Memoria y Punteros

- Materia Algoritmos y Estructuras de Datos
- Cátedra Schmidt
- Esquema de memoria y manejo del tipo puntero

Todo dato con que trabaja la computadora, (así como toda instrucción cuando se ejecuta), en los modelos habitualmente usados de computadoras, está en la memoria principal.

Tanto los datos como las instrucciones están codificados como sucesiones de ceros y unos.

Al momento de ejecutar un programa, para ese programa en particular se distinguen cuatro zonas distintas, todas ubicadas en la memoria principal de la computadora.

Cada zona es un ‘segmento’ y tiene una función determinada.

1 Memoria

1.1 División

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

CD	: Code Segment
DS	: Data Segment
SS	: Stack Segment
ES	: 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.

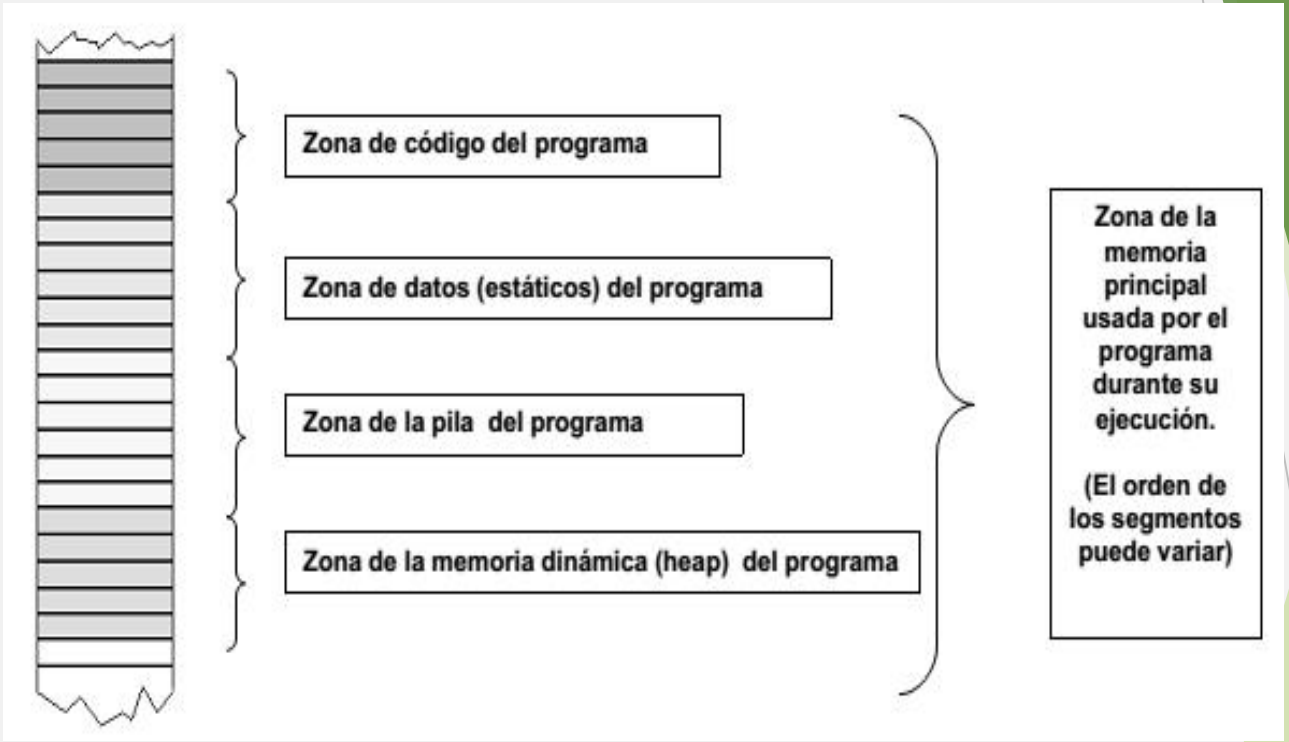
El **Data Segment** almacenará el contenido de las variables definidas en el modulo principal (variables glabales)

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.

Formas de graficar la memoria

Grafico de memoria contigua



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)

Vamos a considerar en particular a los datos.
¿De qué modo se almacenan en la memoria?

