

**Universidad de Buenos Aires**  
**Facultad de Ingeniería**

## **75.41 – Algoritmos y Programación II**

*Cátedra Ing. Patricia Calvo*

---

# Memoria y Punteros

---



## Índice

Índice .....	2
1 Memoria .....	3
1.1 División .....	3
1.2 Codificación .....	3
1.3 Sistemas de Numeración .....	4
1.4 Reglas de Conversión entre Sistemas de Numeración .....	5
1.5 Almacenamiento .....	7
1.6 Direccionamiento.....	8
2 Punteros .....	9
2.1 Concepto.....	9
2.2 Usos y ventajas de los punteros.....	9
2.3 Declaración.....	9
2.4 Diagramas.....	10
2.5 Operaciones sobre Punteros.....	10
2.6 Referenciación y Desreferenciación.....	12
2.7 Casteo .....	13
2.8 Asignación Dinámica de Memoria.....	14
2.9 Dirección NULA .....	15
2.10 Memoria Colgada .....	15
2.11 Referencia Perdida.....	16
2.12 Punteros a función .....	16
2.12.1 Definir un Tipo de Puntero a Función en C++ .....	17
2.12.2 Asignar la Dirección de la Función: .....	18
2.12.3 Invocar la función a travez del Puntero:.....	18
2.12.4 Ejemplo.....	18
3 Ejemplos .....	19
3.1 Ejemplo de Casteo: Cómo es un Integer?.....	19
3.2 Ejemplo de Punteros: Swap.....	21
3.3 Análisis de los distintos Swap .....	23
3.3.1 Invocación a Swap1 .....	23
3.3.2 Invocación a Swap2.....	24
3.3.3 Invocación a Swap3.....	24
3.3.4 Invocación a Swap4.....	25
3.3.5 Comentarios:.....	25
3.4 Ejemplo de Casteo: Que hace? .....	27



# 1 Memoria

## 1.1 División

A la memoria del ordenador la podemos considerar dividida en 4 segmentos lógicos:

<b>CD</b>	: Code Segment
<b>DS</b>	: Data Segment
<b>SS</b>	: Stack Segment
<b>ES</b>	: Extra Segment (ó Heap)

El **Code Segment** es el segmento donde se localizará el código resultante de compilar nuestra aplicación, es decir, la algoritmia en Código Máquina.

El **Data Segment** almacenará el contenido de las variables definidas en el modulo principal (variables globales)

El **Stack Segment** almacenará el contenido de las variables locales en cada invocación de las funciones y/o procedimientos (incluyendo las del main en el caso de C/C++).

El **Extra Segment** está reservado para peticiones dinámicas de memoria.

## 1.2 Codificación

La información se almacena en la memoria en forma electromagnética.

La menor porción de información se denomina **bit** y permite representar **2 símbolos** (0 o 1, apagado o prendido, no o si, etc).

Se denomina **Byte** a la combinación de 8 bits y permite representar **256 símbolos** posibles.

Se denomina **Word** a la combinación de bytes que maneja el procesador (en procesadores de 16 bits serán 2 Bytes, en procesadores de 32bits serán 4 bytes, etc.)

Para definir capacidades de memoria, se suelen utilizar múltiplos del Byte como son:

8 bit	→	1 Byte
1024 Bytes	→	1 Kilo Byte (KB)
1024 KBytes	→	1 Mega Byte (MB)
1024 MBytes	→	1 Giga Byte (GB)
1024 GBytes	→	1 Tera Byte (TB)



## 1.3 Sistemas de Numeración

Matemáticamente existe el concepto de Sistemas de Numeración, de los cuales el Sistema Decimal es el que utilizamos habitualmente y cuyos 10 símbolos bien conocemos:

0, 1, 2, 3, 4, 5, 6, 7, 8, 9

Ahora bien, dijimos que la memoria es **bi-estable**, es decir, sólo maneja 2 símbolos, y para representar su contenido se suele utilizar el **Sistema Binario** de Numeración, cuyos símbolos son:

0, 1

De esta forma, la información contenida en 1 Byte se representaría como la combinación de 8 de estos símbolos (se los suele suceder con la letra “b” para indicar que esta en binario):

01011000b

Tener en cuenta que los bits de un Byte se numeran desde 0 de derecha a izquierda. Y a esta numeración se la suele llamar peso del bit. Así, el bit 1 vale 0 en el ejemplo anterior.

Como pueden notar, esto es poco práctico a la hora de manejar muchos bytes, con lo que se suele utilizar otro Sistema a tales efectos. Al mismo se lo denomina **Sistema Hexadecimal** y comprende 16 símbolos; a saber:

0, 1, 2, 3, 4, 5, 6, 7, 8, 9, A, B, C, D, E, F

Su utilidad radica en que es fácilmente decodificable en **bits** ya que un número hexadecimal de 2 dígitos corresponde exactamente a un número binario de 8 dígitos.



## 1.4 Reglas de Conversión entre Sistemas de Numeración

Una regla práctica de conversión entre los símbolos esta dada por la siguiente tabla:

Hexadecimal	Binario	Decimal
0	0000	0
1	0001	1
2	0010	2
3	0011	3
4	0100	4
5	0101	5
6	0110	6
7	0111	7
8	1000	8
9	1001	9
A	1010	10
B	1011	11
C	1100	12
D	1101	13
E	1110	14
F	1111	15

**Binario → Hexadecimal** : conformar grupos de 4 dígitos binarios y cambiarlos por sus correspondientes dígitos hexadecimales (de requerir, agregar ceros a la izquierda del número inicial).

1011000b → 0101 1000 → 5 8 → 58h

**Hexadecimal → Binario** : cambiar cada dígito hexadecimal por sus 4 correspondientes binarios.

58h → 5 8 → 0101 1000 → 1011000b

**Binario → Decimal** : multiplicar cada bit por (2 elevado al peso del bit) y sumar los resultados

Dígitos	1	0	1	1	0	0	0
Peso	6	5	4	3	2	1	0
$2^{\text{peso}}$	64	32	16	8	4	2	1
Dígito x $2^{\text{peso}}$	64	0	16	8	0	0	0
Suma	88						



Decimal  $\rightarrow$  Binario : se divide sucesivamente el número decimal por 2 hasta tanto se obtenga como resultado un 0 o 1. Luego se compone el número binario con el cociente final seguido de la secuencia de restos hasta el resto inicial.

