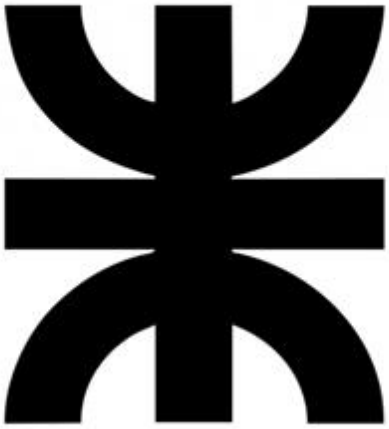


UNIVERSIDAD TECNOLOGICA NACIONAL

FACULTAD REGIONAL LA PLATA

# MANUAL BÁSICO DE STANDARD C

Federico Nicolás Moradillo

1/6/2018 Versión 1.0

UNIVERSIDAD TECNOLÓGICA NACIONAL

# Información del documento


Autor: Federico Nicolás Moradillo

Revisor: Ing. Félix Miguel Paternoster

Diseño gráfico: M. Belén Meyer


## Acreditación:

Este documento se encuentra a la espera de ser revisado y reconocido por la cátedra de Algoritmos y Estructuras de Datos de la Universidad Tecnológica Nacional Facultad Regional La Plata como documentación oficial de la misma.

## Bibliografía de referencia:


- https://www.tutorialspoint.com/cprogramming/index.htm
- https://www.cprogramming.com/tutorial/c-tutorial.html
- https://stackoverflow.com/
- https://www.geeksforgeeks.org/c-language-set-1-introduction/
- Estándar de programación en C ISO/IEC 9899:2011

# Historial de versiones

| Fecha | Versión | Cambios introducidos |
|---|---|---|
| 01/06/2018 | 1.0 | Confección oficial del documento |
| | | |
| | | |
| | | |
| | | |
| | | |
| | | |

# Prólogo

Esta es una guía de uso general de Standard C centrada en el manejo del lenguaje y sus características. Por tanto, se dan por sabidos conceptos básicos de programación y se ahonda en la sintaxis propia y el manejo de C.

Se usará a modo de comparación el lenguaje Pascal, debido a que es el lenguaje imperativo académico actual y con el que se tiene más familiaridad, además de poseer ambos lenguajes una sintaxis y estructura similares hasta cierto punto. Esto no significa que un conocimiento previo de Pascal sea obligatorio para entender el contenido de esta guía, pero servirá de facilidad para los ejemplos.

A lo largo del manual se detallarán las claras diferencias que presenta C respecto a los lenguajes convencionales dado su bajo nivel, las cuales pueden resultar confusas en principio.

Este documento fue confeccionado mediante investigación sobre el lenguaje en distintas guías y sitios web de C. La respectiva bibliografía se menciona en la Información del documento. Todo el código desarrollado en el proceso de entender el lenguaje para la confección del manual, así como también ejercicios resueltos de la práctica de Algoritmos y Estructuras de Datos, está disponible en el siguiente repositorio de Github:

- https://github.com/Zeriel/Aprendiendo-C/

# Índice

# 1. Introducción

## 1.1. Qué es C

C es un lenguaje de programación imperativo desarrollado en 1972 por Dennis Ritchie, sucesor del lenguaje B. Es considerado de medio nivel ya que, si bien es un lenguaje de alto nivel, fue pensado como intermediario del lenguaje ensamblador. Por esto, es un lenguaje que produce código de alta eficiencia ya que es fácil de convertir a lenguaje máquina con unas pocas instrucciones. Además, muchos compiladores incluyen extensiones para poder escribir código ensamblador dentro del código de C y operar directamente con el Sistema Operativo y el hardware.

Los usos de C en la actualidad son muy variados. Desde su uso original como lenguaje de desarrollo de software base (Sistema Operativo y programas que interactúan con este) es también usado para desarrollar firmware de productos electrónicos que poseen microcontroladores, para crear compiladores de más alto nivel, para desarrollar software aplicativo, entre otros.

Muchos lenguajes actuales están influenciados total o parcialmente en la estructura de C. Estos lenguajes son usualmente referidos como "C-like" o de la "Familia C" e incluyen a los derivados de C (C++, Objective-C, etc), Java, Processing, PHP, y más. Estos suelen compartir la estructura y sintaxis de C, aunque no incluyen los manejos de bajo nivel.

Hay que tener siempre en mente que C es un lenguaje *Case Sensitive*, por lo que las mayúsculas y minúsculas erróneas pueden causar errores en la compilación.

## 1.2. Compiladores

Existen múltiples compiladores y entornos para programar en lenguaje C, algunos específicos de Standard C (como GCC) y otros que permiten programación en C y C++ (como el entorno Visual Studio). En este manual se utilizará el Pelles C para Windows, que sólo admite código C.

## 1.3. Librerías

Las librerías en C agregan funciones para simplificar las operaciones que, de otra manera, requerirían codificación más avanzada. Se invocan con la sentencia "*#include*".

Por cuestiones de portabilidad, la mayoría de las librerías siguen un estándar general, lo que permite ejecutar un mismo código en distintos Sistemas Operativos como Windows y Linux. A continuación, se explican las librerías utilizadas a lo largo del manual:

- *<stdio.h>*: Es la librería estándar de entrada y salida (**st**an**d**ard **i**nput **o**utput). De esta librería obtenemos funciones básicas para poder leer y escribir información, operar con archivos, esperar tecla en consola y demás.
- *<conio.h>*: Es otra librería de entrada y salida pensada para la consola de sistemas Windows (**con**sole **i**pout **o**utput). Esta librería ***no*** es estándar y no es reconocida en compiladores de otros sistemas como GCC de Linux. De esta obtendremos funciones específicas como el limpiar pantalla. Sus operaciones son fácilmente identificadas por un "_" al principio.
- *<string.h>*: Agrega un conjunto de operaciones que facilitan el manejo de las cadenas de caracteres, que en C se trabajan como arreglos de caracteres.
- *<stdlib.h>*: Es la librería estándar (**st**an**d**ard **lib**rary) de propósito general de C. Incluye funciones para trabajar sobre la memoria dinámica y el control de procesos. La emplearemos para trabajar con listas.

Todas las librerías tienen la extensión .h, que viene de *header*. Los programas fuentes de C tienen la extensión .c.

## 1.4. Tipos de Datos

En C, los tipos de datos son similares a Pascal, salvo por algunas diferencias. Lo que sí es muy notorio y debe aclararse es que cada tipo de dato, además de su nombre nativo usado en la declaración, posee un carácter específico que se usa para definir explícitamente el tipo de dato en las funciones de lectura e impresión en pantalla, por ejemplo.

Los tipos de datos en C y su respectivo carácter son:

- *Int*: El tipo entero, su carácter es la letra "d". No se diferencia del Integer en Pascal.
- *Char*: El tipo carácter, que almacena un carácter individual. Su carácter es la letra "c".
- *Float*: El tipo flotante, equivalente al real en Pascal. Su carácter es la letra "f". Para acotar la cantidad de decimales en un muestreo se utiliza ".xf", donde "x" es la cantidad de dígitos decimales.

Estos son los tipos nativos de C. Nótese que faltan los tipos "booleano" y "cadena de caracteres"; esto es porque nativamente no existen en C.

Para los booleanos, C emplea como condición booleana cualquier entero distinto de 0 para *TRUE* y 0 para *FALSE*. Por tanto, se podría decir que una variable booleana puede definirse mediante una variable entera que se evalúe respecto a 0. También existen librerías que definen el tipo *bool*.

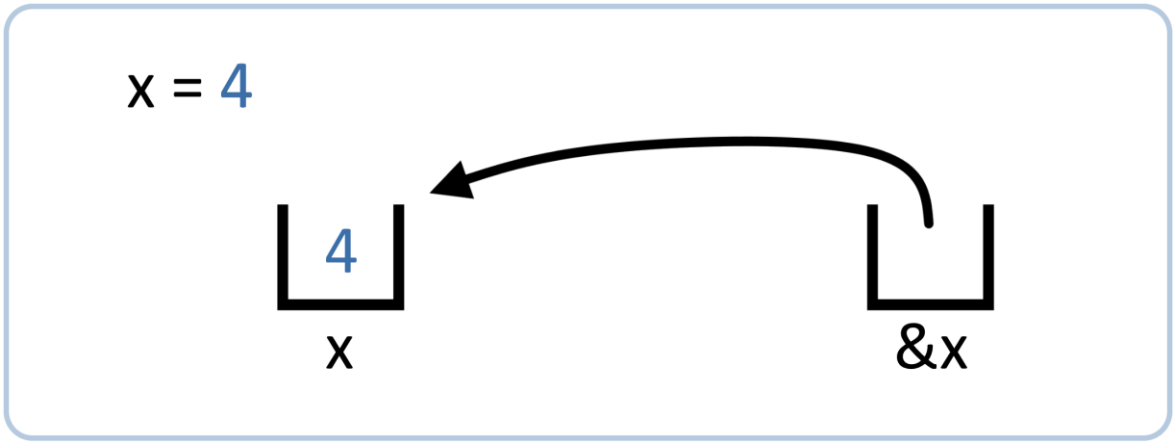
Para las cadenas de caracteres, se debe definir un arreglo de tipo *char* para almacenar cada carácter de la palabra en cada una de las posiciones del arreglo. Este tipo de "*string*" es diferenciado del tipo *char* con el carácter "s".

## 1.5. Caracteres especiales

Uno de los detalles más visibles ni bien se comienza a trabajar con C es la necesidad de emplear ciertos caracteres especiales ya que, a diferencia de otros lenguajes como Pascal, en C se debe aclarar (en la mayoría de los casos) si estamos haciendo referencia al valor de una variable o a la dirección en donde está almacenada para poder asignarle un dato.

- Et / Ampersand

El carácter Et ("&"), que de aquí en más lo llamaremos por su nombre inglés ampersand (dado que así es como lo encontraremos en la mayoría de las bibliografías), es el símbolo que C utiliza para operar con la dirección de memoria de una variable. Así, si tenemos una variable "x" cargada con algún dato, "&x" indicaría que estamos operando con la dirección en memoria de "x" y no con el valor que contiene.




- Porcentaje

El símbolo de porcentaje (%) aclara el tipo de dato esperado en las funciones de impresión y lectura, ya que se debe especificar explícitamente el tipo de la variable que se usa en ambos casos. Este símbolo se emplea con la letra asignada a cada tipo de dato, por ejemplo, "%d" para especificar que el tipo de la variable involucrada es entero.

- Asterisco

El asterisco (*) se emplea para definir y operar con variables de tipo puntero, similar, en cierta forma, al uso del circunflejo (^) en Pascal.

- Flecha

La flecha se forma con un guion y el símbolo de mayor, siendo entonces “->”. Este símbolo es usado en los punteros a registro (como las listas) para acceder a cada campo del registro que el puntero apunta, por ejemplo “puntero->campo”.

Tanto el ampersand como el porcentaje se utilizarán en todo el manual. El asterisco y la flecha, en cambio, sólo lo veremos cuando se aborden los Capítulos 8 y 9 de Modularización y Punteros, respectivamente.

## 1.6. Conectores lógicos

En Pascal, los conectores lógicos se basan en el habla inglesa y por ende son textuales. En C, por otro lado, se opta por una forma más genérica de codificación, y por ello se utilizan símbolos para los mismos. Los operadores/conectores en C son:

| Conector | Símbolo |
|---|---|
| **Igual** | == |
| **Distinto** | != |
| **Not** | ! |
| **Menor** | < |
| **Menor o igual** | <= |
| **Mayor** | > |
| **Mayor o igual** | >= |
| **AND** | && |
| **OR** | \|\| |

# 2. Lenguaje C

## 2.1. Definición del programa

Retomando Pascal a modo de comparación, el programa se define de la siguiente manera:

```
program nombrePrograma;   //El encabezado del programa con el nombre del mismo
uses librería;            //La lista de librerías a invocar
var
//Declaración de variables globales
Begin                     //El bloque principal del programa
   [bloque de código]
end.
```

En C, la definición es más simplificada. El nombre del programa no se especifica en el código, y el *uses* es reemplazado por los *#include* por cada librería. El cuerpo principal se define como un procedimiento más con el nombre reservado "*main*", sin parámetros, y las variables globales se declaran fuera de todos los procedimientos, junto a los *#include*.