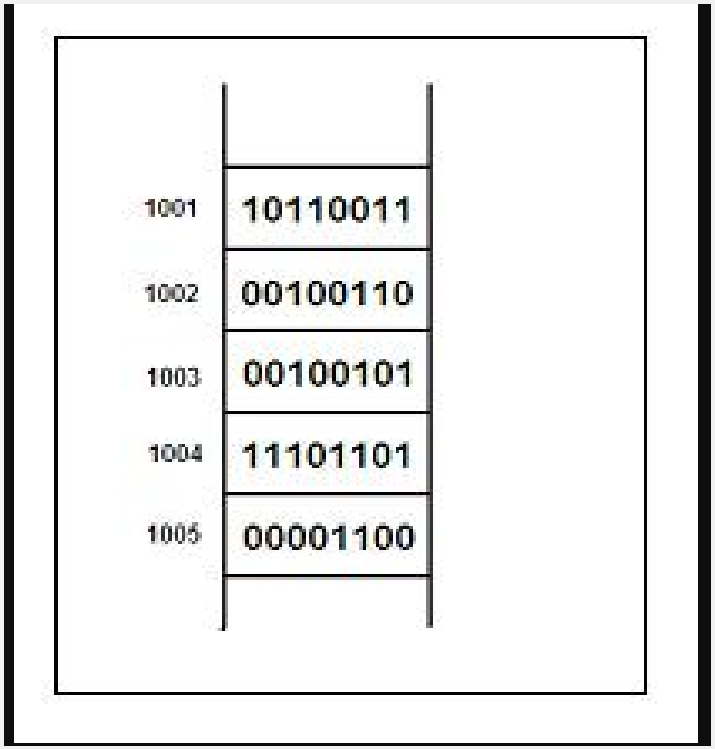
Hay que recordar algunas conceptos: la memoria de la computadora se compone de celdas. Cada celda es un byte, es decir que está formada por 8 bits.

Todas las celdas de la memoria tienen el mismo tamaño, y son idénticas entre sí; solo se diferencian entre ellas por el número de posición que tienen asignado.

El número de posición de una celda es la **dirección** de la celda. Esa dirección es única para cada celda y no puede ser cambiada.

En ningún momento una computadora tiene ‘celdas vacías’. En toda celda hay bits. Cada bit puede tomar valor cero o uno.

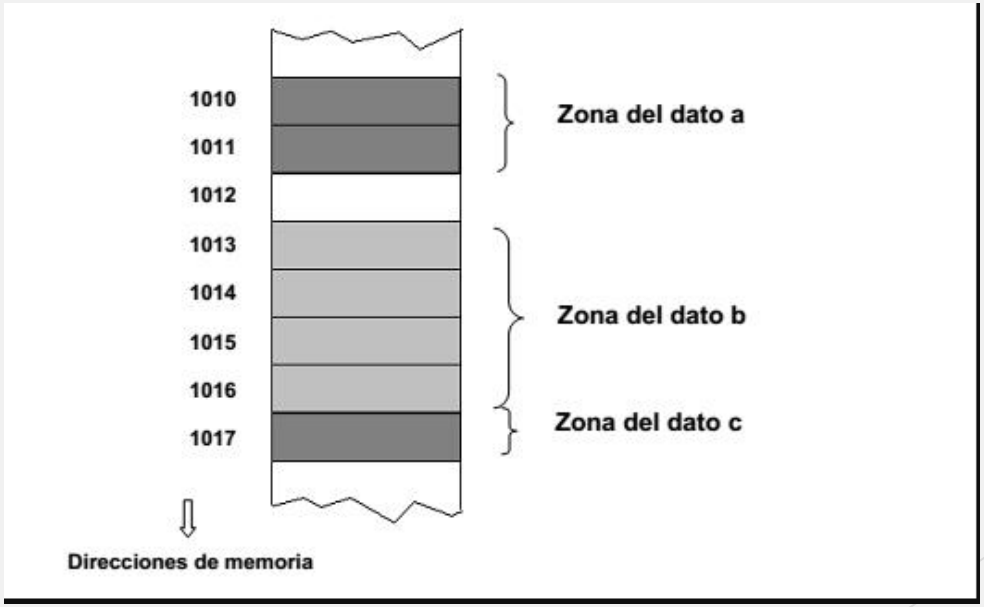
Mediante la codificación con dígitos binarios la computadora trata tanto los datos como las instrucciones.



Cada dato contenido en la memoria ocupa una o más celdas, dependiendo del tamaño del mismo.

Por ejemplo, los caracteres ocupan una sola celda de memoria, los enteros ocupan al menos 2 celdas (en C, en C++ el tamaño está estandarizado), los reales ocupan al menos 4 celdas, y otros datos ocupan más celdas.