Dividendo	Divisor	Cociente	Resto
88	2	44	0
44	2	22	0
22	2	11	0
11	2	5	1
5	2	2	1
2	2	1	0

1011000

Hexadecimal  $\rightarrow$  Decimal : multiplicar cada dígito por (16 elevado al peso del bit) y sumar los resultados

Dígitos	5	8
Peso	1	0
$2^{\text{peso}}$	16	1
Dígito $\times 2^{\text{peso}}$	80	8
Suma	88	

Decimal  $\rightarrow$  Hexadecimal : se divide sucesivamente el número decimal por 16 hasta tanto se obtenga como resultado un número entre 0 y 15. Luego se compone el número hexadecimal con el cociente final seguido de la secuencia de restos hasta el resto inicial. Representar cada uno de estos valores con su correspondiente símbolo hexadecimal.

Dividendo	Divisor	Cociente	Resto
88	16	5	8

58

(Nota: 8 es el dígito hexadecimal que le corresponde al 8 decimal)



## 1.5 Almacenamiento

Antes de explicar el tema en cuestión, es necesario explicar que cuando se almacena un número en memoria compuesto por más de un byte, se denomina byte más significativo al que guarda la porción de mayor orden de magnitud.

El número 516 equivale al binario 1000000100. Por ende son 2 bytes → el 00000010 y el byte 00000100. El primer byte es al que denominamos más significativo por cuanto sus dígitos representan el mayor orden de magnitud ( representa 512 ) en tanto que el otro se denomina menos significativo porque representa la fracción de menor orden de magnitud (en este caso 4).

Existen 2 técnicas para efectuar el almacenamiento de un **Word** en memoria. Las mismas se conocen bajo el nombre de:

**Big Endian** → el byte más significativo precede al menos significativo.

Es decir, el **0058h** se guardaría como **00 58**

**Little Endian** → el byte menos significativo precede al más significativo.

Es decir, el **0058h** se guardaría como **58 00**

En las PC se utiliza el segundo esquema y por ser éstas las más accesibles a los lectores del presente, es el que se utilizará para el desarrollo de los ejemplos de las secciones posteriores.



## 1.6 Direcccionamiento

Modélese a la memoria cual una matriz de 16 columnas donde la cantidad de filas dependerá de la memoria disponible. De esta forma, podemos referenciar una celda de la misma mediante el número de **Fila y Columna**, a los que llamaremos **Segmento y Desplazamiento** respectivamente y que por lo general se expresarán en **hexadecimal**.

	Desplazamiento															
	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
Segmento 0																
Segmento 1																
...																
Segmento n																

De esta forma, en un procesador con palabras de 2 bytes, la celda señalada sería la **\$0001:0007** donde los 2 primeros bytes (Segmento) serían una palabra y los 2 segundos (Desplazamiento) serían otra palabra.

Esta conformación de la dirección no es caprichosa ya que permite “**desplazarnos**” sin cambiar de Segmento cada 16 bytes permitiéndonos mantener un punto fijo (Data Segment, Extra Segment, u otro). De hecho analicemos la siguiente dirección y concluyamos que se trata de la misma celda que la del esquema anterior.

**\$0000:0017**

	Desplazamiento															
	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
Segmento 0																
Segmento 1																
...																
Segmento n																





## 2 Punteros

### 2.1 Concepto

Un puntero es un tipo de variable que **almacena direcciones de memoria**.

Para llevar a cabo su función de almacenamiento debe ocupar el tamaño de una palabra (Word). En el caso de procesadores de 16 bits, serían 2 bytes, para los de 32, de 4 bytes y, para los de 64, 8 bytes.

### 2.2 Usos y ventajas de los punteros

- Permiten el acceso a cualquier posición de la memoria, para ser leída o para ser escrita (en los casos en que esto sea posible).
- Permiten una manera alternativa de pasaje por referencia.
- Permiten solicitar memoria que no fue reservada al inicio del programa. Esto se conoce como uso de memoria dinámica.
- Son el soporte de enlace que utilizan estructuras de datos dinámicas en memoria como las listas, pilas, colas y árboles.

### 2.3 Declaración

Para definir un tipo de dato como puntero tenemos 2 alternativas.

		Pascal	C++
<b>Puntero Genérico</b>	Var	P : <b>pointer</b> ;	void* p;
<b>Puntero a un Tipo de Dato</b>	Var	pInt : <b>^Integer</b> ; pChar : <b>^Char</b> ;	int* pInt; char* pChar;

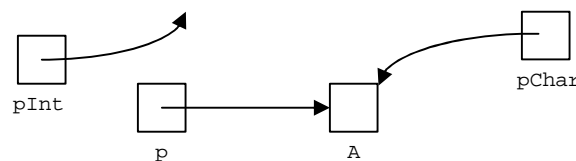
**En el primer caso**, estamos instruyendo al compilador para que sepa que la variable p almacenará una dirección de memoria (será un puntero). Se lleva a cabo mediante el tipo de dato “**pointer**” ó “**puntero a void**”.

**El segundo caso** es análogo pero, además, le indica al compilador como debe interpretar los bytes ubicados en la dirección en cuestión (en el caso de pInt como Entero y en el caso de pChar como char).



## 2.4 Diagramas

Los punteros son variables, por lo que se los modela como tales, es decir: cajas contenedoras con un nombre asociado. La diferencia está en la diagramación de su contenido, el cual bien podría ser una dirección de memoria (\$FA00:4567) pero sería muy confuso trabajarlo. En lugar de ello, se suele representar con flechas que llegan a la dirección de memoria a la cual apuntan.



Claramente se aprecia que “pChar” y “p” apuntan a la misma variable “A”.

## 2.5 Operaciones sobre Punteros

- **Asignación:** el puntero al que se le asigna la dirección debe ser un puntero a un tipo de dato compatible con el de la dirección de memoria que se le quiere asignar. El puntero genérico es siempre compatible.
- **Comparación por Igualdad / Desigualdad:** evalúa si dos direcciones de memoria son las mismas. Los tipos de datos a los que apuntan los punteros deben ser compatibles.
- **Incrementar / Decrementar un puntero:** consiste básicamente en sumar a la dirección de memoria un número entero n. Este número entero no está expresado en bytes sino que se corresponde con  $n * \langle\langle \text{tamaño del tipo de dato apuntado} \rangle\rangle$  bytes. Es decir, si apunto a un entero, y le sumo 1 quedará apuntando al siguiente entero, si apunto a un registro y le sumo 2 quedará apuntando a un área de memoria distante de la primera 2 veces el tamaño del registro. Esto es utilizado para iterar vectores.