Entonces, un programa en C sería:

```
#include <librería.h>     //Invocación a las librerías
//Declaración de variables globales
int main(){                     //El bloque principal del programa
//Declaración de variables locales a main()
   [bloque de código]
return 0;
}
```

Hay que hacer una serie de aclaraciones respecto al código anterior. Como anticipamos, el código en C carece de un encabezado con su nombre, comenzando directamente por la invocación de librerías y la declaración de variables globales. Ignoremos por ahora la palabra "*int*" en el *main*, se explicará en el Capítulo 8 de Modularización.

Puede apreciarse a priori que el cuerpo principal se define como el procedimiento *main* y que su correspondiente código se halla encerrado en llaves ({}). Las llaves son la forma en que C inicia bloques de código, siendo la forma de Pascal el par BEGIN-END. También vemos en "*return 0;*" que el cierre de sentencias, tanto en Pascal como en C, se realiza con el punto y coma (;).

Por último, podemos ver que la última línea de código es la ya mencionada *return 0*. *Return* es la forma en la que las funciones de C devuelven un valor a quien las invocó. Por convención se suele emplear un *return 0* al final del *main* con el fin de verificar si la ejecución del programa fue correcta. Esta verificación se basa en que, si durante la ejecución ocurrió algún error, el valor que retornará *main* será distinto de cero.

Como detalle adicional, nótese que las líneas de comentarios son idénticas a Pascal (//comentario). Los bloques de comentarios, en cambio, se realizan usando el asterisco y la barra lateral o slash (/*un bloque*/).

## 2.2. Operaciones básicas

Llamamos operaciones básicas a las funciones que empleamos para leer información del usuario y para mostrar información en la consola. Mientras que en Pascal estas funciones eran simples de usar, en C presentan cierta rigidez en cuanto al orden de sus parámetros.

### 2.2.1. Printf()

*Printf* es la función en C incluida en la librería *<stdio.h>* que nos permitirá mostrar información en la consola, como "*write"* en Pascal.

Recordando Pascal, el *write* lleva como parámetro la información a mostrar encerrada en comillas, y de ser necesario concatenar el valor de una variable, la misma puede separar el texto entre comillas con el uso de una coma. De manera que la sintaxis del *write* de Pascal es:

```
write("Esto es texto, la variable concatenada tiene: ", nomVariable," como valor");
```

Esta libertad que nos brinda Pascal en el *write* no se aplica en C. Para poder mostrar información con *printf* se debe respetar que primero se coloca el texto entre comillas y luego las variables, no de otra forma. Para poder identificar dónde concatenar el valor de cada variable, se usan los caracteres de cada tipo de dato explicados en el Capítulo 1.4.

Entonces, supongamos que la variable "nomVariable" es de tipo entero, su carácter en C es la letra "d" y se escribiría de la siguiente forma:

```
printf("Esto es texto, la variable concatenada tiene: %d como valor asignado", nomVariable);
```

El "%d" le dice a la función que en esa ubicación está esperando un dato de tipo entero, el cual será el parámetro fuera de las comillas. Se pueden concatenar varias variables con este método, siempre que se tenga en cuenta que la asignación del tipo esperado y la variable es secuencial; es decir, por cada "%tipo" buscará una variable y se asignará el valor, en el orden de izquierda a derecha.

Si se quisiera dejar una línea en Pascal, bastaría con cambiar la función *write* por *writeln*. En C, para dejar una línea se debe incluir en el texto entre comillas "\n" en donde deseemos generar el corte de línea. Es posible, aunque engorroso, escribir un texto completo en un solo *printf* empleando \n para generar los cortes.

### 2.2.2. Scanf()

*Scanf* en C es la función incluida en la librería *<stdio.h>* que nos permitirá leer información del usuario y almacenarla en una variable, como *read/readln* en Pascal.

Esta función es similar a su homóloga en Pascal, con la diferencia que deberemos emplear los caracteres especiales porcentaje y ampersand; porcentaje para especificar el tipo de dato, y ampersand para almacenarlo en la dirección de memoria de la variable objetivo.

Si quisiéramos leer un número entero en la variable "num" en Pascal, el *readln* sería simplemente:

```
readln(num);
```

En C, sin embargo, esta misma operación se vuelve un poco más compleja:

```
scanf("%d", &num);
```

Como dijimos, primero especificamos qué tipo de dato queremos almacenar empleando el "%tipo" entre comillas. Luego, como segundo parámetro, enviamos la dirección de memoria de la variable "num" empelado el ampersand. Si el tipo especificado en la primera parte no coincide con el tipo de la variable en la segunda, el compilador dará error.

Hay que hacer una aclaración importante: este ejemplo se cumple para todos los tipos de datos excepto el *string*. Como ya dijimos en el capítulo 1.4, los *strings* son en realidad arreglos de caracteres, y en C un arreglo está definido por un puntero al primer elemento. Esto se verá en mayor detalle en el Capítulo 6 de Arreglos y Capítulo 9 de Punteros, por lo que por ahora basta con aclarar que, cuando se quiera leer un *string*, no se deberá utilizar el ampersand.

Entonces, si lo que queremos leer es un *string*, la forma de hacerlo es:

```
scanf("%s", nombre);            //Es un string, no uso ampersand
```

### 2.2.3. Getchar() y _getch()

Las funciones *getchar* y *_getch* son el equivalente a *readkey* de la librería "*crt*" en Pascal. Su función consiste en detener la consola, para poder visualizar mensajes de salida, hasta que se ingrese una tecla y se continúe con la ejecución.

La razón por la cual hay dos funciones para el mismo objetivo radica en que la función estándar de C, *getchar()*, puede traer problemas al capturar datos ingresados previamente en una lectura de información si no se tiene cuidado a la hora de la codificación.

La solución más simple a esto es reemplazar *getchar()* por *_getch()*, la cual viene incluida en la librería especial de Windows *<conio.h>.* La principal desventaja de usar *_getch()* es que el código ya no será portable a sistemas fuera de Windows.

Otra diferencia entre ambas funciones es que *getchar(),* en su funcionamiento normal, espera que se ingrese un valor por teclado y se presione *enter* para recibirlo, mientras que *getch()* toma cualquier tecla presionada y automáticamente la recibe para continuar con la ejecución.

### 2.2.4. _clrscr()

De igual manera que con *_getch()*, la función *_clrscr()* (**clr**ear **scr**een) está incluida en *<conio.h>* y no es portable a otros sistemas fuera de Windows. Su función es idéntica a su par *clrscr* de Pascal: limpiar la consola de Windows de todo el texto acumulado hasta el momento.

# 3. Variables

## 3.1. Declaración

Como se vio en el Capítulo 1, las variables pueden declarase globalmente fuera del *main()* o localmente dentro del mismo. En C la declaración de una variable es al revés que en Pascal; primero se declara el tipo de la variable y luego el nombre de la misma.

A continuación, veremos un ejemplo de variables declaradas global y localmente:

```
#include <stdio.h>
#include <conio.h>
int num1;           //Declaración de un número entero
char letra=a;       //Declaración de un carácter y asignación de valor inicial
char palabra[15];   //Declaración de un string
int main(){
      float num2, num3;   //Declaración de dos números flotantes
      return 0;
}
```

En el ejemplo anterior *num1*, *letra* y *palabra* son variables globales, mientras que *num2* y *num3* son variables locales a *main*. Además, *letra* se declara con un valor inicial; es posible declarar una variable con un valor inicial fijo si se le asigna el valor en la misma declaración. La asignación en C es "=", a diferencia de Pascal que usa la sintaxis ":=".

Nótese que "*palabra*" es un *string*; como dijimos anteriormente los *strings* en C son en realidad arreglos de caracteres, y la forma de definir una variable tipo arreglo es con los corchetes ([]) al final de la variable. En el Capítulo 6 de Arreglos se verá de manera más detallada.

Cuando declaramos variables locales en el *main* (o en cualquier módulo) debemos hacerlo al principio de cualquier otra línea de código, ya que C no permite declarar variables en mitad de ejecución del módulo.

## 3.2. Aplicación

Ahora que sabemos cómo declarar variables podemos empelar las funciones vistas en el Capítulo 2.2 para crear una aplicación simple, donde se pida datos al usuario y se los trabaje generando una salida.

```
#include<stdio.h>
#include<conio.h>
int main(){
	int num1=2, num2, res;	//Declaro num1 con valor inicial, y num2 y res
	printf("Ingrese valor numérico: ");
	scanf("%d", &num2);
	_clrscr();
	res=num1+num2;
	printf("La suma de %d y %d es: %d", num1, num2, res);  //Véase el orden de izquierda
	                                                       //a derecha
	_getch();
	_clrscr();
	return 0;
}
```

La aplicación anterior le pide un valor al usuario en *num2*, la suma con *num1* y almacena el resultado en *res*.

## 3.3. Contadores y acumuladores

Con el objetivo de agilizar la programación, C define una manera más rápida de trabajar con contadores y acumuladores:

- cont= cont+1 puede reducirse a cont++. Lo mismo para decrementar con cont--.
- acum= acum+valor puede reducirse a acum+=valor.

Esta forma de trabajo puede verse reflejada en todos los lenguajes C-like, como Java o PHP.

# 4. Estructuras de Decisión

## 4.1. Si

La estructura SI/SINO no difiere mucho de su versión en Pascal, sólo debemos procurar que cada bloque booleano esté debidamente encerrado entre paréntesis, ya que de esta manera es como operará correctamente la condición.

Una estructura básica de SI/SINO en C:

```
if ([condición]) {
	[bloque de código]
}
else {
	[Bloque de código]
}
```

Si quisiéramos evaluar múltiples condiciones:

```
if (([condicion1] && [condicion2]) || [condicion3]) {
	[Bloque de código]
}
```

Nótese los paréntesis. Cada evaluación booleana debe estar correctamente delimitada entre paréntesis, para que el compilador entienda lo que se quiere evaluar. En este caso, se evalúa que, tanto *condicion1* como *condicion2* sean *TRUE,* o que *condicion3* sea *TRUE* para ejecutar el bloque. Para la simbología, recordar la tabla de operadores booleanos del Capítulo 1.6.

## 4.2. Caso

El caso en C sí difiere bastante de su versión en Pascal en cuanto a sintaxis. La palabra reservada para el caso es *SWITCH* y cada opción posible va precedida de un *CASE*. Además, el cierre de cada opción del caso debe realizarse manualmente a través de un *BREAK* al final del bloque. La rama falsa es *DEFAULT*.

Ejemplo de uso de caso en C:

```
scanf("d%", dia);
swtich (dia){
	case 1:
		printf("Es lunes");
	break;
	case 2:
		printf("Es martes");
	break;
	…
	…
	case 7:
		printf("Es domingo");
	break;
	default:
		printf("Opción incorrecta");
	break;
}
```

El ejemplo anterior lee un número ingresado por el usuario, y devuelve el día de la semana que corresponde a ese número, o un error si se ingresa un número incorrecto.

El dato a validar en las opciones del caso debe ser un valor constante. Las comparaciones no son reconocidas.

# 5. Estructuras de Control

## 5.1. Repetición Incondicional

La estructura de repetición incondicional en C es muy particular, y está presente en todos los lenguajes C-Like. La variable de control de bucles puede declararse en la misma repetición y usarse de manera local.

La estructura del *FOR* en C es:

```
for (int varControl=ValorInicial; condicionSalida; modificacionVarControl){
      [Bloque de código]
}
```

Nótese el *int* que empleamos en la variable de control, esto es porque la estamos declarando localmente para el *FOR* como explicamos anteriormente.

El *FOR* de C consta de tres partes separadas por punto y coma:

- El valor inicial.
- La condición de salida.
- La forma en que se modificará la variable de control.

El valor inicial es similar a Pascal, simplemente define el valor que tendrá la variable de control al momento de iniciarse el bucle. Como ya dijimos, la variable de control puede estar ya declarada o declararse dentro del *FOR*.

La condición de salida es una expresión booleana que mantendrá el bucle, siempre y cuando sea *TRUE*. Por ejemplo, para una iteración de 10 ciclos se podría emplear “variableControl <= 10” si tenemos en cuenta que la variable de control comenzó con valor inicial 1.

La forma de modificación es la operación que se aplicará a la variable de control por cada vuelta del bucle. Para un *FOR* clásico que incrementa en 1 la variable de control hasta llegar a cierto valor, se puede usar el incremento “*variableControl++*”.