En todo caso, lo que siempre se verifica es que un dato, del tipo que sea, ocupa un conjunto de celdas que deben ser contiguas.



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 › **0**101 1000 › 5 8 › 58h

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

58h › 5 8 › **0**101 1000 › 1011000b

Binario › Decimal : multiplicar cada bit por (2 elevado al peso del bit) y sumar los resul (Ctrl)

Dígitos	1	0	1	1	0	0	0
Peso	6	5	4	3	2	1	0
2^{peso}	64	32	16	8	4	2	1
Digito x 2^{peso}	64	0	16	8	0	0	0
Suma	**88**						

Decimal › 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 › Decimal : multiplicar cada dígito por (16 elevado al peso del bit) y sumar los resultados

Dígitos	5	8
Peso	1	0
2^{peso}	16	1
Digito x 2^{peso}	80	8
Suma	**88**	

Decimal › 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**	**5**

55

(Nota: 5 es el dígito hexadecimal que le corresponde al 5 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 mas 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**

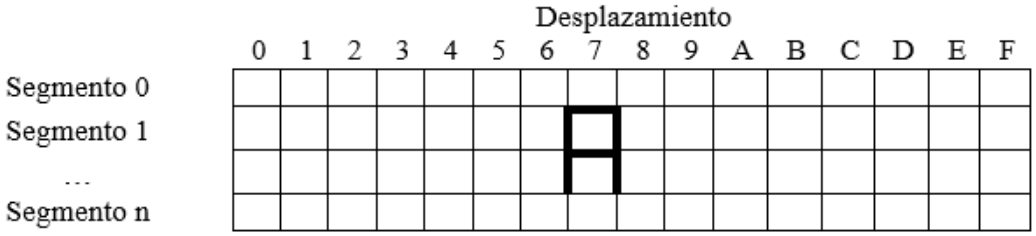
Little Endian → el byte menos significativo precede al más significativo.

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 Direccionamiento

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**.



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

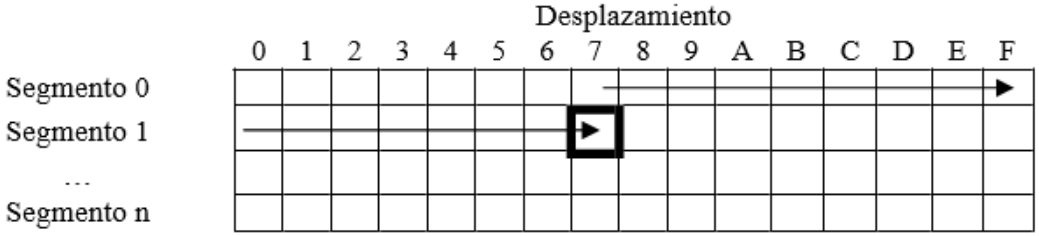
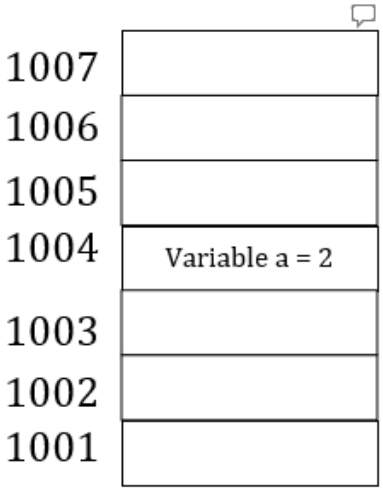
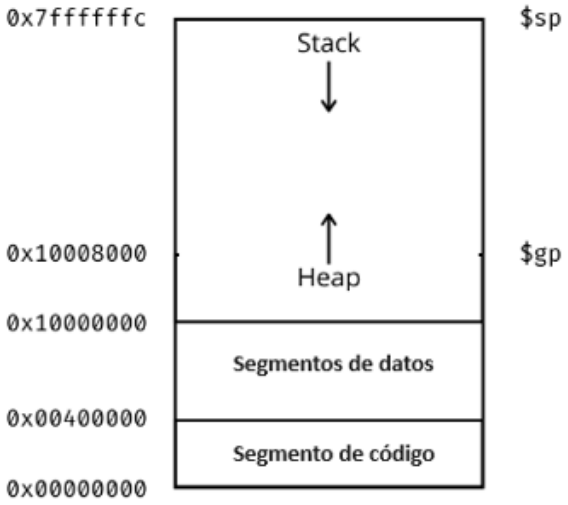