	Pascal	C++
Asignación	<code>P1 := P2;</code>	<code>P1 = P2;</code>
Comparación por Igualdad / Desigualdad	<code>P1 = P2;</code> <code>P1 &lt;&gt; P2;</code>	<code>P1 == P2;</code> <code>P1 != P2;</code>
Comparación por Mayor / Menor (o iguales)	<No Soportado>	<code>P1 &gt; P2 / P1 &lt; P2</code> <code>P1 &gt;= P2 / P1 &lt;= P2</code>
Incrementar o Decrementar	<No Soportado>	<code>P1 = P1 + 1</code>



**No es posible realizar las siguientes operaciones:**

- Sumar punteros (esta operación no tiene sentido lógico).
- Multiplicar punteros.
- Dividir punteros.
- Sumarles cantidades que no son enteras.
- Operaciones de desplazamiento.
- Enmascarar punteros (operaciones lógicas).



## 2.6 Referenciación y Desreferenciación

Desreferenciar una variable significa determinar la dirección de memoria en la que se encuentra (todas las variables se encuentran en una dirección de memoria y ocupan una cantidad determinada de bytes desde esa dirección inclusive). Se lleva a cabo mediante el **operador @ en Pascal y el operador & en C++** (también conocido como **Operador de Indirección**).

Referenciar un puntero significa obtener la dirección que esta almacenada en el puntero y almacenar/leer en ella la información en cuestión. Se lleva a cabo mediante el **operador ^ en Pascal y el operador \* en C++**.

Pascal	C++
<pre> Var     P      : pointer;     pInt   : ^Integer;     pChar  : ^Char;     vInt   : Integer;     vChar  : Char;  Begin     vInt := 65;     vChar := 'A'; </pre>	<pre> void *p; int *pInt; char *pChar; int vInt; char vChar;  vInt = 65; vChar = 'A'; </pre>
Asignamos a cada puntero una dirección de una variable de tipo compatible	
<pre> pInt := @vInt; pChar := @vChar; p := @vInt; </pre>	<pre> pInt = &amp;vInt; pChar = &amp;vChar; p = &amp;vInt; </pre>
Imprimir el contenido de lo apuntado	
<pre> WriteLn(pInt ^) {65} WriteLn(pChar ^) {'A'} </pre>	<pre> cout &lt;&lt; *pInt cout &lt;&lt; *pChar </pre>
End	

¿Sería correcto hacer WriteLn(p^) o cout<<\*p?

Dijimos que había dos formas de declarar un puntero y que básicamente se diferenciaban en la forma en que el compilador trataría la zona de memoria a la que el puntero apuntase. Pues bien, el **pointer** o **void\*** no le indicaba al compilador a que tipo de dato apunta con lo que, una vez obtenida la dirección que almacena el puntero:

- 1) ¿Como sabe cuántos bytes leer desde ella en adelante?



- 2) ¿De qué forma debe interpretar esa serie de bytes (como char, como Entero, como String)?

## 2.7 Casteo

De alguna forma, se le debe indicar al compilador el tipo de tratamiento que le debe dar a la información contenida en la sección de memoria cuya dirección almacena un puntero genérico. A este mecanismo se lo denomina casteo.

En Pascal se lleva a cabo mediante la anteposición del tipo de dato que se quiere interpretar seguido del puntero entre paréntesis. Formato → <TipoDeDato>(<Expresión>)

En C++ el análogo de Pascal se define anteponiendo al puntero el tipo de dato como se lo quiere interpretar entre paréntesis. Formato → (<TipoDeDato>) <Expresión>

Sin embargo, C++ cuenta con 3 formas más para castear mediante las siguientes palabras reservadas que se describen con su formato:

```
static_cast    <<TipoDeDato>> ( <Expresión> );
dynamic_cast   <<TipoDeDato>> ( <Expresión> );
reinterpret_cast <<TipoDeDato>> ( <Expresión> );
```

El static\_cast verifica que los tipos de datos sean compatibles en tiempo de compilación.

El dynamic\_cast verifica que los tipos de datos sean compatibles en tiempo de ejecución.

El reinterpret\_cast no verifica que los tipos de datos sean compatibles (es análogo al comportamiento con paréntesis).

De esta forma, podríamos...

Pascal	C++
Imprimir el contenido de la dirección guardada interpretando a p como pInt o pChar	
WriteLn ( TpInt (p)^ ) {65}	cout << *static_cast<int*>(p)
WriteLn ( TpChar(p)^ ) {'A'}	cout << *static_cast<char*>(p)

En definitiva, mediante el casteo forzamos al compilador a que dada una dirección de memoria, sea la contenida en un puntero o la de una variable cualquiera, la interprete conforme el tipo de dato que le indicamos al castear.



## 2.8 Asignación Dinámica de Memoria

¿Qué sentido tiene manejar una variable declarada por nosotros mediante punteros? ¿Por qué llamar  $p^*$  o  $*p$  a nuestra variable  $A$  en lugar de, simplemente,  $A$ ? La respuesta es sencilla: no tiene sentido.

Los punteros nos permiten generar estructuras en memoria alocada en tiempo de ejecución (**estructuras dinámicas**). Esta memoria alocada en tiempo de ejecución corresponde al área denominada Heap.

