de comienzo del miembro `apyn` de la variable `info` de `main`

```
int cargar(t_info *p)
{
    printf("Cero o negativo, sale\n"
           "D. N. I. : ");
    fflush(stdin);
    scanf("%ld", &p->dni);
    if(p->dni > 0L)
    {
        printf("Apellido y Nombre : ");
        fflush(stdin);
        gets(p->apyn);
    }
    return p->dni > 0L;
}

void mostrar(const t_info *p)
{
    printf("Apellido y Nombre : %s\n"
           "D. N. I. : %ld\n",
           p->apyn, p->dni);
}
```


Argumentos de funciones - Punteros a estructuras .



Si se invoca a `mostrar`, en primer lugar se evalúa el operador `&` que obtiene en qué dirección de memoria se encuentra la variable `info`, y con este valor (la dirección de la variable `info`). se invoca a la función .

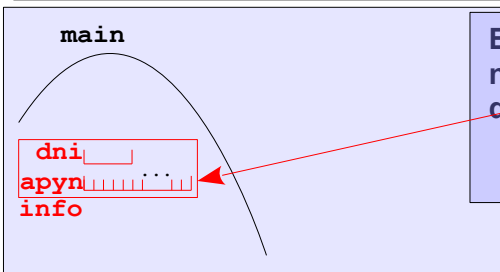
Lo que debe hacer la función es muy simple, tan sólo debe mostrar el contenido de los miembros de información de la variable del tipo `t_info` cuya dirección de memoria recibió en la variable `p` junto con los mensajes aclaratorios de qué es cada uno .

Cuando se ejecute el `printf`, antes se evaluarán sus argumentos, con lo que el primero es la dirección de memoria de la cadena de control de impresión, el segundo la dirección de comienzo del miembro `apyn` de la variable `info` de `main`, y el tercero será el miembro `dni` de la variable `info` de `main` .

```
int cargar(t_info *p)
{
    printf("Cero o negativo, sale\n"
           "D. N. I. : ");
    fflush(stdin);
    scanf("%ld", &p->dni);
    if(p->dni > 0L)
    {
        printf("Apellido y Nombre : ");
        fflush(stdin);
        gets(p->apyn);
    }
    return p->dni > 0L;
}

void mostrar(const t_info *p)
{
    printf("Apellido y Nombre : %s\n"
           "D. N. I. : %ld\n",
           p->apyn, p->dni);
}
```

Argumentos de funciones - Punteros a estructuras .



En los próximos dibujos que hagamos para ilustrar nuestros ejemplos, reemplazaremos esta ilustración de **variables** que responden a estructuras

```
#include <stdio.h>
```

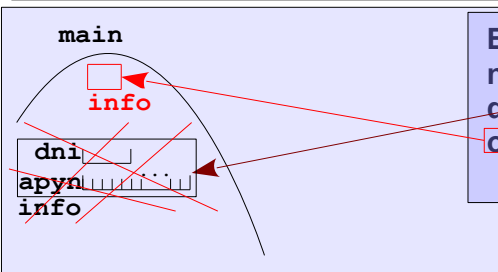
```
typedef struct  
{  
    long dni;  
    char apyn[31];  
} t_info;
```

```
int cargar(t_info *p);  
void mostrar(const t_info *p);
```

```
void main(void)  
{  
    t_info info;  
  
    if(cargar(&info))  
        mostrar(&info);  
    else  
        puts("No se cargó información");  
}
```

```
int cargar(t_info *p)  
{  
    printf("Cero o negativo, sale\n"  
        "D. N. I. : ");  
    fflush(stdin);  
    scanf("%ld", &p->dni);  
    if(p->dni > 0L)  
    {  
        printf("Apellido y Nombre : ");  
        fflush(stdin);  
        gets(p->apyn);  
    }  
    return p->dni > 0L;  
}  
  
void mostrar(const t_info *p)  
{  
    printf("Apellido y Nombre : %s\n"  
        "D. N. I. : %ld\n",  
        p->apyn, p->dni);  
}
```

Argumentos de funciones - Punteros a estructuras .



En los próximos dibujos que hagamos para ilustrar nuestros ejemplos, reemplazaremos esta ilustración de variables que responden a estructuras, por esta otra

```
#include <stdio.h>
```

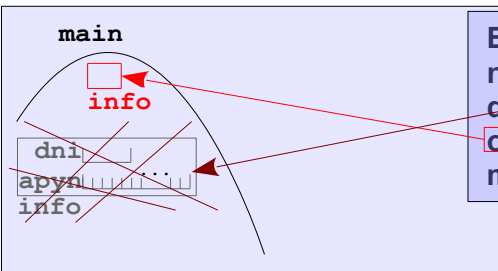
```
typedef struct  
{  
    long dni;  
    char apyn[31];  
} t_info;
```

The diagram shows a memory layout for a `t_info` structure. It contains a box labeled 'dni' and a box labeled 'apyn' and 't_info'.

```
int cargar(t_info *p);  
void mostrar(const t_info *p);  
  
void main(void)  
{  
    t_info info;  
  
    if(cargar(&info))  
        mostrar(&info);  
    else  
        puts("No se cargó información");  
}
```

```
int cargar(t_info *p)  
{  
    printf("Cero o negativo, sale\n"  
        "D. N. I. : ");  
    fflush(stdin);  
    scanf("%ld", &p->dni);  
    if(p->dni > 0L)  
    {  
        printf("Apellido y Nombre : ");  
        fflush(stdin);  
        gets(p->apyn);  
    }  
    return p->dni > 0L;  
}  
  
void mostrar(const t_info *p)  
{  
    printf("Apellido y Nombre : %s\n"  
        "D. N. I. : %ld\n",  
        p->apyn, p->dni);  
}
```

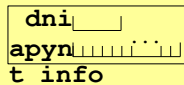
Argumentos de funciones - Punteros a estructuras .



En los próximos dibujos que hagamos para ilustrar nuestros ejemplos, reemplazaremos esta ilustración de variables que responden a estructuras, por esta otra, sin olvidar que responden a este tipo de información .

```
#include <stdio.h>
```

```
typedef struct  
{  
    long dni;  
    char apyn[31];  
} t_info;
```



```
int cargar(t_info *p);  
void mostrar(const t_info *p);
```

```
void main(void)  
{  
    t_info info;  
  
    if(cargar(&info))  
        mostrar(&info);  
    else  
        puts("No se cargó información");  
}
```

```
int cargar(t_info *p)  
{  
    printf("Cero o negativo, sale\n"  
          "D. N. I. : ");  
    fflush(stdin);  
    scanf("%ld", &p->dni);  
    if(p->dni > 0L)  
    {  
        printf("Apellido y Nombre : ");  
        fflush(stdin);  
        gets(p->apyn);  
    }  
    return p->dni > 0L;  
}  
  
void mostrar(const t_info *p)  
{  
    printf("Apellido y Nombre : %s\n"  
          "D. N. I. : %ld\n",  
          p->apyn, p->dni);  
}
```