Diagrama de memoria contigua

```
11
12 int main() {
13     short a = 2;
14     return 0;
15 }
16
```

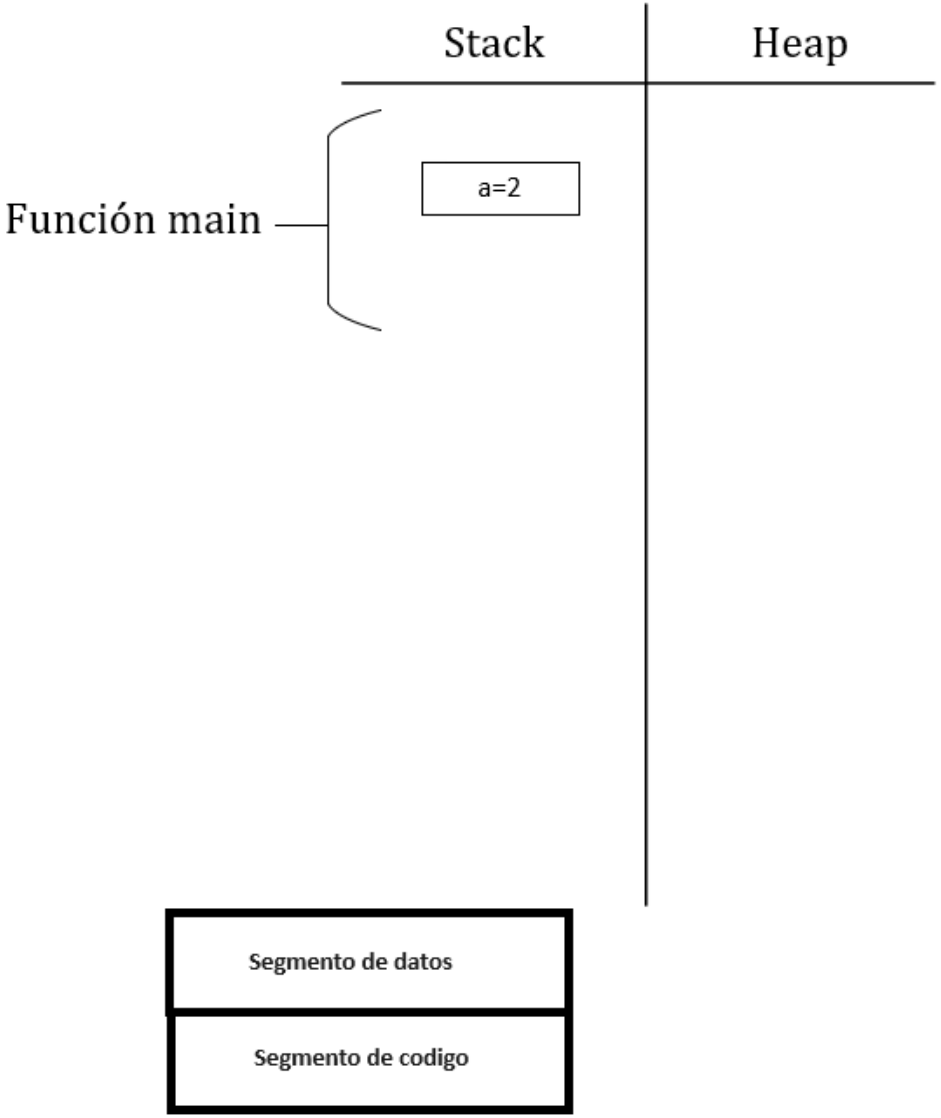




Al momento de terminar de ejecutar la linea 13, la variable a queda posicionada en la dirección de memoria 1004. Y el valor de a es 2.