Para ello contamos con varias instrucciones para el manejo / asignación de memoria dinámica; a saber:

	Pascal	C++
<b>Reservar Memoria (Puntero a Tipo)</b>	<code>New (pInt);</code> ó <code>pInt = new(integer);</code>	<code>pInt = new int;</code>
<b>Liberar Memoria (Puntero a Tipo)</b>	<code>Dispose (pInt);</code>	<code>delete pInt;</code>
<b>Reservar Memoria (Puntero Genérico)</b>	<code>GetMem (p, 10);</code>	<code>p = (int*) malloc(10);</code> //función de C, no recomendada
<b>Liberar Memoria (Puntero Genérico)</b>	<code>FreeMem (p, 10);</code>	<code>free(p);</code> //función de C, no recomendada

Ahora bien, la memoria Heap no es infinita, con lo que podría no haber memoria para alojar. En este aspecto los lenguajes tienen políticas distintas; a saber:

Pascal lanzaría un error de solicitarse más memoria de la disponible, por lo tanto hay que prever este problema antes de efectuar la invocación a las funciones de asignación y para ello se cuenta con las funciones:

**function MaxAvail: Longint;** → devuelve el tamaño del mayor bloque continuo factible de ser alocado

**function MemAvail: Longint;** → devuelve el total de memoria factible de ser alocada.

Cabe destacar que, hasta la estandarización de C++ de julio de 1998, el operador `new` retornaba un puntero a `NULL` en caso de no poder alojar la memoria solicitada. A partir de dicha estandarización, arroja una excepción del tipo `bad_alloc`, la cual debería tener un manejador adecuado.

Por otro lado, si bien es posible utilizar las funciones de asignación de memoria de C en C++, éstas retornan punteros a `void`, por lo que se deberá realizar después el casting al tipo



requerido. En contraposición, `new` devuelve un puntero del tipo especificado e incluso, cuando se trabaja bajo el paradigma de POO, construye un objeto.

## 2.9 Dirección NULA

La palabra reservada **NIL** de Pascal o la macro definida en el librería `cstdlib` (adaptación a C++ de la librería `stdlib.h`, de C) **NULL** de C++ representan la dirección utilizada para indicar que un puntero no contiene una referencia concreta. De hecho, el hacer referencia a la misma generaría un error en tiempo de ejecución por tratar de acceder a una dirección de memoria prohibida.

<b>Pascal</b>	<b>C++</b>
<code>pInt := nil;</code>	<code>pInt = NULL;</code>
<code>pInt^ := 10;</code>	<code>*pInt = 10;</code>
<code>{Error ya que pInt es nulo}</code>	<code>// Error ya que pInt es nulo</code>

## 2.10 Memoria Colgada

Supongamos el siguiente fragmento de código...

### Pascal

```
Var
    p1, p2 : ^Integer;

Begin
    New (p1);      {se reserva un integer en el heap y se asigna su dirección a p1}
    New (p2);      {se reserva otro integer en el heap y se asigna su dirección a p2}
    p1 := p2       {se asigna a p1 la misma dirección que p2}
    {...}
End.
```

### C++

```
int *p1;
int *p2;

p1 = new int; // se reserva un integer en el heap y se asigna su dirección a p1
p2 = new int; // se reserva otro integer en el heap y se asigna su dirección a p2
p1 = p2       // se asigna a p1 la misma dirección que p2
{...}
```

Podemos observar que más haya de las operaciones que realicemos a posteriori, la dirección del primer integer reservado que estaba en `p1`, a raíz de la última asignación, se ha perdido (no la tenemos en ningún puntero).



Esto significa que no podremos liberar esa memoria por cuanto las operaciones de desalocación requieren que se les pase la dirección a desalocar (la cual hemos perdido).

A esta situación se la conoce como **“Memoria Colgada”** (Memory Leak, en inglés)

## 2.11 Referencia Perdida

Supongamos el siguiente fragmento de código...

### Pascal

```
Var
    p1, p2 : ^Integer;

Begin
    New (p1);      {se reserva un integer en el heap y se asigna su dirección a p1}
    p2 := p1;      {se asigna a p2 la misma dirección que p1}
    Dispose (p1);  {se libera la memoria reservada}
    {...}
End.
```

### C++

```
int *p1;
int *p2;

p1 = new int; // se reserva un integer en el heap y se asigna su dirección a p1
p2 = p1;      // se asigna a p2 la misma dirección que p1
delete p1;    // se libera la memoria reservada
{...}
```

Podemos observar que en este momento, si efectuásemos operaciones sobre lo apuntado por p2 estaríamos realizándolas sobre una memoria ya liberada y que por lo tanto puede haber sido utilizada (luego de su liberación) para otros fines.

A esta situación se la conoce como **“Referencia Perdida”**.

## 2.12 Punteros a función

Si bien una función no es una variable, tiene asociada una dirección de memoria correspondiente al comienzo de su código.

A través del nombre de la función es que se puede acceder a la misma, podemos considerar entonces al nombre de la función como un puntero que apunta al inicio de la función.

Si bien esto es similar a lo que ocurre con los nombres de los arreglos, en cuanto a que ellos





apuntan a la dirección de inicio del array, existe una diferencia conceptual importante:

**Mientras que el nombre de un array apunta al Data Segment o Stack Segment, el nombre de una función apunta al Code Segment.**

### 2.12.1 Definir un Tipo de Puntero a Función en C++

```
tipo (*nombre_puntero)(tipo_arg1, tipo_arg2, ... , tipo_arg3);
```

<code>tipo</code>	→ es el tipo de dato devuelto por la función.
<code>nombre_puntero</code>	→ es el identificador de la variable puntero.

Si la función a la que se desea apuntar no tiene argumentos, simplemente se dejan los paréntesis abierto y cerrado.

Hay que notar que el nombre del tipo función está entre paréntesis para diferenciar un puntero a función de una función que devuelve un puntero.



## 2.12.2 Asignar la Dirección de la Función:

```
nombre_puntero = nombre_funcion;
```

## 2.12.3 Invocar la función a través del Puntero:

```
(*nombre_puntero)(A, B)      ó      *nombre_puntero(A, B)
```

siendo **A** y **B** los argumentos.

## 2.12.4 Ejemplo

Se considerarán 2 funciones; **sumar** y **restar** y un arreglo de 2 punteros a función llamado **p[ ]**. La asignación de dicho puntero tendrá lugar simultáneamente con su declaración. La selección se realizará a través del subíndice del array de punteros.

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

int sumar (int x, int y) {
    return x + y;
}

int restar (int x, int y) {
    return x - y;
}

int main (void) {
    int a, b, opcion;
    int (*p[2])(int, int) = {sumar, restar};
    std::cout << "Ingrese 2 valores enteros." << std::endl;
    std::cin >> a;
    std::cin >> b;

    do {
        std::cout << " Ingrese opción 0-Sumar 1-Restar " << std::endl;
        std::cin >> opcion;
    } while ((opcion != 1) && (opcion != 0));

    std::cout << " El resultado es :" << (*p[opcion])(a, b) << std::endl;
    return 0;
}
```



## 3 Ejemplos

### 3.1 Ejemplo de Casteo: Cómo es un Integer?