El siguiente ejemplo mostrará en pantalla la tabla de multiplicar del 1 al 10 de un número ingresado por el usuario:

```c
#include <stdio.h>
#include <conio.h>

int main(){
	int num;			//Variable local a main
	printf("Ingrese valor: ");	//Valor a realizar tabla
	scanf("%d", &num);		//Nótese el uso de caracteres especiales
	_clrscr();
	for (int i=1; i<=10; i++){	//Repetición  de 1 a 10
		printf("%d x %d = %d\n", num, i, num*i);
	}
	_getch();			//Para que no termine la ejecución sin mostrar
	return 0;
}
```

En C puede resultar un poco confuso que el *FOR*, que es una repetición incondicional, posea una condición booleana; contrario a Pascal, que sólo podía realizar bucles por incrementos o decrementos de la variable.

## 5.2. Repetición Condicional

La estructura de repetición condicional en C es prácticamente idéntica a su versión en Pascal, tanto para la repetición normal *WHILE* (repetir mientras) como para el caso particular *DO..WHILE* (repetir hasta – *repeat until*). En general, esta segunda repetición condicional no suele darse en la materia Algoritmos y Estructuras de Datos, por lo que se explicará brevemente cuando la ejemplifiquemos.

- *While*

La sintaxis del *while* en C es la siguiente:

```c
int true=1;
while (true){
	[bloque de código]
}
```

Como anticipamos, la sintaxis del *while* es casi idéntica a la de pseudocódigo y Pascal, por lo que no se considera que haya mucho que explicar sobre la misma.

Recordemos que, como explicamos en el Capítulo 1.4, en C no existe el tipo booleano nativamente, sino que evalúa 0 como *FALSE* y cualquier otro valor entero como *TRUE*. Por tanto, en este ejemplo empleamos una variable entera como nuestro “bool” para generar un bucle infinito en la iteración.

Respecto a las condiciones, se aplican las mismas reglas para los conectores lógicos

- *Do..while*

Antes de mostrar la sintaxis del *repetir hasta*, explicaremos brevemente su función. En la repetición condicional *mientras,* el bloque de código puede no ejecutarse nunca si la condición da *FALSE* la primera vez que se evalúa.

En la repetición *hasta*, la condición va al final de la estructura y no al principio, lo que permite que el bloque de código sea ejecutado mínimamente una vez antes de evaluar la condición de salida. Esta estructura es útil en ciertos casos, como la muestra de un menú en donde la primera validación no es necesaria.

Cuando se trabaje con *mientras* y *hasta* hay que tener claro que no siempre una repetición *mientras* se va a poder reemplazar con un *hasta*, en tanto que el *hasta* siempre es reemplazable por un *mientras*, que es el caso más general de repetición condicional.

Habiendo explicado ya la funcionalidad de la repetición *hasta*, su sintaxis es la siguiente:

```
int true=1;
do{
      [bloque de código]
}while(true);
```

Tener en cuenta que, como la condición del *do..while* va después del cierre de llave, lleva punto y coma.

## 5.3. Continue y Break

Ya vimos el uso del *break* con anterioridad, cuando se explicó *caso* en el Capítulo 3.2. Ahora veremos más en detalle tanto esta sentencia como la sentencia *continue,* en lo que a repetición compete.

El *continue* y el *break* son sentencias que pueden usarse dentro de una repetición para alterar la ejecución normal de cada bucle. Estas herramientas no suelen darse en Algoritmos y Estructuras de Datos, dado que dificultan entender el concepto de cada repetición al permitir manipularla sin importar las condiciones de salida. Se explicarán para que se tenga conocimiento de ellas, pero no se usarán en los ejemplos de este manual.

- *Continue*

El *continue* ignora todo el código restante que lo sucede cuando es encontrado por el compilador, y vuelve al inicio del bucle. Si se realiza en un *for*, se aplicará la modificación a la variable de control del mismo.

- *Break*

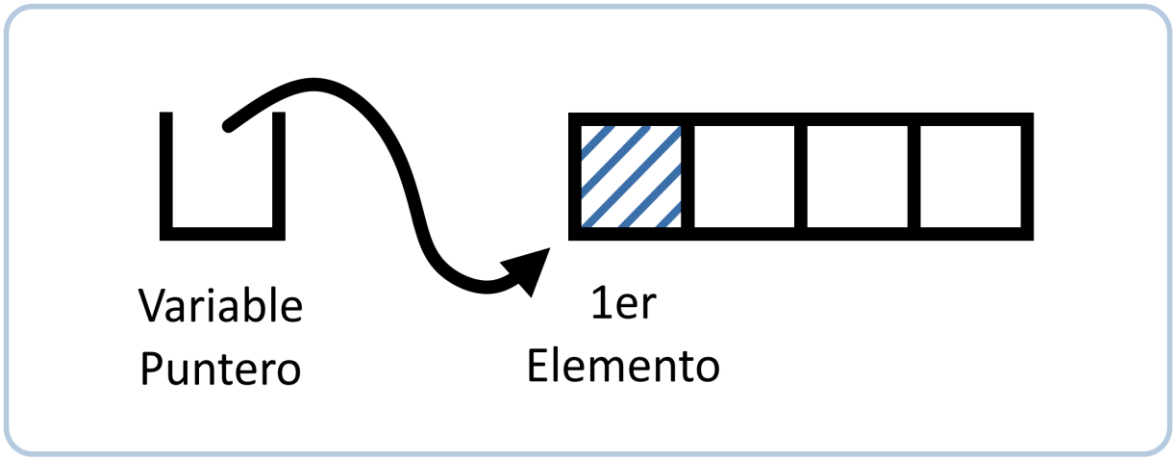
El *break* permite salir de una iteración en el momento que la sentencia es ejecutada por el compilador, sin importar el estado de las condiciones de salida del bucle.

Es obvio, habiendo explicado estas sentencias, el por qué podrían entorpecer el entendimiento conceptual de las estructuras de repetición, lo que no quita que su uso en determinados casos pueda ser más eficiente. Sin embargo, y como se aclaró al principio, no se usarán estas herramientas en los ejemplos de este manual.

# 6. Arreglos

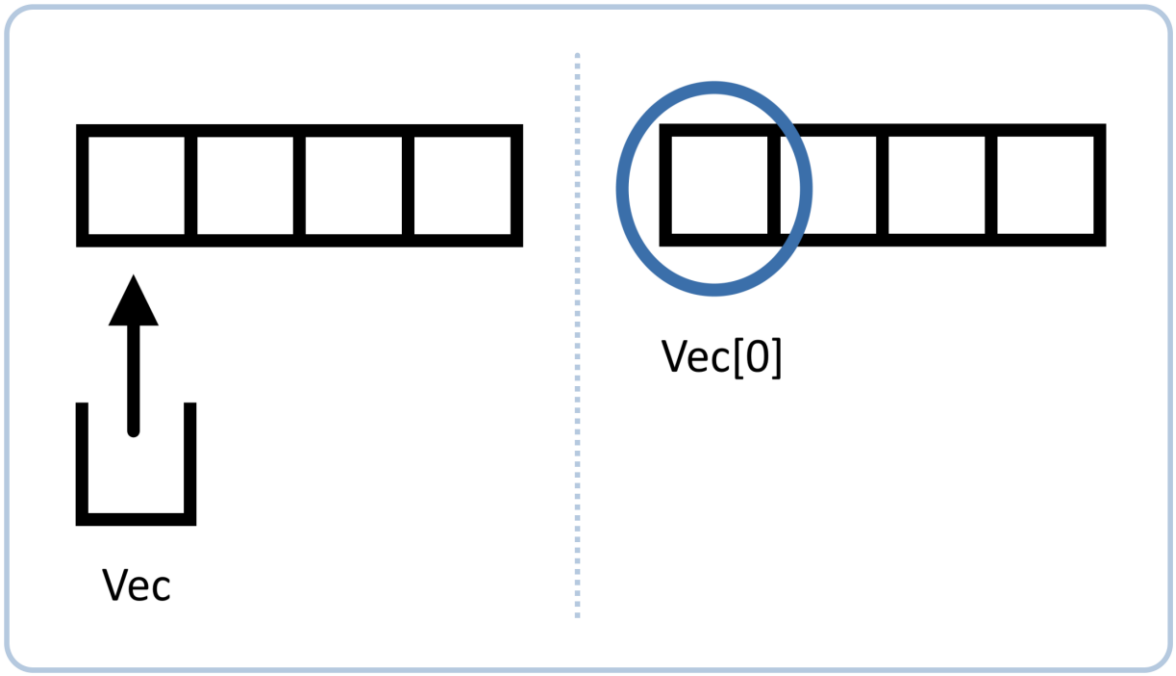
## 6.1. Definición

Primero que nada, para trabajar con arreglos en C hay que tener en cuenta que un arreglo es, en realidad, un puntero de memoria al primer elemento del mismo. Esto quiere decir que la variable tipo arreglo almacena una *dirección de memoria*, motivo por el cual, cuando se trabajó con una variable así (los *string*) en ejemplos anteriores, no se utilizó el ampersand.




Como ya se explicó, el ampersand se emplea para indicar que queremos operar con la dirección de memoria de una variable y no con el valor contenido en la misma. En la mayoría de los casos no habría problemas, pero cuando trabajamos con arreglos en su forma general (es decir, sin usar un índice) debemos omitir el ampersand ya que la variable, como se aclaró al principio, almacena como valor la dirección en memoria del primer elemento. La variable tipo arreglo *es un puntero*.

Cuando se trabaje con campos específicos del arreglo mediante un índice, sí usaremos el ampersand, ya que sería igual a trabajar con una variable del mismo tipo de dato que el arreglo.




Teniendo esto en claro, pasamos al código para definir un arreglo tipo vector y otro tipo matriz:

```
#include <stdio.h>
#include <conio.h>

int vecNum[5];      //Defino un arreglo tipo vector de enteros de 5 posiciones
int matNum[2][2];   //Defino un arreglo tipo matriz de enteros de 2x2 posiciones
int i, j;           //Defino dos enteros como índices

int main(){
      [bloque de código]
}
```

En este ejemplo vemos que, para definir el rango de un arreglo, se coloca dentro de los corchetes el número de posiciones del mismo. También vemos que, si en lugar de un arreglo queremos definir una matriz, basta solamente con agregar otro par de corchetes. Así, podemos definir arreglos de *n* dimensiones definiendo *n* pares de corchetes.

Tener en cuenta, cuando se trabaje con arreglos en C, que las posiciones se cuentan desde 0, no desde 1. Así, el arreglo *vecNum* del ejemplo, el cual tiene tamaño 5, tiene las posiciones 0 a 4.

Veamos ahora un ejemplo en donde se cargue un arreglo de enteros:

```
#include <stdio.h>
#include <conio.h>

int vecNum[5], i;         //Defino un arreglo tipo vector de enteros de 5 posiciones y un índice

int main(){
      for (i=0; i<5; i++){      //Recorro el arreglo
            vecNum[i]= 0;       //Inicializo con valor 0
      }
      for (i=0; i<5; i++){      //Realizo la carga del vector
            printf("Vector posicion: %d\n", i);
            printf("Ingrese valor entero: ");
            scanf("%d", &vecNum[i]);  //Nótese que en el scan el vector va con & porque use índice
            _clrscr();
      }
}
```

En este ejemplo, recorremos dos veces el arreglo "*vecNum*", primero para inicializarlo en cero y luego para cargar valores por consola.

En el *scanf,* como tipo de dato esperado debemos ingresar el tipo del arreglo, entero ("*%d*") en este caso.

Podríamos no haber definido un índice junto al vector y, en su lugar, definirlo localmente dentro de cada *FOR*, pero se hizo de esta manera para mantener una semejanza con Pascal y Pseudocódigo. A partir de ahora, los índices se definirán en el *FOR* a no ser que su uso sea necesario fuera del mismo.

Haciendo repaso sobre lo que ya dijimos al principio, si queremos operar con el dato almacenado en una posición del arreglo, se utiliza el ampersand dado que es semejante a trabajar con una variable. Si, en cambio, queremos trabajar con el arreglo en sí, no se utiliza el ampersand.

A grandes rasgos, y como recordatorio: si se utiliza un arreglo con índice se debe usar ampersand.

Por último, cabe aclarar que los arreglos pueden cargarse manualmente mediante un recorrido por el mismo, como hicimos en el ejemplo anterior, o crearse con datos precargados. Esto último no requiere el uso de un *FOR*, ya que los datos se definen en la declaración del arreglo. La forma de hacer esto es:

```
int vecNum[3] = {1,2,3};  //Defino un arreglo con datos ya cargados
```

En donde a nuestro arreglo vecNum le asignamos de forma predefinida los valores 1, 2 y 3, los cuales se cargarán secuencialmente en las posiciones 0, 1 y 2 del mismo, respectivamente.