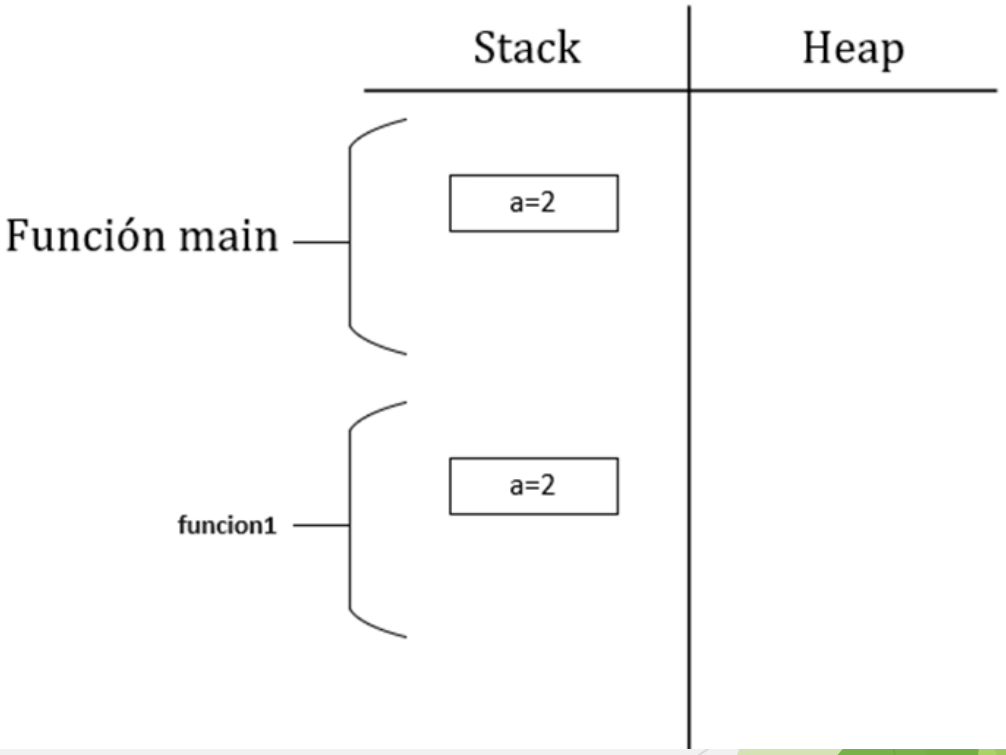
Diagrama de memoria practico

```
11
12 int main() {
13     short a = 2;
14     return 0;
15 }
16
```



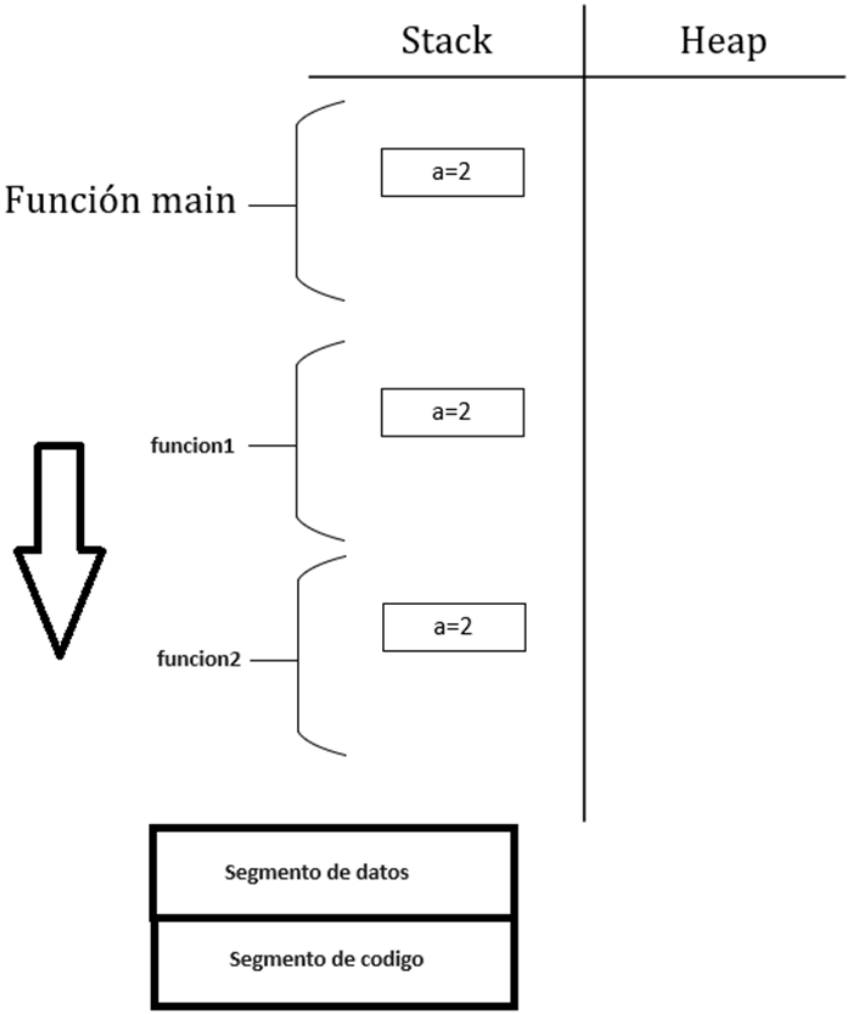
```
12
13 int funcion1() {
14     short a = 2;
15     return 0;
16 }
17
18 int main() {
19     short a = 2;
20     funcion1();
21     return 0;
22 }
23
```