```

TYPE
  IntegerEnBytes = Array[1..2] of Byte;
  pIntegerEnBytes = ^IntegerEnBytes;

VAR
  I : Integer;
  p : pIntegerEnBytes;

BEGIN
  {1} I := 98;           { en Hexadecimal: 62h}
  {2} p := pIntegerEnBytes( @I );
      WriteLn( P^[1]);   {98}
      WriteLn( P^[2]);   { 0}
END.

```

La idea de este ejemplo es, por un lado, ver como es posible castear un tipo de lo más diverso a otro y, por otro lado, ver como se compone un Integer en memoria.

Para lograr nuestro objetivo, diagramamos a continuación el estado de la memoria en los diversos instantes enumerados en el código; a saber:

Nota: a diferencia de lo acostumbrado, y para poder detallar el ejemplo, las cajitas simbolizarán bytes y el recuadro en el que se encuentran simbolizarán a la variable (conjunto de bytes)

Inicial	Después de 1	Después de 2
<div style="border: 1px solid black; padding: 5px; margin-bottom: 10px;"> <div style="border: 1px solid black; width: 40px; height: 20px; margin: 0 auto; display: flex; justify-content: space-around;"> <span>-</span><span>-</span> </div> <p>I - \$C001:0010</p> </div> <div style="border: 1px solid black; padding: 5px;"> <div style="border: 1px solid black; width: 80px; height: 20px; margin: 0 auto; display: flex; justify-content: space-around;"> <span>-</span><span>-</span><span>-</span><span>-</span> </div> <p>P - \$C001:0012</p> </div>	<div style="border: 1px solid black; padding: 5px; margin-bottom: 10px;"> <div style="border: 1px solid black; width: 40px; height: 20px; margin: 0 auto; display: flex; justify-content: space-around;"> <span>62</span><span>00</span> </div> <p>I - \$C001:0010</p> </div> <div style="border: 1px solid black; padding: 5px;"> <div style="border: 1px solid black; width: 80px; height: 20px; margin: 0 auto; display: flex; justify-content: space-around;"> <span>-</span><span>-</span><span>-</span><span>-</span> </div> <p>P - \$C001:0012</p> </div>	<div style="border: 1px solid black; padding: 5px; margin-bottom: 10px;"> <div style="border: 1px solid black; width: 40px; height: 20px; margin: 0 auto; display: flex; justify-content: space-around;"> <span>62</span><span>00</span> </div> <p>I - \$C001:0010</p> </div> <div style="border: 1px solid black; padding: 5px;"> <div style="border: 1px solid black; width: 80px; height: 20px; margin: 0 auto; display: flex; justify-content: space-around;"> <span>10</span><span>00</span><span>01</span><span>C0</span> </div> <p>P - \$C001:0012</p> </div>

Ahora bien, p es un puntero a un registro compuesto por 2 bytes y como mediante el casteo forzamos a que apuntara a la dirección de I (un entero), estos dos bytes deberían coincidir físicamente con los dos bytes que componen un integer. De hecho, al imprimir se confirman nuestros diagramas (es muy ilustrativo asignarle a I un valor mayor que 255 y ver que la segunda salida deja de ser cero).



¿Qué significado tuvo el casteo en este ejercicio? El casteo le indica al compilador la forma en la que debe interpretar la dirección de memoria que recibe entre paréntesis. En el ejemplo, sin el casteo no podríamos haber asignado a p la dirección de un entero, ya que el compilador nos lo prohibiría por tratarse de tipos distintos (un puntero a un registro no espera la dirección de un entero, espera la dirección de un registro).



## 3.2 Ejemplo de Punteros: Swap

### 3.2.1.1 En Pascal

```

TYPE
    TPChar = ^Char;
    TPPChar = ^TPChar;

PROCEDURE Swap1 ( p1 : TPChar; p2 : TPChar);
VAR
    Aux : TPChar;
BEGIN
    Aux := p1;
    p1 := p2;
    p2 := Aux;
END;

PROCEDURE Swap2 ( var p1 : TPChar; var p2 : TPChar);
VAR
    Aux : TPChar;
BEGIN
    Aux := p1;
    p1 := p2;
    p2 := Aux;
END;

PROCEDURE Swap3 ( p1 : TPChar; p2 : TPChar);
VAR
    Aux : Char;
BEGIN
    Aux := p1^;
    p1^ := p2^;
    p2^ := Aux;
END;

PROCEDURE Swap4 ( p1 : TPPChar; p2 : TPPChar);
VAR
    Aux : TPChar;
BEGIN
    Aux := p1^;
    p1^ := p2^;
    p2^ := Aux;
END;

VAR
    c1: Char;
    c2: Char;
    pc1 : TPChar;
    pc2 : TPChar;

BEGIN
    c1 := 'A';
    c2 := 'B';
    pc1 := @c1;
    pc2 := @c2;

    Swap1 (pc1, pc2);
    WriteLn(c1, c2, pc1^, pc2^); {ABAB}

    Swap2 (pc1, pc2);
    WriteLn(c1, c2, pc1^, pc2^); {ABBA}

    Swap3 (pc1, pc2);
    WriteLn(c1, c2, pc1^, pc2^); {BABA}

    Swap4 (@pc1, @pc2);
    WriteLn(c1, c2, pc1^, pc2^); {ABBA}

END.
```



## En C++

```
void Swap1 ( char* p1, char* p2) {
    char* aux = p1;
    p1        = p2;
    p2        = aux;
}
```

```
void Swap2 ( char* &p1, char* &p2) {
    char* aux = p1;
    p1        = p2;
    p2        = aux;
}
```

```
void Swap3 ( char* p1, char* p2) {
    char  aux = *p1;
    *p1     = *p2;
    *p2     = aux;
}
```

```
void Swap4 ( char** p1, char** p2) {
    char* aux = *p1;
    *p1     = *p2;
    *p2     = aux;
}
```