## 6.2. Strings

Ya dijimos antes que los *strings* no existen como tales en C y, en su lugar, se utilizan arreglos de caracteres. Para poder operar con *strings* (ya sea compararlos, asignarle valor, copiar uno en otro y demás) debemos empelar la librería *<string.h>*.

De dicha librería utilizaremos las siguientes funciones:

- *strlen(str)*: Recibe un *string* y retorna su longitud (***str**ing **len**gth*).
- *strcmp(str1, str2)*: Recibe dos *strings*, los compara y retorna 0 si son iguales, más de 0 si *str1* es mayor que *str2* o menos de 0 si *str2* es mayor que *str1* (***str**ing **comp**are*).
- *strcpy(destino, fuente)*: Copia el contenido de fuente en destino. Lo utilizaremos para asginar un *string* en una variable (***str**ing **copy***).
- *strcat(destino, fuente)*: concatena el contenido de fuente al final de destino (***str**ing con**cat**enate*).

Veremos ahora un ejemplo donde se utilice esta librería para poder dar cierre a los *strings* de C:

```
#include <stdio.h>
#include <conio.h>
#include <string.h>

char cadena1[10], cadena2[10];

int main(){
	strcpy(cadena1, "hola");		//Asignamos un valor a cadena 1
	strcpy(cadena2, " mundo");		//Asignamos un valor a cadena 2
	strcat(cadena1, cadena2);		//Concatenamos ambos para formar "hola mundo"
	if (strcmp(cadena1, "hola mundo")==0){
		printf("La concatenacion funciono");
		_getch();
		_clrscr();
	}
	return 0;
}
```

Como se ve en el ejemplo, cargamos *cadena1* y *cadena2* con “hola” y “ mundo” respectivamente, lo concatenamos y lo comparamos para ver si efectivamente es igual a “hola mundo”. Si el código se ejecuta, se verá que efectivamente entra por la rama verdadera del *IF*.

La única manera que tenemos de cargar un *string* sin usar estas funciones es leyendo el dato directamente del usuario mediante el *scanf*, el cual sería:

```
printf("Ingrese nombre: ");
scanf("%s", nombre);			//Es un string, no uso ampersand
```

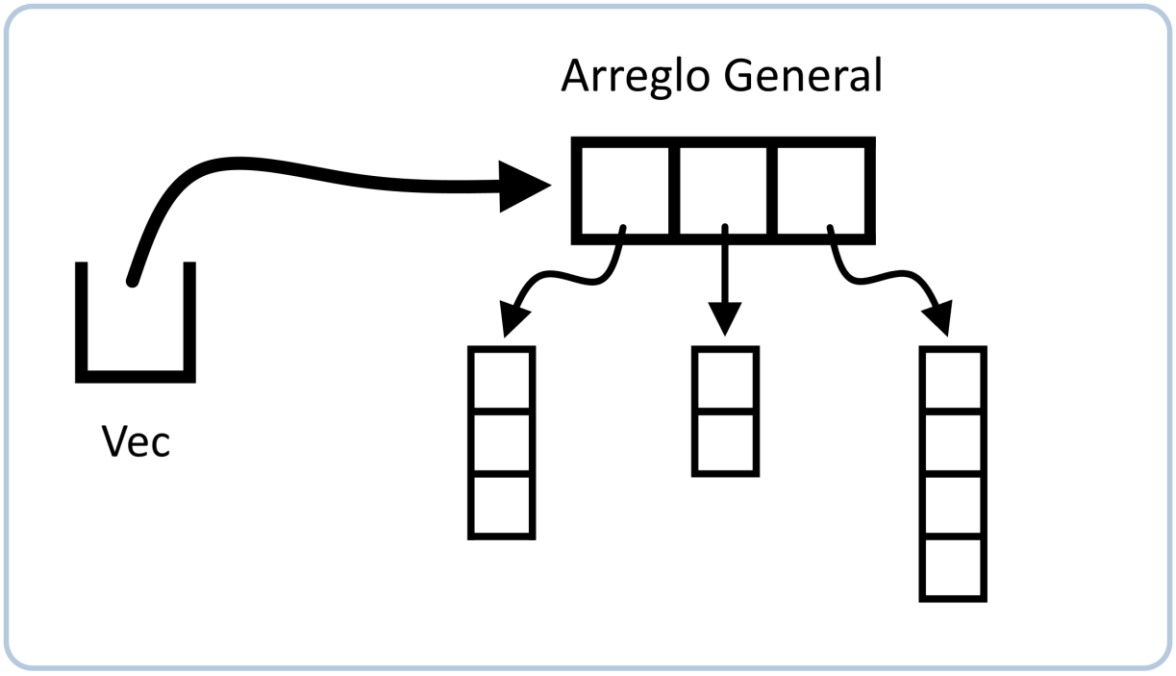
Dato muy importante: los *strings* añaden una posición adicional al arreglo, para marcar internamente el fin del mismo. Es decir, si definimos un *string* de diez posiciones como el del ejemplo, el mismo tendrá las posiciones 0-9 para almacenar caracteres y una posición interna "10", que marcará el fin del arreglo mediante un "/0".

No importa si se trabaja con una variable, un campo de registro o un campo de lista, si el tipo de dato almacenado es un *string*, se aplica todo lo visto en este capítulo.

## 6.3. Arreglo de arreglos

El manejo de arreglo de arreglos en C requiere conocimientos sobre el manejo de punteros, los cuales se verán en detalle el Capítulo 9. Por tanto, lo que veamos en esta sección del manual podrá resultar algo confusa, por lo que se recomienda volver a repasarla una vez que se haya entendido el uso de punteros.

Como sabemos, un arreglo es un puntero al primer elemento. Un arreglo de arreglos, entonces, no es más que un arreglo de punteros, donde cada puntero apunta a su vez a otro arreglo, el cual contiene los datos almacenados.




Se debe definir, entonces, un arreglo de punteros, luego los arreglos que serán apuntados, y finalmente vincular cada uno de estos arreglos con el puntero correspondiente en el arreglo general. Adicionalmente, se definirá un puntero auxiliar para acceder al arreglo que queramos recorrer en la i-ésima posición dentro del arreglo general.

Cabe aclarar que, como los arreglos son un tipo de dato homogéneo, tanto el arreglo general como cada arreglo que lo conforma deberán ser del mismo tipo de dato.

Seguramente todo esto parecerá muy confuso, pero quedará más claro una vez lo veamos plasmado en código. La definición de un arreglo de arreglos entonces será:

```
int *arregloDeArreglos[3];      //Defino un arreglo de tres punteros
                                //de tipo entero
```

En donde el asterisco nos indica que nuestro arreglo de tres posiciones enteras es, en realidad, un arreglo de tres posiciones que apuntan cada una a un entero.

El siguiente paso es crear cada uno de los arreglos que irán en cada uno de los punteros:

```
int arreglo1[3] = {1,2,3};  //Defino el primer arreglo con datos ya cargados
int arreglo2[3] = {4,5,6};  //Defino el segundo arreglo con datos ya cargados
int arreglo3[3] = {7,8,9};  //Defino el tercer arreglo con datos ya cargados
```

Estos arreglos no necesariamente deben ser del mismo tamaño. Los siguientes arreglos también son válidos:

```
int arreglo1[2] = {1,2};        //Defino un arreglo de tamaño 2
int arreglo2[4] = {3,4,5,6};    //Defino un arreglo de tamaño 4
int arreglo3[3] = {7,8,9};      //Defino un arreglo de tamaño 3
```

Ahora, definimos nuestro puntero auxiliar, con el cual podremos recorrer cada uno de estos arreglos:

```
int *punteroAArreglo;     //Defino la variable puntero que usaré para
                          //recorrer cada arreglo
```

Por último, asignamos cada uno de estos arreglos a nuestro arreglo general:

```
//Asigno cada arreglo a una posición del arreglo general
arregloDeArreglos[0] = arreglo1;
arregloDeArreglos[1] = arreglo2;
arregloDeArreglos[2] = arreglo3;
```

Ya tenemos definida nuestra estructura para el arreglo de arreglos. Si quisiéramos recorrerlo para mostrar sus datos, bastaría con emplear el siguiente código:

```
for (int i=0; i<3; i++){              //Un for por cada puntero del arreglo general
      printf("Arreglo: %d\n", i+1);   //El +1 corrige el arrancar desde la posición 0
      punteroAArreglo= arregloDeArreglos[i]; //Obtengo el arreglo actual mediante índice
      for (int j=0; j<3; j++){        //Recorro dicho arreglo para mostrar los datos

//Si los arreglos tuvieran tamaños distintos, cada uno llevaría su propio for
//Como yo los definí a todos iguales, puedo usar el mismo for para recorrerlos a los tres

            printf("Posicion: %d - Valor: %d\n", j+1, punteroAArreglo[j]);
            _getch();
      }
      _clrscr();
}
```

Nótese el uso que le damos al puntero auxiliar, igualándolo en cada iteración al arreglo actual que apunta el arreglo general a través del índice *i*.

Con esto puede darse cierre al tema de arreglo de arreglos. Se aconseja rever esta parte una vez entendido el Capítulo 9 de Punteros, para un mejor entendimiento de lo que se está haciendo.

# 7. Registros

## 7.1. Definición

Los registros en C son, a grandes rasgos, idénticos a los de Pascal. La palabra reservada en el Sistema para registros es *struct* y se usa, tanto para definir un registro, como para declarar una variable de dicho tipo.

Definición de registro en Pascal:

```
Persona = record
           apellido: string;
           nombre: string;
           edad: integer;
          end;
```

Definición de *struct* en C:

```
struct Persona{           //Definición de un registro (struct) en C
       char apellido[15];
       char nombre[15];
       int edad;
};
```

El registro del ejemplo define a una persona con tres campos: dos *strings* para apellido y nombre, y un entero para su edad. Como es una definición de un tipo de dato y no un bloque de código, la llave de cierre lleva un punto y coma.

El acceso a un campo de registro se realiza de la misma manera que en Pascal, con un punto seguido del nombre del campo:

```
printf("Ingrese nombre: ");      //Suponemos p una variable tipo Persona
scanf("%s", p.nombre);           //Asignación del usuario por consola
p.edad= 5;                       //Asignación directa a un campo del registro
```

Ejemplo de uso del registro *Persona* en una aplicación:

```c
#include <stdio.h>
#include <conio.h>
#include <string.h>

struct Persona{           //Definición de un registro (struct) en C
	char apellido[15];
	char nombre[15];
	int edad;
};

struct Persona p;         //Declaro una variable tipo Persona

int main(){
	strcpy(p.apellido, "Moradillo");       //Cargo un apellido
	strcpy(p.nombre, "Federico");          //Cargo un nombre
	p.edad= 23;                            //Cargo una edad
	printf("Datos de la persona:\n");
	printf("Apellido: %s\n", p.apellido);
	printf("Nombre:   %s\n", p.nombre);
	printf("Edad:     %d\n", p.edad);
	_getch();
	_clrscr();
	return 0;
}
```

En el ejemplo se declaró una variable tipo *Persona* y se cargó con datos, en este caso los míos.

Para declarar una variable tipo registro, se debe colocar primero el nombre del tipo de registro seguido del nombre de la variable que estamos declarando.

Tanto *nombre* como *apellido* se trabajaron con funciones de la librería *<string.h>*, mientras que *edad* se trabajó de manera normal, accediendo al campo con un punto y asignando el valor directamente.

Al igual que en Pascal, los campos de un registro pueden ser de tipos nativos o de tipos definidos por el usuario (arreglos, otros registros, listas).

## 7.2. Arreglo de Registros

Se pueden definir arreglos que contienen registros para trabajar con situaciones más complejas. La declaración de un arreglo de registros es tan simple como agregarle corchetes a una declaración normal de registro:

```
struct Persona{           //Definición de un registro (struct) en C
	char apellido[15];
	char nombre[15];
	int edad;
};

struct Persona vecP[10];  //Declaro un vector tipo Persona
```

En este ejemplo se declaró una variable *vecP* de tipo Persona que, además, es un arreglo de 10 posiciones. Es decir, es un vector que contiene a 10 personas.