Diagrama de memoria practico



```
11
12 int funcion2() {
13     short a = 2;
14     return 0;
15 }
16
17
18 int funcion1() {
19     short a = 2;
20     funcion2();
21     return 0;
22 }
23
24 int main() {
25     short a = 2;
26     funcion1();
27     return 0;
28 }
```

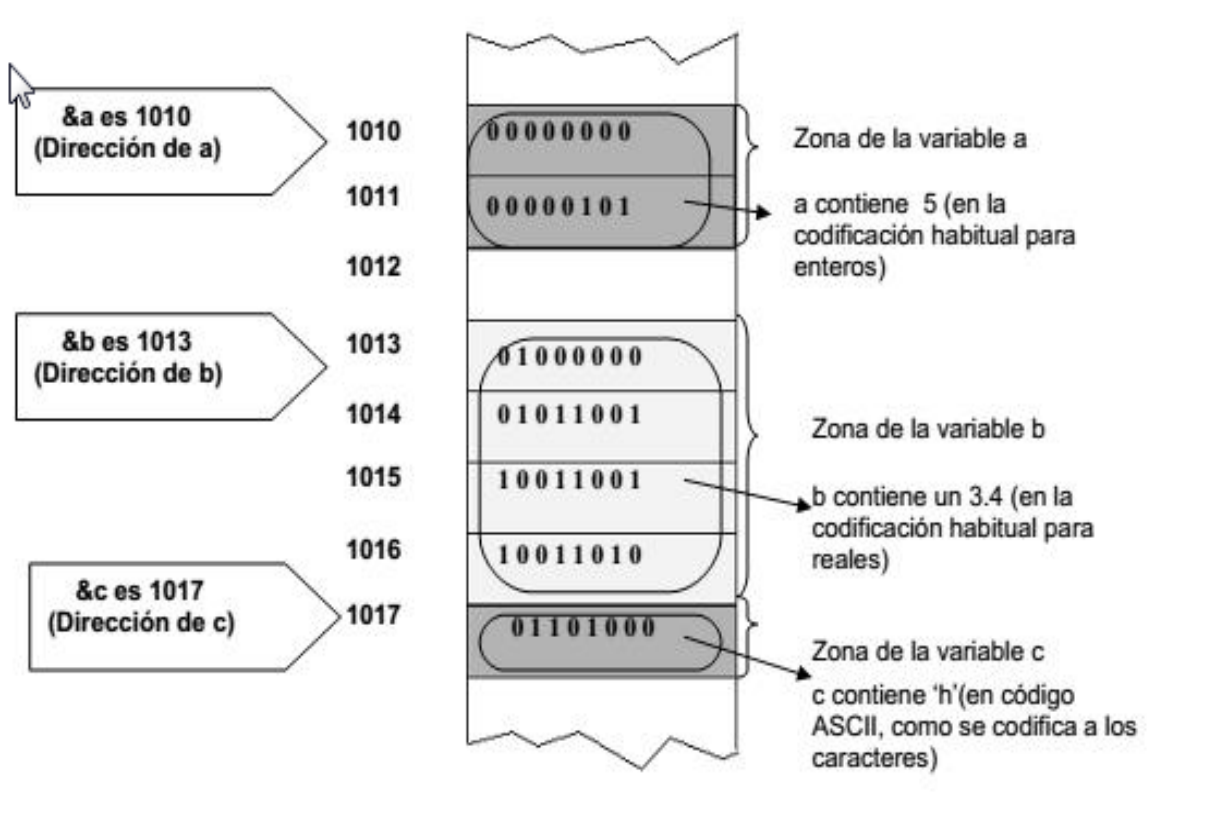
Diagrama de memoria practico



```
# include <iostream>
using namespace std;

int main()

{

   int a =5, b=10;

  cout<<a<<" " <<&a<<endl;
  cout<<b<<" " <<&b<<endl;

   return 0;
}
```

Los punteros

Un puntero es un tipo de dato que corresponde a una dirección de memoria, o al valor NULL (0, cero). Sirven para manipular direcciones de elementos (de 'objetos', usando esta palabra en sentido general).