```
void main () {
    char c1      = 'A';
    char c2      = 'B';
    char* pc1    = &c1;
    char* pc2    = &c2;
```

```
    Swap1( pc1, pc2);
    cout << c1 << c2 << *pc1 << *pc2;
```

```
    Swap3( pc1, pc2);
    cout << c1 << c2 << *pc1 << *pc2;
```

```
    Swap2( pc1, pc2);
    cout << c1 << c2 << *pc1 << *pc2;
```

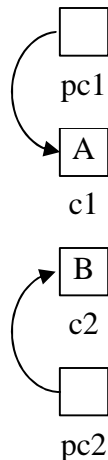
```
    Swap4(&pc1, &pc2);
    cout << c1 << c2 << *pc1 << *pc2;
```

```
}
```



### 3.3 Análisis de los distintos Swap

Hasta antes de la invocación de los SwapN, la distribución de memoria es semejante y es la siguiente:

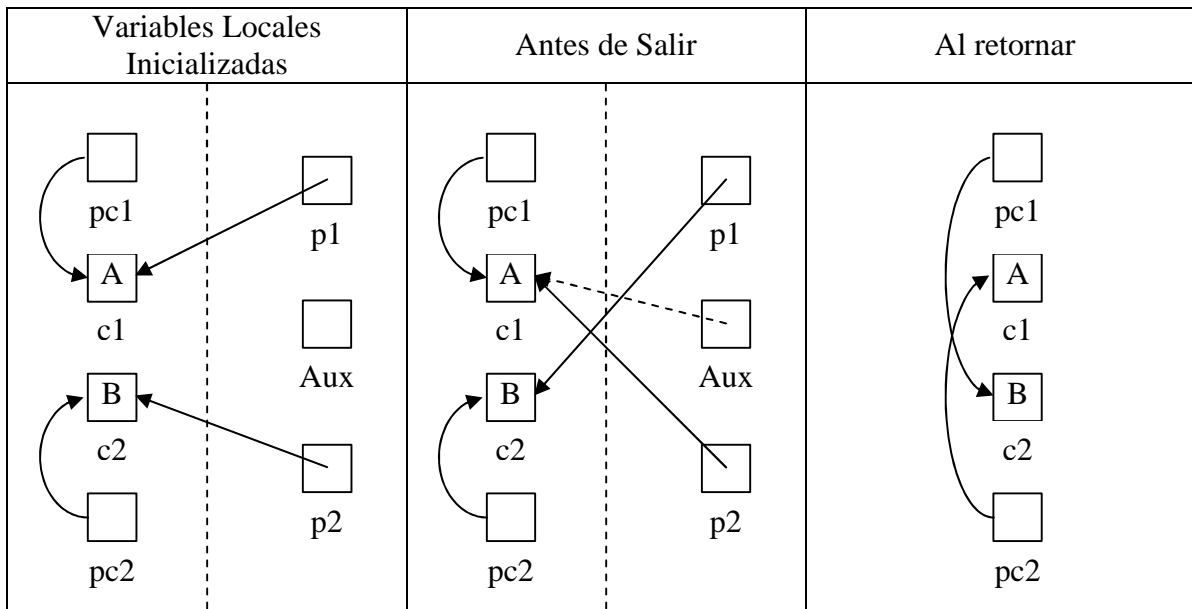


#### 3.3.1 Invocación a Swap1

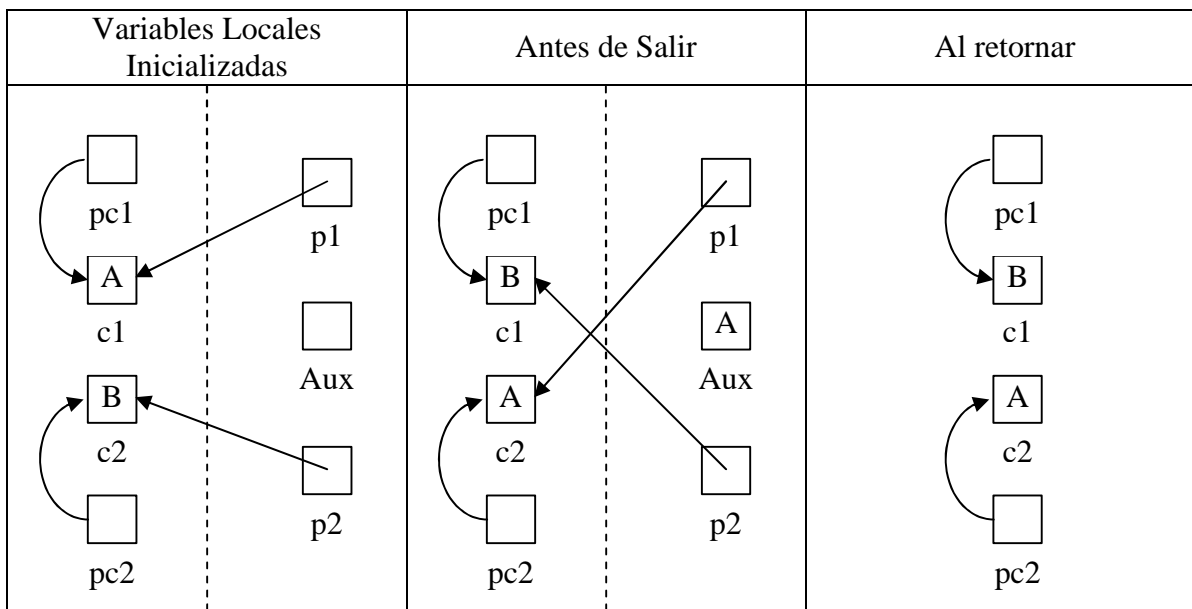
Variables Locales Inicializadas	Antes de Salir	Al retornar



### 3.3.2 Invocación a Swap2



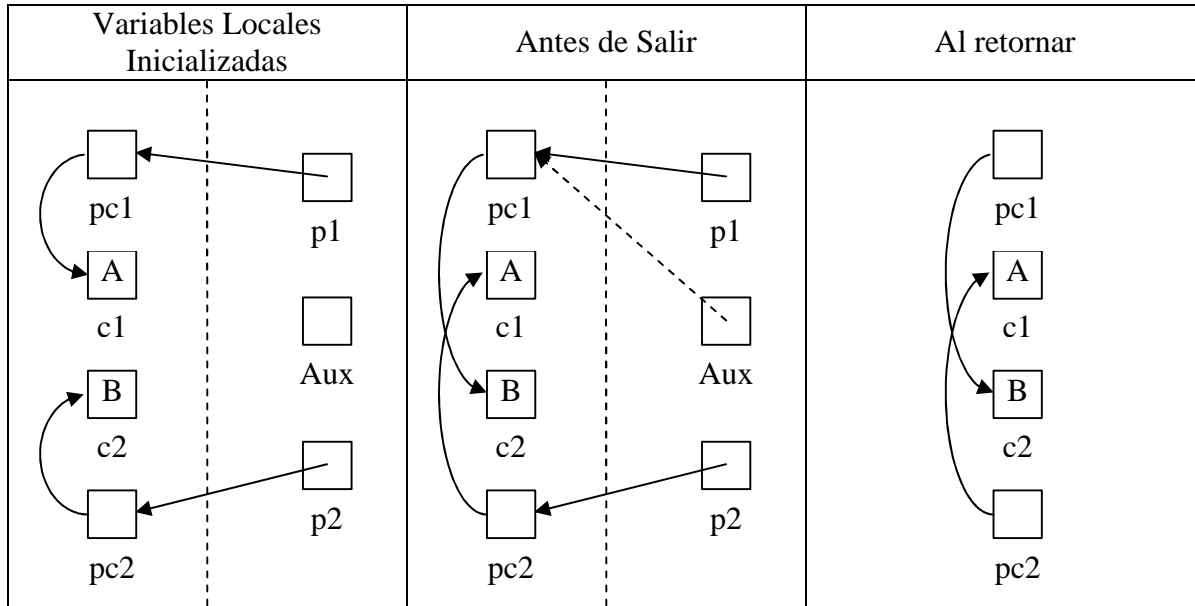
### 3.3.3 Invocación a Swap3







### 3.3.4 Invocación a Swap4



### 3.3.5 Comentarios:

**Swap1:** si bien el swap intercambia el contenido de p1 y p2 este cambio no se refleja en pc1 y pc2 por cuanto eran variables pasadas por copia, no por referencia.

**Swap2:** Se realizó un swap de las direcciones de memoria contenidas por pc1 y pc2 (en esta oportunidad se refleja el cambio debido al pasaje por referencia)

**Swap3:** Se efectuó el intercambio del contenido de lo apuntado por pc1 y pc2, es decir los punteros siguen apuntando a las mismas direcciones pero el contenido de las direcciones apuntadas fue el swapeado.

**Swap4:** Se efectuó el intercambio del contenido de pc1 y pc2. Obsérvese que el esquema es similar al del Swap3 ya que ambos intercambian el contenido de lo apuntado por los punteros pasados por parámetro pero la diferencia radica en el tipo de contenido (Chars en el Swap3 y punteros a char en el Swap4).