Como no hay más que explicar, pasamos directamente a un ejemplo práctico donde se cargan 10 personas en el arreglo y luego se las muestra por consola:

```
#include <stdio.h>
#include <conio.h>
#include <string.h>

struct Persona{           //Definición de un registro (struct) en C
	char apellido[15];
	char nombre[15];
	int edad;
};

struct Persona vecP[10];  //Declaro un vector tipo Persona
int edad;

int main(){
	for (int i=0; i<10; i++){              //El índice para moverme por el arreglo
		printf("Persona nº: %d\n", i+1); //Se agrega un +1 porque arranca en 0
		printf("Ingrese apellido: ");
		scanf("%s", vecP[i].apellido);   //Recordar: es un string, no lleva ampersand
		printf("Ingrese nombre: ");
		scanf("%s", vecP[i].nombre);
		printf("Ingrese edad: ");
		scanf("%d", &vecP[i].edad);      //Es un entero, lleva ampersand
		_clrscr();
	}
	for (int i=0; i<10; i++){              //Recorro el arreglo y muestro los datos
		printf("Datos de persona nº: %d\n", i+1);
		printf("Apellido: %s\n", vecP[i].apellido);
		printf("Nombre:   %s\n", vecP[i].nombre);
		printf("Edad:     %d\n", vecP[i].edad);
		_getch();
		_clrscr();
	}
	return 0;
}
```

De este ejemplo se puede detallar el uso y no uso del ampersand, que está sujeto al tipo de dato que recibe el *scanf*. También podemos notar un incremento del índice en el primer *printf* dado que, como el arreglo comienza en la posición 0, sería extraño mostrar a la "Persona nº: 0". Esto se corrige con dicho incremento.

Finalmente, y como dato más relevante, podemos ver que el acceso a cada campo del registro dentro del vector es prácticamente idéntico a como se haría en Pascal o pseudocódigo; primero accediendo a la posición deseada mediante el índice y luego, con el punto, accediendo al campo específico que queremos operar.

# 8. Modularización

## 8.1. Definición

En C, los módulos se definen de la misma forma que el *main*, que no es más que un módulo reconocido por el compilador como el primero a ejecutar. Una forma estándar de definir módulos es mediante un prototipo antes del *main,* que consiste sólo en el encabezado del módulo con sus parámetros y la definición del cuerpo del módulo debajo de *main*.

Si bien la forma estándar con prototipo es la más recomendada por considerarse buena práctica, tener en cuenta que, de omitirse el prototipo, el módulo debe definirse *antes* del *main*, como se hace en Pascal y pseudocódigo.

Definición estándar de un módulo con prototipo:

```
#include <stdio.h>
#include <conio.h>

int moduloEjemplo1 (void);      //Prototipo del módulo moduloEjemplo sin parámetros

int main()                      //Programa principal main
{
      [bloque de código de main]
}

int moduloEjemplo(void)
{
      [bloque de código del módulo]   //La definición del módulo
}
```

Definición de un módulo sin prototipo, como se ve en pseudocódigo y Pascal:

```
#include <stdio.h>
#include <conio.h>

int moduloEjemplo(void)               //El módulo sin prototipo
{
      [bloque de código del módulo]   //La definición del módulo
}

int main()                      //Programa principal main
{
      [bloque de código de main]
}
```

La definición del módulo es igual a la del *main*, con su código encerrado entre llaves. El prototipo, sin embargo, no tiene código interno y por ende no lleva llaves, por lo que se debe cerrar con punto y coma la sentencia del mismo.

El *void* se explicará a continuación.

## 8.2. Parámetros

Los módulos pueden o no recibir parámetros. Hasta ahora los dos módulos que vimos (*main* y *moduloEjemplo*) no recibían ningún parámetro. Existen dos formas de aclarar que un módulo no recibe parámetros, pero sólo una de ambas es aceptada como estándar en los compiladores modernos.

Utilizando la palabra reservada *void* entre los paréntesis, podemos aclarar que ese módulo *NO* recibe ningún tipo de parámetro y tratar de enviarle uno causará error.

La otra opción es la que venimos empleando con *main,* que consiste en dejar vacíos los paréntesis. El problema con esta opción es que, en realidad, no está aclarando que el módulo *main()* no recibe parámetros, sino que lo que le está diciendo al compilador es que *no se sabe si va a recibir y en qué cantidad parámetros*.

La declaración de un *main* bajo esta opción, en la gran mayoría de los casos y en la totalidad de los vistos en este manual, no supondría problemas, dado que nunca realizamos invocaciones a *main* desde otro módulo. En su lugar, siempre corremos los ejecutables generados al compilar nuestro código, por lo que emplear *main()* y *main(void)* no tiene ninguna diferencia.

Sin embargo, sí podemos cometer errores si no usamos el *void* para los módulos que invocaremos desde *main*, por lo que se respetará el estándar y correcta definición de un módulo sin parámetros mediante el uso de *void*.

Para terminar, aclaramos que la versión con paréntesis vacíos puede servir si se trabaja con un compilador antiguo de C o a modo de compatibilidad con el mismo, el cual no tiene implementado *void* en la lista de parámetros.

En el caso de sí querer enviar parámetros, los mismos se declaran de la misma forma que declaramos variables:

```c
#include <stdio.h>
#include <conio.h>

int modulConParametro (int para1, int para2);  //Módulo que recibe dos parámetros enteros

int main()
{
	int x, y, z;
	...
	z=modulConparametro(x,y); //Invocación al módulo enviándole x e y como parámetros
	...
	return 0;
}

int modulConParametro (int x, int y)
{
	[bloque de código]
}
```

En el ejemplo anterior vemos como enviarle parámetros a un módulo, y también como invocarlo. Nótese la variable "*z*", esto es porque nuestro módulo de ejemplo era una función y necesitaba devolver un dato. En el siguiente Capítulo se verá cómo definir funciones y procedimientos.

A diferencia de Pascal, donde los parámetros se separan por punto y coma en la definición del módulo y por coma en la invocación, en C tanto en la invocación como en la definición los parámetros se paran por coma simple.

El equivalente a Pascal del anterior código sería:

```pascal
program ejemploModulo;
uses crt;

function mult(x: integer; y: integer): integer; //Podría definirse x e y juntos, pero lo
begin                                           //hacemos así para mostrar la separación
   [bloque de código]
end;
var
   x,y,z: integer;

begin            //Programa principal
   ...
   z:=mult(x,y); //Invoco a mult con x e y
   ...
end.
```

Un tema importante respecto a los parámetros de C: *sólo existe pasaje por copia*. A diferencia de Pascal, C sólo admite pasajes por copia. Los pasajes por referencia se realizan enviando como parámetro una

copia de la dirección de memoria del mismo, es decir, aplicando el ampersand y trabajando con un parámetro puntero en el bloque del módulo.

En realidad, esto es lo que hace Pascal con su palabra reservada *var*. En C debemos trabajar sin esta herramienta, haciendo explícito el envío de la dirección de memoria como pasaje.

Un pasaje por referencia entonces seria:

```
#include <stdio.h>
#include <conio.h>

void modulReferencia(int par1, int par2, int *resul);     //Módulo que recibe resul por
                                                          //referencia

int main()
{
      int x, y, z;
      ...
      modulReferencia(x, y, &z);              //z es enviado con &, por lo que en realidad
                                              //estamos enviando su dirección en memoria
      ...
      return 0;
}

void modulReferencia(int par1, int par2, int *resul) //El parámetro que enviamos era una
{                                                    //dirección, así que la recibimos como
      [bloque de código]                             //puntero
}
```

En donde la variable que se envía como parámetro lleva el ampersand para enviar su dirección en memoria en lugar de su valor copiado, mientras que el parámetro referenciado se declara con un asterisco, que es la forma en la que se emplea un puntero. Los punteros se verán en detalle en el Capítulo 9, pero basta con saber que para manejar un puntero se debe usar el asterisco.

## 8.3. Procedimientos y Funciones

Al igual que Pascal, C trabaja con procedimientos que pueden o no retornar información en un parámetro referenciado, y funciones que siempre retornan un tipo de dato nativo cuando son invocadas.

La forma de diferenciar una función de un procedimiento es bastante sencilla: antes de la declaración de un módulo, se especifica el tipo esperado de dato, como *int main(),* que aclara que va a retornar un entero. Si queremos definir un procedimiento, el mismo no debe tener un tipo esperado ya que no es la forma en la que trabaja, y esto se aclara mediante un *void*.

Entonces, una función se define:

```
int nombreFuncion(int par1, int par2);
```

Y un procedimiento:

```
void nombreProcedimiento(int par1, int par2, int *resul);
```

En el siguiente ejemplo, damos cierre a todo lo dado hasta ahora de modularización realizando un procedimiento y una función que reciban dos valores, los multipliquen y retornen el resultado:

```
#include <stdio.h>
#include <conio.h>

int mult (int x, int y);  //Función que devuelve la multiplicación de dos números enteros

void multReferencia (int x, int y, int *resul); //Procedimiento que hace lo mismo que mult pero
                                                //devuelve el resultado en un parámetro por referencia
int main()
{
      int x, y, z;

      printf("Ingrese dos numeros para multiplicarlos: ");
      scanf("%d", &x);
      scanf("%d", &y);
      printf("El producto de los numeros es: %d\n", mult(x, y));  //Invoco la función, el valor
      _getch();                                                   //devuelto es asignado al printf
      _clrscr();
      printf("Ahora haremos lo mismo pero por referencia");
      _getch();
      _clrscr();
      multReferencia(x, y, &z);   //Nótese como no enviamos la variable sino su dirección en memoria
      printf( "El producto de los numeros es: %d\n", z);  //La variable z por referencia tendrá el
      _getch();                                           //valor
      _clrscr();
      return 0;
}

int mult (int x, int y)
{
  return x * y;                 //Las funciones se asignan su valor con return
}

void multReferencia (int x, int y, int *resul){ //El parámetro que enviamos era una dirección, asique
                                                //la recibimos como puntero
      *resul=x*y;  //De nuevo, nótese como operamos con resul como un puntero, así modificamos por
}                  //referencia
```

## 8.4. Definición de *main* según estándares C99/C11

Este capítulo no es necesario para proseguir con el manual, se dedicará a abarcar la definición de un *main* según los estándares vigentes para la correcta programación en C.

Previamente, en el Capítulo 8.1, discutimos brevemente sobre cómo definir a *main* y concluimos que utilizaríamos *int main()*, dado que no presenta mayor problema. Esta definición no es incorrecta, pero, desde un punto de vista más estricto, no satisface a los estándares vigentes hoy en día, los cuales dan a *int main()* como una forma obsoleta de definir el mismo.

Tanto el anterior estándar C99 (ISO/IEC 9899:1999), como el actualmente vigente C11 (ISO/IEC 9899:2011), nos especifican cuatro formas correctas de definir un *main*:

1) int main (void) { body }
2) int main (int argc, char *argv[]) { body }
3) int main (int argc, char *argv[] , other_parameters ) { body }
4) Alguna otra forma definida según el entorno que ejecuta al programa

Expliquemos cada punto del estándar:

Punto 1) Es la definición que explicamos como alternativa en el Capítulo 8.1, el cual agrega un *void* para especificar la nulidad de parámetros.

Punto 2) Define el *main* con dos parámetros:

- *Argc*: Valor numérico no negativo, que representa el número de parámetros enviados al programa desde el ambiente de ejecución en donde estamos corriendo el mismo.
- **Argv*: Puntero a un arreglo de punteros de tipo Null-Terminated Multibyte Strings (NTMBS), que representan los parámetros pasados al programa desde el ambiente de ejecución (desde argv[0] hasta argv[argc-1]. El valor de argv[argc] será 0.

El *NTMBS* es una cadena de bytes distinta de cero que termina con un byte en cero, marcando el final.

Punto 3) Es igual al segundo, pero deja lugar a otros parámetros en caso que sea necesario

Punto 4) Este punto nos permite definir nuestro propio *main* basándonos en el ambiente de ejecución sobre el que estemos trabajando, definiendo por ejemplo un *void main*.

Si bien nosotros seguiremos trabajando en el manual con la forma *int main()*, es bueno que se esté informado respecto a lo que indican los actuales estándares.

## 8.5. Recursión

La recursión en C no difiere en nada con la vista tanto en pseudocódigo como en Pascal, basta con adaptarse al manejo de la sintaxis de C y su comportamiento. Lo único que necesitamos es entender el concepto de recursión y cómo aplicarlo.