En el gráfico anterior se ha indicado el contenido de cada variable tal como se almacena; a la derecha se especifica el valor correspondiente a cada secuencia de bits (los codificados en ASCII, los enteros en complemento a 2, los reales en la notación IEEE 754).

Según la situación graficada antes a modo de ejemplo, a contiene el dato 5.

La dirección de a, llamada &a es 1010, b contiene el dato 3.4.

La dirección de b es &b, con valor 1013 c contiene el dato ‘h’.

La dirección de c es &c, con valor 1017.

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.

Puntero Genérico:

```
void* p;
```

Puntero a un Tipo de Dato:

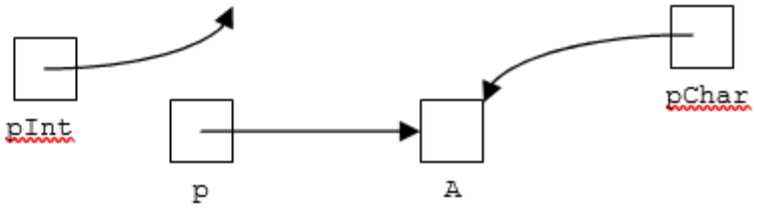
```
int * pInt;
char * pChar
```

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 **"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 esta 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 * <<tamaño del tipo de dato apuntado>> 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.

	C++
Asignación	`P1 = P2;`
Comparación por Igualdad / Desigualdad	`P1 == P2;` `P1 != P2;`
Comparación por Mayor / Menor (o iguales)	`P1>P2 / P1< P2` `P1>=P2 / P1<= P2`
Incrementar o Decrementar	`P1 = P1 + 1`

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 acabo mediante el **operador 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 C++.**

C++

```
void *p;
int  *pInt;
char *pChar;
int   vInt;
char  vChar;


vInt  = 65;
vChar = 'A';
```

tipo compatible

```
pInt  =  &vInt;
pChar = &vChar
p     = &vInt;
```

```
cout << *pInt
cout << *pChar
```

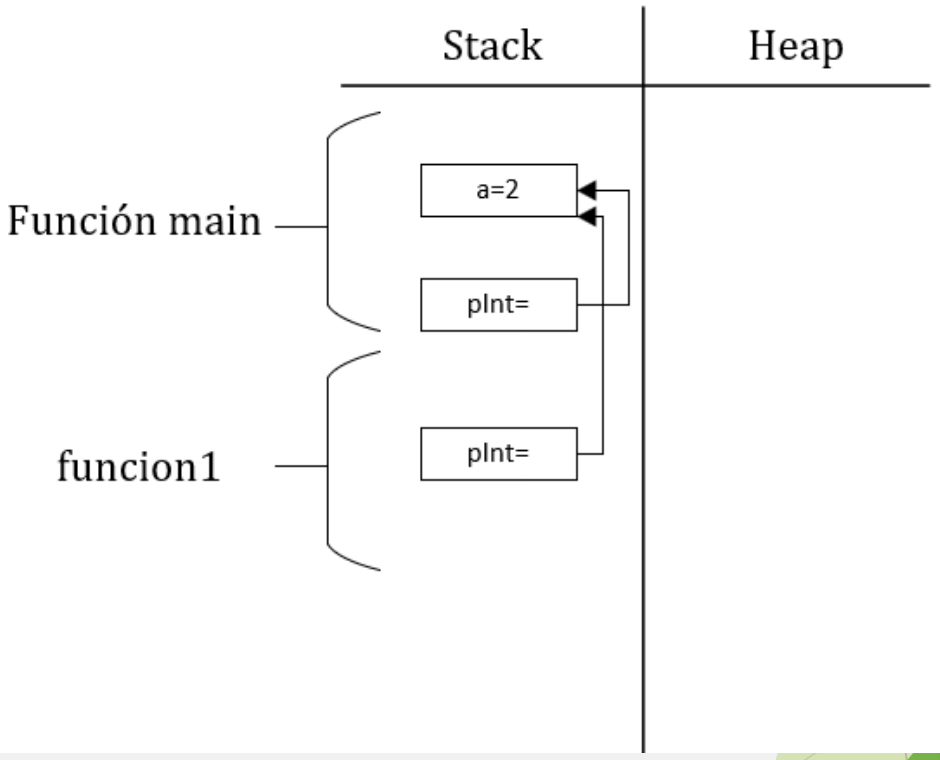
¿Sería correcto hacer cout<<*p?