Si se observan los resultados: ABAB, ABBA, BABA, ABBA se puede apreciar que el Swap2 y el Swap4 coinciden en su resultado final. ¿Será casualidad que el pasaje por referencia y el pasaje por valor de la indirección coincidan en cuanto al resultado final?

La respuesta es no, al indicarle al compilador que un pasaje se realiza por referencia estamos instruyéndolo para que implemente el esquema planteado en Swap4. Obviamente,



y debido a que es un beneficio que nos brinda el compilador, esto es transparente al programador.



### 3.4 Ejemplo de Casteo: Que hace?

#### En Pascal

Program Ejercicio;

{ Aclaraciones:

1) En la Tabla ASCII:

A = 65; B = 66; ... a = 97; b = 98; ...

2) La posición 0 de un string es su longitud.

3) procedure GetMem(var P: Pointer; Size: Word);  
le asigna a P una zona de memoria de tamaño Size

4) procedure FreeMem(var P: Pointer; Size: Word);  
devuelve la memoria reservada con GetMem. Size debe coincidir.

}

Type

Str20 = string[20]; PChar = ^char; PStr20 = ^Str20; PByte = ^Byte;

Procedure ProcA ( p1: PChar);

Var i : byte; j : char;

Begin

i := p1^;

While (i>0) do

Begin

j := (PStr20(p1)^[i]);

If (j>='a') and (j<='z') Then

Begin

j := byte(j) - 32;

(PStr20(p1)^[i]) := j;

End;

i := i-1;

End;

End;

Procedure ProcB ( p1: PChar; p2: PChar);

Var i : byte;

Begin

for i := 0 to Byte(p1^)-1 do

(PStr20(p2)^[i]) := (PStr20(p1)^[i+1]);

i := i+1;

(@PStr20(p2)^[i])^ := 0;

End;

Procedure ProcC( var p1: byte);

Var i: byte;

Begin

i:=0;

while PStr20(@p1)^[i] <> 0 do

Begin

Write(PStr20(@p1)^[i]);

i := i+1;

End;

WriteLn;

p1 := 0;

End;

Var c : Str20; d : PByte;

BEGIN

ReadLn (c );

GetMem(d, Length(c)+1);

ProcA (@c);

WriteLn (c );

ProcB (@c , d);

ProcC (d^);