Veremos el clásico ejemplo de factorial de *n* realizado en una función en C.

```
#include<stdio.h>
#include<conio.h>

int factorial(int x);     //Prototipo de la función recursiva.

//MAIN

int main(){
	int num;
	printf("Calculo de factorial mediante funcion recursiva\n");
	printf("Ingrese un numero: ");
	scanf("%d", &num);
	printf("El factorial de %d es: %d", num, factorial(num));
	_getch();    //Invoco a la función recursiva
	_clrscr();
	return 0;
}

//FUNCION RECURSIVA

int factorial(int x){     //Paso el número que quiero calcular por copia.
	if (x==0){          //Primero chequeo el caso base, en el factorial es x=0.
		return 1;    //El factorial de 0 es 1.
	}
	else{               //Si no llegue a 0, invoco recursivamente a factorial con x-1
		return x*factorial(x-1);  //IMPORTANTE: Esta es la llamada recursiva a la
	}                                 //función, es decir, una auto invocación.
}
```

No hay más que agregar respecto a recursión. La lógica no cambia de un lenguaje a otro, sólo la sintaxis para que el lenguaje pueda entender lo que se quiere lograr.

## 8.6. Arreglos y registros como parámetro

Ya vimos, en el Capítulo 8.1 de Parámetros, cómo se envían estos a un módulo y como los recibe y opera el mismo. Veremos ahora como trabajar con tipos estructurados de datos, especialmente los arreglos que tienen un manejo particular.

Aclaremos algo muy importante antes de comenzar: ***no existe pasaje por copia de un arreglo***. Los arreglos en C son punteros al primer elemento y, por ende, no hay forma de enviar un arreglo como parámetro sin modificar su contenido original. Sin embargo, podemos ingeniárnosla para simular una copia: cuando queramos enviar un arreglo por copia generaremos localmente otro arreglo del mismo tipo y tamaño y copiaremos uno a uno todos los valores. De esta forma tendremos una "copia" del arreglo para trabajar sin riesgo de modificar el original.

Entonces, si tenemos un arreglo del tipo:

```
int vecNum[5];  //Arreglo de 5 posiciones de tipo entero
```

Un pasaje por referencia sería simplemente:

```
void inicializarVecReferencia (int v[]){
	for (int i=0; i<5; i++){
		v[i]=0;		//Opero directamente sobre el arreglo pasado como parámetro
	}
}
```

Mientras que un pasaje por copia sería:

```
void inicializarVecCopia (int v[]){
	int vCopia[5];		//Genero una copia local del mismo formato
	for (int i=0; i<5; i++){
		vCopia[i]=v[i];		//Copio uno a uno los datos del arreglo original
	}
	for (int i=0; i<5; i++){
		vCopia[i]=1; //Opero sobre el arreglo copiado
	}
}
```

Podemos verificar la diferencia entre ambos procedimientos en el siguiente código:

```c
#include <stdio.h>
#include <conio.h>

void inicializarVecCopia (int v[]); //Procedimiento por copia

void inicializarVecReferencia (int v[]); //Procedimiento por referencia

int main(){
	int vecNum[5];  //Defino el arreglo
	inicializarVecReferencia(vecNum); //Invoco al procedimiento que modifica por referencia
	printf("Arreglo modificado por referencia");
	_getch();
	_clrscr();
	for (int i=0; i<5; i++){  //Muestro los valores del arreglo tras la inicialización
		printf("Valor del arreglo en posición %d: %d\n", i, vecNum[i]);
		_getch();
	}
	inicializarVecCopia(vecNum); //Invoco al procedimiento que modifica una copia
	_clrscr();
	printf("Arreglo modificado por copia");
	_getch();
	_clrscr();
	for (int i=0; i<5; i++){  //Vuelvo a mostrar los valores del arreglo
		printf("Valor del arreglo en posición %d: %d\n", i, vecNum[i]);
		_getch(); //Modifiqué el arreglo con valor 1, pero como era copia
	}		  //no se reflejó fuera del procedimiento
	_clrscr();
	return 0;
}

void inicializarVecCopia (int v[]){
	int vCopia[5];	//Genero una copia local del mismo formato
	for (int i=0; i<5; i++){
		vCopia[i]=v[i];	//Copio uno a uno los datos del arreglo original
	}
	for (int i=0; i<5; i++){
		vCopia[i]=1; //Opero sobre el arreglo copiado
	}
}

void inicializarVecReferencia (int v[]){
	for (int i=0; i<5; i++){
		v[i]=0;	//Opero directamente sobre el arreglo pasado como parámetro
	}
}
```

Si se ejecuta el código anterior, se verá que el valor del arreglo al inicializarse por referencia es 0, y al modificarse por copia sigue siendo 0, por lo que realizamos efectivamente un “pasaje por copia”.

Resaltemos entonces que, si se quiere enviar por copia un arreglo, se debe tener en cuenta el hecho de que un arreglo es un *puntero al primer elemento* y, por ello, sólo se envía por referencia, ya que enviamos una *copia de la dirección en memoria*.

Por otro lado, tenemos el pasaje de registros. A diferencia de los arreglos, los registros no tienen casos particulares que evaluar y es mucho más simple su pasaje por copia o referencia.

Teniendo un registro del tipo:

```
struct Persona {    //Definición de un registro Persona
       char apellido[15];
       char nombre[15];
       int dni;
};
```

Un pasaje por copia sería:

```
void modificarCopia(struct Persona p){
       p.dni=1; //Asigno valor a la copia
}
```

Mientras que un pasaje por referencia:

```
void modificarReferencia(struct Persona *p){ //p es un puntero al registro
       p->dni=0;  //Nótese cómo se accede al DNI en el puntero
}
```

El pasaje por referencia se realiza, al igual que con los pasajes de tipos nativos, enviando la dirección en memoria del registro y operándolo como un puntero, con la particularidad que debemos definir el registro.

Como se aclaró en el comentario del código, nótese que al campo *DNI* del puntero al registro no accedemos con un punto como hacemos normalmente con registros, ni tampoco con el asterisco como hacemos cuando se trata de punteros. Esto es así porque tenemos un símbolo reservado especialmente para los punteros a registro: la flecha “->”. Lógicamente, como una lista no es más que un conjunto de registros referenciados por puntero, este símbolo también se utilizará con ellas.

Al igual que con arreglo, verificaremos la diferencia entre pasaje por copia y por referencia con el siguiente ejemplo:

```c
#include <stdio.h>
#include <conio.h>

struct Persona{  //Defino el registro Persona
	char apellido[15];
	char nombre[15];
	int dni;
};

void modificarReferencia(struct Persona *p); //Módulo por referencia

void modificarCopia(struct Persona p); //Módulo por copia

int main(){
	struct Persona per;	//Defino la variable tipo Persona
	modificarReferencia(&per); //Invoco al módulo con per por referencia
	printf("DNI modificado por referencia: %d", per.dni);
	_getch();	//Verificamos que efectivamente el DNI vale 0
	_clrscr();
	modificarCopia(per);  //Invoco al módulo con per por copia
	printf("DNI modificado por copia: %d", per.dni);
	_getch();	//El DNI sigue valiendo 0, porque fue por copia
	_clrscr();
	return 0;
}

void modificarReferencia(struct Persona *p){
	p->dni=0;	//Como es un puntero de registro, usamos la flecha
}

void modificarCopia(struct Persona p){
	p.dni=1;  //Modificamos el parametro por copia
}
```

# 9. Punteros

## 9.1. Definición

A estas alturas ya hemos hecho un extenso uso de punteros en los módulos dada la forma en que se envían por referencia parámetros en C, por lo que su concepto y sintaxis deberían estar bastante familiarizados. Se explicará puntero nuevamente en caso que no haya quedado del todo claro.

Un puntero, como ya sabemos, es una variable cuyo contenido no es un dato, sino una dirección de memoria que apunta a un dato y con la cual puede modificarse el mismo. Los punteros en C se definen con el símbolo asterisco (*), y para asignarle la dirección de una variable a un puntero debemos usar el ampersand para obtener dicha dirección.

El valor de dirección nulo en C no es *nil*, como sucede en Pascal. En cambio, en C se usa el valor numérico 0 (cero) para definir que un puntero no apunta a ninguna dirección o, lo que es lo mismo, que tiene valor nulo.

Una variable de tipo puntero se define:

```
int *punteroNumero; //Un puntero a un entero, con el asterisco en el nombre de variable
```

A continuación, un ejemplo de cómo trabajar con un puntero:

```
#include<stdio.h>
#include<conio.h>

int num, *punteroNumero;

int main(){
	printf("Ejemplo de uso de punteros\n");
	printf("Ingrese un numero: ");
	scanf("%d", &num);		//Lectura normal de un valor
	_clrscr();
	puntero=&num;			//Asigno a la variable puntero la dirección de num. Nótese el
					//uso del ampersand
	printf("Ingreso: %d\nLa variable puntero contiene: %d", num, *puntero); //Muestro el contenido
	_getch();                                                                //de num y del puntero
	_clrscr();                                      //El puntero se accede con asterisco (*)
	*puntero=10;			//Modificamos la información de num a través del puntero a su
					//dirección de memoria
	printf("Ahora la variable 'num' tiene valor: %d", num); //Verificamos que efectivamente se
	_getch();                                               //modificó el valor
	_clrscr();
	return 0;
}
```

El puntero en sí mismo no tiene mayor complejidad que la vista en el ejemplo. Tener en cuenta siempre los usos de asterisco y ampersand.

Muchas veces necesitaremos liberar la memoria direccionada por nuestro puntero. Para ello existe la función nativa "*free()*":

```
free(puntero);      //Libera la memoria ocupada en la dirección del puntero
```

## 9.2. Puntero como parámetro

Los punteros, al enviarse como parámetros, tienen un comportamiento muy particular que debemos explicar en profundidad de detalles.

Digamos que enviamos un puntero a una variable entera por parámetro y modificamos el valor apuntado, es decir, usamos asterisco.

```
#include <stdio.h>
#include <conio.h>

void modificarPuntero(int *p);

int main(){
	int x=5, *punteroNum;	//Declaro un entero y un puntero a entero
	punteroNum=&x;		//Apunto el puntero a la variable entera.
	printf("Valor del puntero: %d\n", *punteroNum); //El puntero muestra 5
	_getch();
	modificarPuntero(punteroNum);	//Modifico el valor de x a través del puntero
	printf("Valor del puntero modificado: %d", *punteroNum);
	_getch();
	return 0;
}

void modificarPuntero(int *p){
	*p=0;
}
```

El valor será modificado dado que sería como enviar un parámetro por referencia.

Digamos que enviamos el mismo puntero por parámetro, pero ahora modificamos sin el asterisco ya que queremos modificar la dirección a la que apunta ese puntero.

```c
#include <stdio.h>
#include <conio.h>

void modificarPuntero(int *p);

int main(){
	int x=5, *punteroNum;	//Declaro un entero y un puntero a entero
	punteroNum=&x;			//Apunto el puntero a la variable entera.
	printf("Valor del puntero: %d\n", *punteroNum); //El puntero muestra 5
	_getch();
	modificarPuntero(punteroNum);	//Modifico la dirección del puntero
	printf("Valor del puntero modificado: %d", *punteroNum);
	_getch();			//El valor mostrado sigue siendo 5, no cambió
	return 0;
}

void modificarPuntero(int *p){
	p=0;	//No uso asterisco para modificar la dirección del puntero
}
```

Este último caso no funcionará, dado que C, como ya aclaramos, *no permite pases por referencia, sólo por copia*. Es decir, si enviamos un puntero como parámetro, el dato al que apunta podrá modificarse, pero el valor que contiene el puntero es un valor duplicado del original y cualquier cambio realizado sobre él *no se reflejará fuera del módulo*.

Para el caso de un puntero en particular podría no darse mucha importancia al hecho de no poder modificar hacia dónde apunta desde un módulo. Pero recordemos que una lista *también es un puntero*, y si quisiéramos modificar una lista mediante un módulo enviando su cabecera o raíz, por lo planteado anteriormente, *no sería posible*.

Entonces, ¿es imposible modificar un puntero o lista en un módulo en C? No, existen formas de poder modificar la dirección apuntada por el puntero.

Una de ellas consiste en definir el puntero como variable global, con lo cual no tendríamos que enviarlo como parámetro y podríamos operarlo desde cualquier módulo en nuestro ejecutable. Esta forma resulta bastante sencilla, pero tiene como contraparte que el uso de variables globales puede facilitar la confusión y la aparición de errores al codificar.