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 **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)?

```
12
13 int funcion1(int * pInt) {
14     short a = 2;
15     return 0;
16 }
17
18 int main() {
19     short a = 2;
20     int * pInt = &a;
21     funcion1(pInt);
22     return 0;
23 }
24
```



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…

C++
Imprimir el contenido de la dirección
`cout << *static_cast<int*>(p)` `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 / alocación de memoria dinámica; a saber:

	C++
Reservar Memoria (Puntero a Tipo)	`pInt = new int;`
Liberar Memoria (Puntero a Tipo)	`delete pInt;`

Ahora bien, la memoria Heap no es infinita, con lo que podría no haber memoria para alocar. 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 alocación 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.

Ahora bien, la memoria Heap no es infinita, con lo que podría no haber memoria para alocar. 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 alocación 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 alocar 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 alocación 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 macro definida en el librería cstdlib (adaptación a C++ de la libreria 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.

C++

```
pInt  = NULL;
*pInt = 10;
// Error ya que pInt es nulo
```

2.10 Memoria Colgada

Supongamos el siguiente fragmento de código…

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…

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);
```

- `tipo` › es el tipo de dato devuelto por la función.
- `nombre_puntero` › 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 travez 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[])(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;
}
```

Vectores y vectores dinámicos

```
int lon = 4;
int vector[4];
sizeof(vector);
for(int i = 0; i< sizeof(vector); i++) {
    vector[i] = 0;
}

int * vector1 = new int[lon];
sizeof(vector1); //devuelve el tamaño del puntero
for(int i = 0; i < lon; i++) {
    vector1[i] = 0;
}
delete [] vector1;
```

Vectores con dimensión mayor

```
int ** vector2 = new int*[lon];
for(int i = 0; i < lon; i+
    vector2[i] = new int[l
    for(int j = 0; j < lon; j++) {
        vector2[i][j] = 0;
    }
}

vector2[3][3] = 1;
for(int i = 0; i < lon; i++) {
    delete [] vector2[i];
}
delete [] vector2;
```

```
int lon = 4;
Press 'F2' for focus
```

```
#include<iostream>
using namespace std;


typedef struct
{
 int legajo;
 char curso;
 int nota;
}alumno;

 int main()
 {
  alumno a, b;

  a.legajo=100;
  a.curso='c' ;
  a.nota=9;

  b.legajo=200;
  b.curso=a.curso;
  b.nota= a.nota -1;

 cout<<a.legajo <<"  "<<a.curso<< "  "<<a.nota<<endl;
 cout<<b.legajo <<"  "<<b.curso<< "  "<<b.nota<<endl;


  alumno *p;
   p=&a;
```

```
#include <iostream>
using namespace std;
typedef int* Pint;
typedef char* Pchar;
int main(){
Pint A, C, F;
Pint* B;
Pchar D, E;
char G = 'E';
int H = 66;
A = new int;
F = new int;
(*A) = 64;
B = &F;
(*F) = (*A);
cout << (**B) <<"  "<< (*A) << endl;
D = (Pchar)(*B);
E = (Pchar)A;
C = (Pint)D;
(*C) = H;
cout << (*D) <<"  "<< (*C) <<"  "<< (**B) << endl;
(*F) = (*C) + 3;
if ((*D) == G)
cout << (*E) <<"  "<< (*C) << endl;
(*E) = 'A';
(*F) = (*C) -(*A);
if (F == (*B))
cout << (*A) << "  "<<(*C) << endl;
delete A;
delete F;
return 0;
}
```

Indicar la salida

```
#include <iostream>
using namespace std;
typedef int* Pint;
typedef char* Pchar;

int main()
{
Pint A, C, F;
Pint* B;
Pchar D, E;
char G;
int H;
H = 67;
G = 'A';
A = new int;
(*A) = 64;
B = &C;
F = A;
C = &H;
cout << (**B) << "   "<<(*A) << endl;
D = (Pchar)(*B);
E = (Pchar)A;
cout << (*D) <<"   "<< (*C) << "   "<< (*E) << endl;
(*D) = G;
if ((*C) == 67)
cout << (*E) << "   "<<(*C) << endl;
(*A) = 66;
while ((*A) > 0) {
cout << (*A) << "    "<<(*E) << endl;
(*F)= (*F) -H;
(*A) = (*A) -1;
}
delete A;
return 0;
}
```

Indicar la salida

'@' es 64
'A' es 65

Fin