```
FreeMem(d, Length(c)+1);  
END.
```



El objetivo del Ejercicio es determinar que operaciones se realizan en los procedimientos ProcA, ProcB y ProcC y, determinar y subsanar posibles faltas de casteo en el código.

Comenzaremos analizando el bloque principal y diagramaremos la memoria con una matriz que tiene por columnas los tipos de datos y por fila el orden de dirección. Con lo cual se entiende que cualquier asignación que implique un cruce de columnas deberá implementarse vía **casteo** (salvo String con char por motivos señalados mas abajo). Y el contenido de las variables ubicadas en la segunda y tercer fila sería lógico que apunte a variables de la primera y segunda fila respectivamente.

	Str20	Char	Byte
var	c 		
^			d 
^^			

BEGIN




Esta es la disposición inicial de la memoria, c es una variable de tipo Str20 y d es un puntero a Byte.

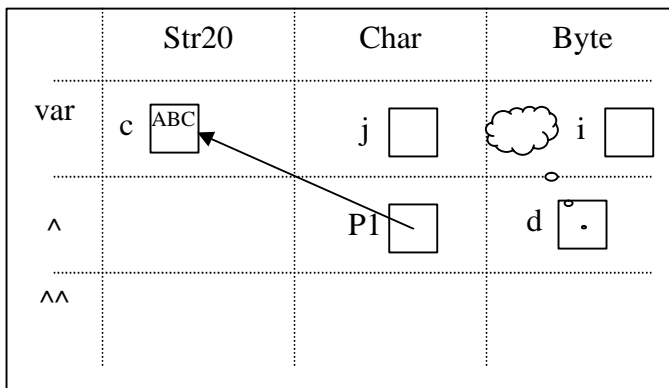
**ReadLn(c);**  
**GetMem(d, Length(c) + 1)**

El readln carga un contenido en c que para el caso podría ser 'ABC'.

El getmem reserva memoria y almacena la dirección del comienzo de la misma en d.

El tamaño reservado es la longitud de la cadena leída + 1 (4).

	Str20	Char	Byte
var	c 		
^			d 
^^			



### ProcA (@c) → ProcA (p1:pchar)

ProcA espera recibir la dirección de un char y un string es una lista de chars con lo que la dirección del string es la dirección de un char (del primer char, que en particular guarda la longitud del string).

**I = p1^**

...se puede entender como ...

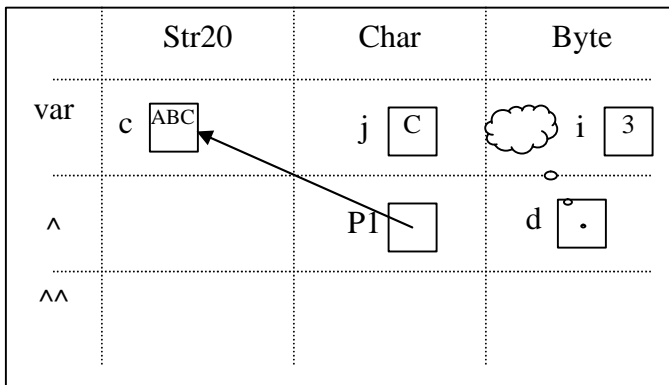
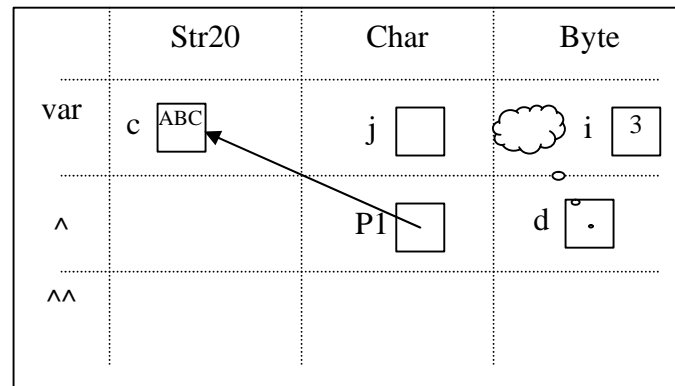
<byte> := <PStr20>^

<byte> := <Str20> **ERROR**

...para subsanarlo casteamos así

**I = Byte(p1^)** ó **I = PByte(p1^)**

I queda con el valor de 3 que es el valor (interpretado como Byte) del primer byte de la memoria a la que apunta P1 (el comienzo del String c).



Dentro del ciclo, **j = Pstr20(p1)^[i]**

Veámoslo de esta manera...

<Char> = **PStr20** (<Pchar> )^[ <byte> ]

<Char> = <PStr20> ^ [ <byte> ]

<Char> = <Str20> [ <byte> ]

<Char> = <Char>

con lo que concluimos que no hay errores con los tipos de dato. El valor que toma j por ende es 'C' ya que es la posición i=3 del string apuntado por P1.



**J:=Byte(J)-32;**

Léase como...

<char>:=Byte ( <char> ) - <byte>;

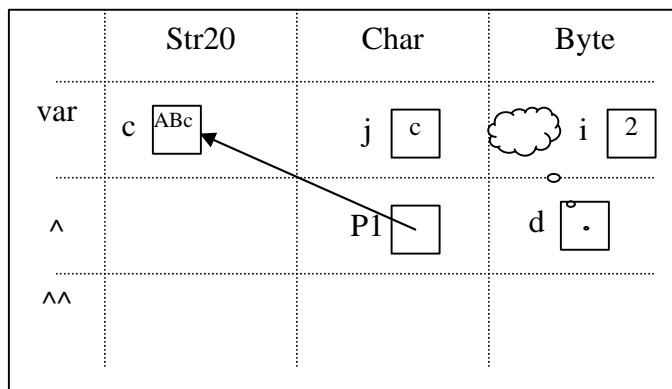
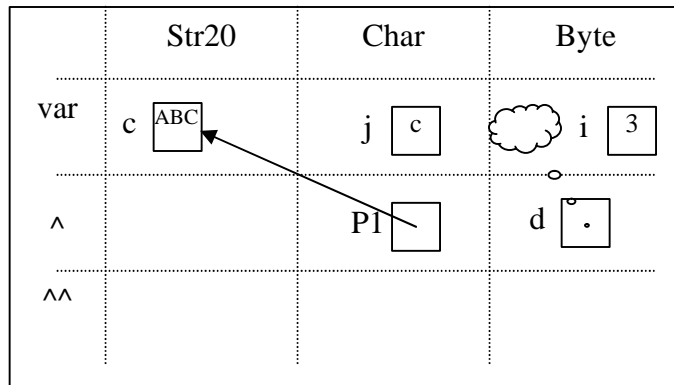
<char>:=<byte>-<byte>

<char>:=<byte> **ERROR**

Para subsanarlo castearemos cómo se muestra a continuación...

J:=char(Byte(J)-32);

Luego, el caracter que se encuentra 32 posiciones más abajo en la tabla ASCII de una Letra Mayúscula es su minúscula, con lo que j:=c;



**Pstr20(p1)^[i]=j;**

**End**  
**I=I-1**

Lo que hará es volver a poner en la posición de la que fue extraído el carácter de j pero en minúscula, luego se decrementará i para continuar el ciclo.

Finalmente, esto se repetirá hasta que i = 0 con lo que deducimos que el procA convierte a minúsculas el string recibido por parámetro.

Queda como ejercicio para el lector continuar el análisis del resto del código...