El ejemplo anterior podría resolverse entonces con la siguiente redefinición del puntero:

```c
#include <stdio.h>
#include <conio.h>

void modificarPuntero(void); //No envío parámetros

int *punteroNum;          //Declaro un puntero global

int main(){
	int x=5;              //Declaro un entero
	punteroNum=&x;        //Apunto el puntero a la variable entera.
	printf("Valor del puntero: %d\n", *punteroNum); //El puntero muestra 5
	_getch();
	modificarPuntero();//Modifico la dirección del puntero
	printf("Valor del puntero modificado: %d", *punteroNum);
	_getch();             //El programa falla porque el puntero ahora vale nil
	return 0;
}

void modificarPuntero(void){
	punteroNum=0;         //No uso asterisco para modificar la dirección del puntero
}                         //Además, uso el nombre global
```

Si ejecutamos este código, notaremos que el programa falla en la ejecución. Es lógico, estamos queriendo mostrar un valor apuntado por *p*, siendo que *p* vale *nil* (0) tras la ejecución del módulo. Lo importante en este ejemplo es que antes *p* apuntaba a *x*, y luego paso a apuntar a *nil*, lo que quiere decir que efectivamente logramos modificarlo.

Otra forma de modificar un puntero es enviar como parámetro un *puntero al puntero*, es decir, enviar la dirección en memoria de nuestro puntero, y recibirla como un puntero que apunta al puntero definido en *main*. Este juego de palabras puede ser confuso, pero se entenderá mejor con el siguiente ejemplo:

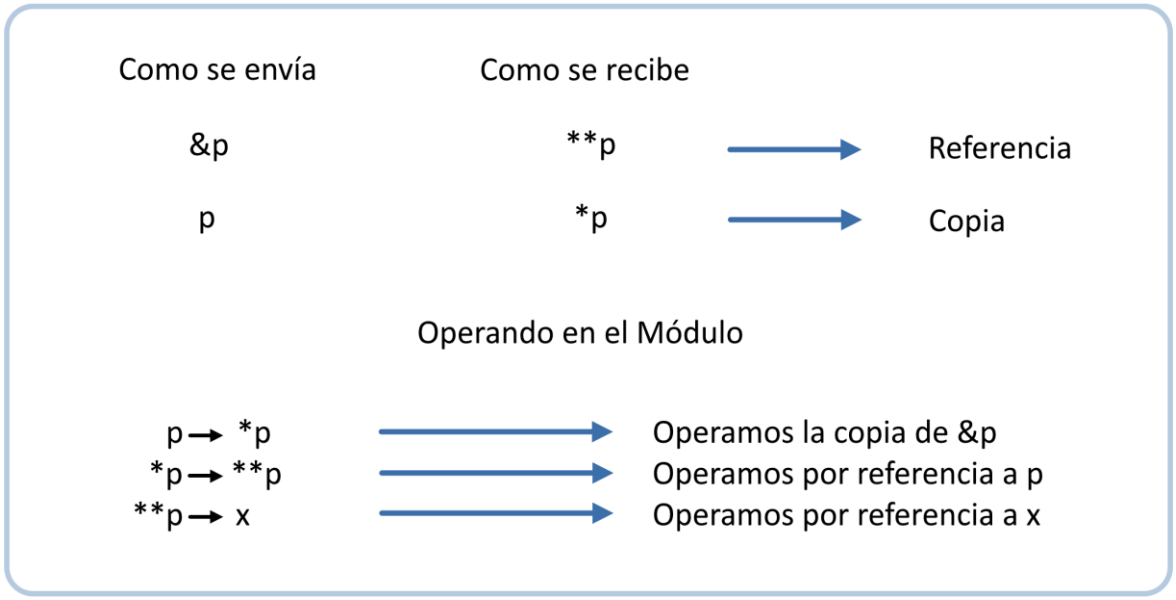
```c
#include <stdio.h>
#include <conio.h>

void modificarPuntero(int **p); //Doble asterisco, quiere decir que es un puntero
                                //que apunta a otro puntero
int main(){
	int x=5, *punteroNum;     //Declaro un entero y un puntero a entero
	punteroNum=&x;            //Apunto el puntero a la variable entera.
	printf("Valor del puntero: %d\n", *punteroNum); //El puntero muestra 5
	_getch();
	modificarPuntero(&punteroNum);  //Envio la dirección del puntero como parámetro
	printf("Valor del puntero modificado: %d", *punteroNum);
	_getch();    //El programa falla porque el puntero ahora vale nil
	return 0;
}

void modificarPuntero(int **p){
	*p=0;  //Hago nil el puntero que envié. Si usara **p modificaría a x
}
```

¿Qué hicimos en el ejemplo anterior? Definimos el parámetro con *doble asterisco*, dado que queremos enviar la dirección de memoria de un puntero, que es *otra dirección de memoria*. Asimismo, cuando invocamos el módulo le pasamos como parámetro la dirección de memoria de nuestro puntero con el ampersand.

Finalmente, en el módulo, operamos el puntero con *un solo asterisco*. Si utilizamos un solo asterisco, estamos operando sobre la dirección del puntero que enviamos por parámetro. Si en cambio utilizamos dos asteriscos, estaremos modificando la variable *x* a la que apunta.



Si se lo piensa detenidamente en analogía con Pascal, el “puntero de puntero” es justamente lo que hacemos cuando enviamos una lista por parámetro con la palabra *var*, ya que *var* indica que es un pasaje por referencia o, lo que es lo mismo, un pasaje de la dirección en memoria. Pero si lo que estamos enviamos es un puntero, estamos enviando una dirección de memoria como parámetro, y *var* enviará entonces la dirección de *esa dirección*, ósea, un puntero a la dirección, ósea, un puntero de puntero.

Se usará en el manual esta segunda forma (puntero de puntero) para trabajar dado que es más eficiente y refleja efectivamente un “pasaje por referencia” del puntero, tal cual se ve en Algoritmos y Estructuras de Datos.

# 10. Listas

## 10.1. Definición y uso

La definición de una lista en C varía ligeramente a la que conocemos en pseudocódigo y Pascal. Ya no definimos un tipo lista que apunte a un nodo y el nodo como registro, sino que definimos directamente nuestro registro cuerpo de la lista y el enlace se realiza declarando una variable tipo puntero, que será nuestra lista o nodo.

En Pascal definimos una lista y las variables para un uso básico de la forma:

```
program ejemploLista;
uses crt;

type
   //Defino el registro persona
   persona=record
           nombre: string;
           apellido: string;
           anioNacimiento: integer;
           end;
   //Defino la lista y su nodo
   lista=^nodo;
   nodo=record
           dato: persona;
           ps: lista;
        end;

var
   l:lista;   //Variable tipo lista
   p:persona; //Dato para cargar
```

En C, la misma lista se define:

```
struct Persona{                 //Defino el registro Persona
       char nombre[15];
       char apellido[15];
       int anioNacimiento;
};

struct Lista{                   //Defino la lista (nótese que es en el mismo nodo)
       struct Persona dato;     //Dato de la lista
       struct Lista *ps;        //Puntero siguiente, esto es lo que nos permite operar
};                              //el registro como lista

struct Persona per;             //Declaro variable del dato
struct Lista *lis=0;            //Declaro variable de la lista y la hago nil
```

Antes de ver una aplicación completa en donde carguemos y mostremos una lista, debemos detallar cómo se inicia la misma. En Pascal, para iniciar un nuevo nodo basta con hacer uso de la función *new(nuevoNodo)* el cual ya prepara el puntero al nodo para cargarlo e insertarlo. En C, esta operación es mucho más compleja, ya que debemos desglosar el *new* en toda la operación que realiza.

Entonces, lo que en Pascal es:

```
var nuevoNodo: lista;     //Declaro un nuevo nodo de la lista
 ...
 ...
new(nuevoNodo);           //Genero el nuevo nodo
```

En C es:

```
#include <stdlib.h>
 ...
 ...
struct Lista *nuevoNodo = malloc(sizeof(struct Lista)); //Declaro y genero el nuevo nodo
```

Repasemos este ejemplo. Antes que nada, vemos que hemos incluido una nueva librería a nuestro código, *<stdlib.h>*, la cual, como ya explicamos al principio en el Capítulo 1.3, agrega funciones para operar sobre la memoria dinámica y procesos. Una función incluida en esta librería es *malloc()* (***m**emory **alloc**ation*), la cual se encarga de reservar una cantidad de bits en memoria generando un bloque al que apuntará nuestro nodo o lista.

Otra función que vemos en el código es *sizeof()*. Esta función retorna el tamaño en bits del parámetro que se le envía, en este caso un registro.

Entonces, ¿qué está haciendo C con la línea de código anterior? Aunque parezca algo engorroso, notarán que si lo analizamos es en realidad bastante sencillo. Leído de derecha a izquierda, primero obtenemos el tamaño en bits de nuestro registro que es la lista, luego reservamos un bloque de memoria con ese tamaño exacto y, por último, le indicamos a nuestro puntero que apunte a dicho bloque reservado por *malloc*. El resultado final será el mismo que con el new de Pascal, pero queda en evidencia todo lo que ocurre detrás de dicha función. Como ya vimos con el *var* de los parámetros, lo que en Pascal se hace de manera sencilla con una función o palabra clave, nos lleva algo más de trabajo en C dado el bajo nivel sobre el que opera.

Para asignarle valor a un nodo de la lista, trabajamos de la misma manera que los registros cuando se envían por referencia: se utiliza la flecha "->" para acceder al campo. Entonces, si quisiera crear y cargar un nuevo nodo de la lista anterior, lo haría de la forma siguiente:

```
struct Lista *nuevo = malloc(sizeof(struct Lista)); //Defino el nuevo nodo
nuevo->dato=p;
nuevo->ps=0;
```

Para el ejemplo anterior suponemos que *p* ya estaba cargado con datos y listo para ser agregado al campo *dato* de la lista.

También se podría cargar datos directamente desde el usuario en la lista, si así lo quisiera:

```
printf("Ingrese nombre: ");
scanf("%s", nuevo->dato.nombre); //Accedo al string nombre, en dato, apuntado por nuevo
_clrscr();
printf("Ingrese apellido: ");    //Idem con el string de apellido
scanf("%s", nuevo->dato.apellido);
_clrscr();
printf("Ingrese anio de nacimiento: ");
scanf("%d", &nuevo->dato.anioNacimiento); //Como es un tipo nativo y no un arreglo, debo
_clrscr();                                //usar el ampersand (como siempre hacemos)
```

Como se aclara en los comentarios, recordar siempre el uso del ampersand cuando los campos a cargar no sean arreglos o punteros.

## 10.2. Lista como parámetro

Todo lo que vimos respecto a pasaje como parámetro de punteros en el Capítulo anterior se aplica de la misma forma a las listas, lo cual resulta lógico ya que ,en definitiva, una lista puede reducirse a una cadena enlazada de punteros.

Existen tres maneras de trabajar con listas en módulos:

- Definir la lista como variable global.
- Pasar un puntero a la lista como parámetro (puntero de puntero).
- Modificar la lista en una función y devolverla con un *return*.

Al igual que con punteros, utilizaremos la segunda opción dado que es así como se enseña el manejo de listas normalmente y porque considero es la manera más eficiente de trabajar con las mismas.

Un módulo clásico de inicialización de lista por referencia sería entonces:

```
void inicializar(struct Lista **l){    //Envío una lista de tipo “Lista” por referencia
      *l=0;                            //Hago nil la lista
}
```

Donde enviamos un puntero a nuestra lista mediante el doble asterisco y modificamos hacia dónde apunta la misma mediante un asterisco simple. De usar un doble asterisco para la modificación, estaríamos operando sobre la copia local del puntero a la lista.

Como vemos, es prácticamente idéntico a cuando trabajamos con punteros. Veamos ahora un módulo de inserción al final de la lista, considerando que ya recibimos un registro dato de tipo *Persona* cargado con información.

```
void insertar(struct Lista **l, struct Persona p){ //Envío por referencia la lista y por
                                                   //copia un registro
      struct Lista *aux; //Defino el aux para moverme por l
      struct Lista *nuevo = malloc(sizeof(struct Lista)); //Defino el nuevo nodo
      nuevo->dato=p;      //Asigno el dato al nuevo nodo, en este caso una Persona
      nuevo->ps=0;        //El puntero siguiente es nil
      if (*l==0){         //Si la lista está vacía, inserto al principio
            *l=nuevo;
      }
      else{               //Sino, me muevo hasta el final y lo inserto
            aux=*l;       //Asigno un aux para no modificar a la cabeza l
            while(aux->ps!=0){ //Recorro aux hasta el final de la lista
                  aux=aux->ps;
            }
            aux->ps=nuevo;      //Realizo la inserción
      }
}
```

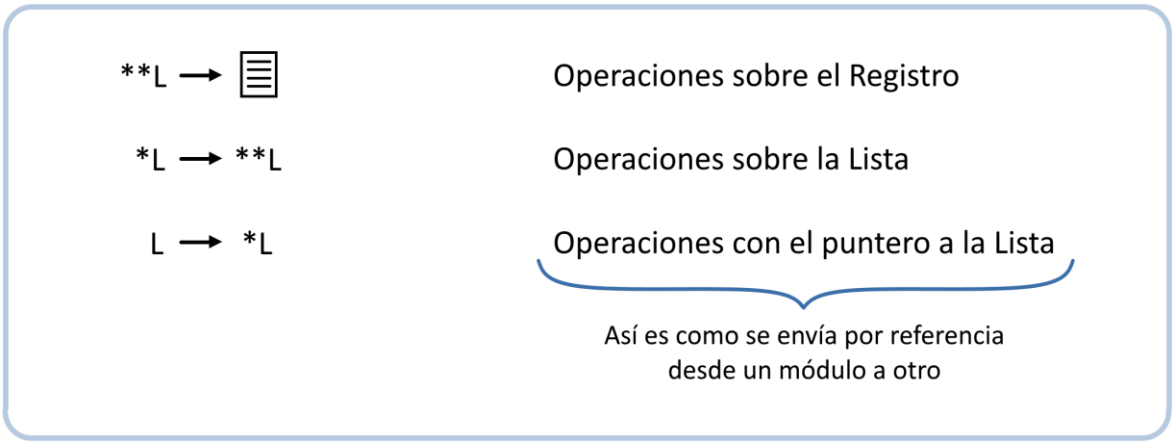
Una aclaración importante: supongamos que tenemos un procedimiento A al cual le pasamos por referencia nuestra lista, y queremos pasar dicha lista por referencia a un procedimiento interno B. Se hará de la siguiente forma:

```
void procedimientoA (struct Lista **l);      //Este es nuestro procedimiento A.
void procedimientoB (struct Lista **l);      //Este es nuestro procedimiento B.

int main(){
      struct Lista *L;          //Definimos nuestra lista.
      procedimientoA (&L);      //Envío la dirección de memoria de L.
}

void procedimientoA (struct Lista **l){      //Lista por referencia, prestar
                                             //atención al doble *.
*l= 0; //Realizo una operación con la lista, para ello necesito usar
      //un *, ya que doble asterisco significa operar sobre el valor
      //final del puntero.
procedimientoB(l); //IMPORTANTE: Nótese que pasamos la lista sin
                   //asterisco, esto es porque, al no usar asteriscos,
                   //trabajamos con el puntero del puntero de la
                   //lista, algo necesario para poder pasarla por
                   //referencia en un sub procedimiento interno.
}
```

Nótese el uso de los asteriscos. Cada asterisco accede un nivel más al valor final de la variable. En el caso del parámetro, usar ***l* accedería al registro de la lista, **l* accede al puntero al primer registro, lo que nos permite operar dinámicamente, y finalmente *l* accede al puntero que, a su vez, accede al puntero anterior.




De esta forma, se puede enviar por referencia a nuestra lista ya referenciada en nuestro procedimiento actual.

## 10.3. Aplicación

Con todo lo visto de lista hasta ahora y de C en general, podemos hacer un cierre de contenido con una aplicación que nos permita realizar ABM y consulta de una lista determinada. Para ello, tendremos distintos módulos que nos permitan operar sobre la lista y un módulo principal *main* con un menú de opciones que nos permita elegir qué tareas ejecutar. Planteamos para esta aplicación el siguiente enunciado:

> *Realizar una aplicación que permita agregar, modificar y eliminar productos de una lista. Un producto está compuesto por código, nombre y stock del mismo. La lista deberá estar ordenada por el código de producto. Además, se deberá poder mostrar todos los elementos de la lista, consultar por uno específico mediante su código, y obtener el total de productos cargados. El módulo principal deberá contar con un menú de opciones para navegar por la aplicación.*

Pasemos a definir uno por uno los módulos del programa y el principal.

- Cargar producto

```
void cargarProducto(struct Producto *p, int codigo){
	if (codigo == -1){		//Es la creación de un producto
		printf("Ingrese codigo: ");
		scanf("%d", &p->codigo);
		_clrscr();
	}
	else{				//Es la modificación de un producto ya
					//existente. No modifico código
		p->codigo= codigo;
	}
	printf("Ingrese nombre: ");
	scanf("%s", p->nombre); //String, sin ampersand
	_clrscr();
	printf("Ingrese stock: ");
	scanf("%d", &p->stock);
	_clrscr();
}
```

- Inicializar lista

```
void inicializarLista (struct Lista **l){
	*l=0;  //Lista en NIL
}
```

- Insertar ordenado

```
void insertarOrdenado(struct Lista **l, struct Producto p){
	struct Lista *nuevo=malloc(sizeof(struct Lista));   //Defino el nuevo nodo
	struct Lista *ant;  //Defino el nodo anterior
	struct Lista *act;  //Defino el nodo actual
	nuevo->dato=p;      //Cargo producto en nuevo
	nuevo->ps=0;        //Inicializo ps de nuevo en NIL
	if (*l==0){         //La lista está vacía, no hace falta moverme
		*l=nuevo;
	}
	else{               //La lista no está vacía, busco el punto de inserción
		ant=0;      //Puntero al nodo anterior
		act=*l;     //Puntero al nodo actual
		while ((act!=0) && (act->dato.codigo < nuevo->dato.codigo)){
			ant=act;
			act=act->ps;
		}
		if (ant==0){ //Si ant es NIL, inserto al principio
			nuevo->ps= act;
			*l=nuevo;
		}
		else{               //Debo insertar en un punto medio o al final
			nuevo->ps=act;
			ant->ps=nuevo;
		}
	}
}
```

- Modificar producto

```
void modificarProducto(struct Lista **l){
	struct Lista *act;		//Defino mi auxiliar
	struct Producto p;		//Defino el producto a modificar
	int codigo;			//Código con el cual buscar el producto
	act=*l;
	if (*l!=0){			//La lista no está vacía
		printf("Ingrese codigo de producto a modificar: ");
		scanf("%d", &codigo);
		_clrscr();
		//Busco el producto, si existe
		while((act != 0) && (act->dato.codigo != codigo)){
			act=act->ps;
		}
		if (act == 0){		//El producto del código dado no existe
			printf("El producto de codigo %d no existe", codigo);
		}
		else{			//Encontró el producto, realiza modificación
			printf("Ingresando nuevos datos del producto\n");
			cargarProducto(&p, codigo);
			act->dato= p;
			_clrscr();
			printf("Modificacion realizada con exito");
		}
	}
	else{
		printf("La lista esta vacia");
	}
	_getch();
	_clrscr();
}
```

- Eliminar producto

```
void eliminarProducto(struct Lista **l){
	struct Lista *act;
	struct Lista *ant;
	int cod;
	if(*l!=0){				//La lista no está vacía
		ant=0;
		act=*l;
		printf("Ingrese codigo del producto a eliminar: ");
		scanf("%d", &cod);
		_clrscr();
		while((act!=0) && (act->dato.codigo!=cod)){
			ant=act;
			act=act->ps;
		}
		if (act==0){		//El elemento no existe
			printf("El producto de codigo %d no existe", cod);
		}
		else if(ant==0){	//Es el primer elemento
			*l=act->ps;	//Muevo la cabeza de la lista
			free(act);	//Elimino el nodo actual
			printf("Eliminacion realizada con exito");
		}
		else{				//Es un elemento intermedio o final
			ant->ps=act->ps;	//Reacomodo el ps de anterior
			free(act);		//Elimino el nodo actual
			printf("Eliminacion realizada con exito");
		}
	}
	else{
		printf("La lista esta vacia");
	}
	_getch();
	_clrscr();
}
```

- Listar productos

```
void listarProductos(struct Lista *l){
	int i=1;
	if(l==0){
		printf("La lista esta vacia");
		_getch();
	}
	else{		//Si la lista no está vacía, recorro cada producto
		while(l!=0){
			printf("Producto numero: %d\n", i);
			printf("Codigo:    %d\n", l->dato.codigo);
			printf("Nombre:    %s\n", l->dato.nombre);
			printf("Stock:     %d\n", l->dato.stock);
			printf("--------------------\n");
			_getch();
			l=l->ps;
			i++;
		}
	}
	_clrscr();
}
```

- Buscar producto

```
void buscarProducto(struct Lista *l){
	int cod;
	if(l==0){
		printf("La lista esta vacia");
		_getch();
	}
	else{
		printf("Ingrese codigo del producto: ");
		scanf("%d", &cod);
		_clrscr();
		while((l!=0) && (l->dato.codigo != cod)){
			l=l->ps;
		}
		if(l==0){	//Si es NIL, no encontró un producto de código cod
			printf("El producto de codigo %d no existe", cod);
		}
		else{		//Encontró el producto, lista sus datos
			printf("Datos del producto:\n");
			printf("--------------------\n");
			printf("Codigo:    %d\n", l->dato.codigo);
			printf("Nombre:    %s\n", l->dato.nombre);
			printf("Stock:     %d\n", l->dato.stock);
		}
		_getch();
	}
	_clrscr();
}
```

- Total de productos

```
int totalProductos(struct Lista *l){
	int contador=0;			//Inicializo contador en 0
	if (l!=0){
		while (l!=0){
			l=l->ps;
			contador++;	//Incremento por cada elemento nuevo que
					//encuentra
		}
	}
	return contador;	//Retorno el total de productos de la lista (0 si
				//no hay productos)
}
```

- Librerías, tipos estructurados y módulos prototipos

```
#include <stdio.h>
#include <conio.h>
#include <string.h>
#include <stdlib.h>

struct Producto{			//Defino Producto
	int codigo;
	char nombre[15];
	int stock;
};

struct Lista{				//Defino la lista
	struct Producto dato;
	struct Lista *ps;
};

//----Defino modulos prototipados
void cargarProducto(struct Producto *p, int codigo);
void inicializarLista (struct Lista **l);
void insertarOrdenado(struct Lista **l, struct Producto p);
void modificarProducto(struct Lista **l);
void eliminarProducto (struct Lista **l);
void listarProductos (struct Lista *l);
void buscarProducto (struct Lista *l);
int totalProductos (struct Lista *l);
```

- Módulo principal – parte 1

```
int main(){
	int r, c;
	struct Lista *l;		//Declaro la lista
	struct Producto p;		//Declaro el producto que agrego a la lista
	inicializarLista(&l);		//Inicializo la lista
	r=1;
	while (r!=0){
		printf("MENU PRINCIPAL\n");
		printf("--------------------\n");
		printf("1) Agregar producto\n");
		printf("2) Modificar producto\n");
		printf("3) Eliminar producto\n");
		printf("4) Listar productos\n");
		printf("5) Buscar producto\n");
		printf("6) Total de productos cargados\n\n");
		printf("0) Salir\n");
		printf("Ingrese una opcion: ");
		scanf("%d", &r);
		_clrscr();
```

- Módulo principal – parte 2

```
            switch(r){
                  case 1:
                        c= -1;                                    //Control
para saber si el producto existe o no
                        cargarProducto(&p,c);           //Cargo el producto
                        insertarOrdenado(&l, p);  //Lo inserto ordenado
                        break;
                  case 2:
                        modificarProducto(&l);
                        break;
                  case 3:
                        eliminarProducto(&l);
                        break;
                  case 4:
                        listarProductos(l);
                        break;
                  case 5:
                        buscarProducto(l);
                        break;
                  case 6:
                        printf("El total de productos cargados es: %d",
totalProductos(l));
                        _getch();
                        break;
                  case 0:
                        printf("Terminando ejecucion");
                        _getch();
                        break;
                  default:
                        printf("Opcion incorrecta ingresada");
                        _getch();
                        break;
            }
            _clrscr();
      }
      return 0;
}
```

Esta aplicación nos sirve tanto para cierre de listas como para cierre general de todo lo visto hasta ahora en C.