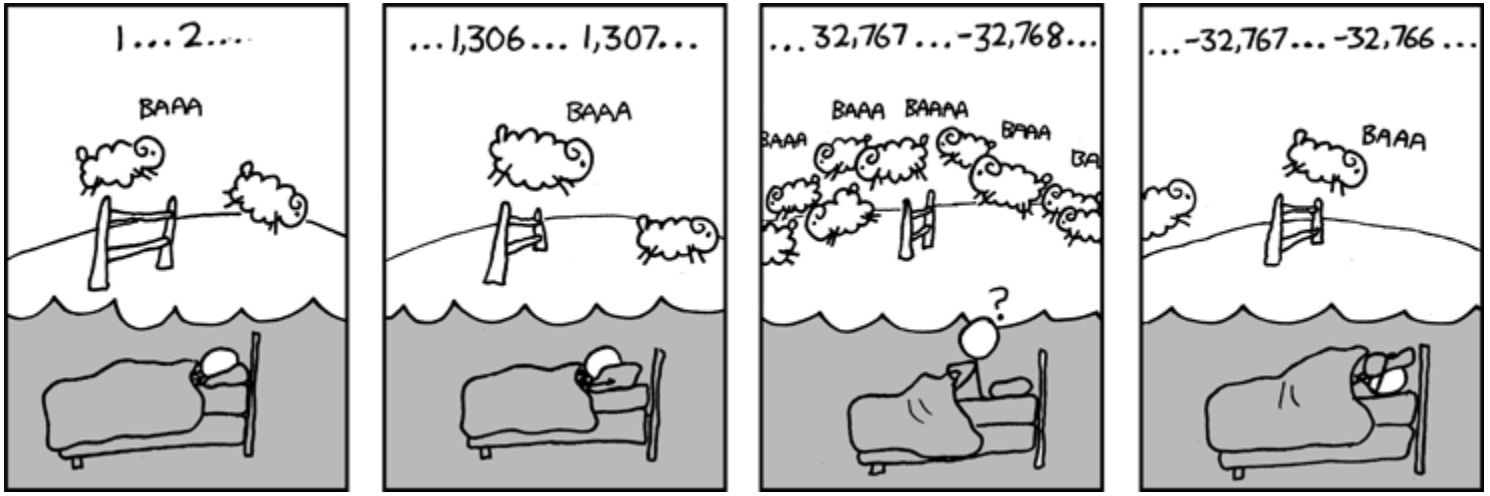
# “Introducción” al lenguaje C

Sistemas Operativos
Universidad de Antioquia
Laboratorio 1-3

# “Introducción” al lenguaje C




Fuente:
http://stackoverflow.hewgill.com/questions/845/56.html

# apuntadores

# Apuntadores

Un apuntador es una variable que contiene la dirección de una variable.

**MEMORIA**

| DIRECCIÓN | CONTENIDO |
|---|---|
| 0 | 10 |
| 4 | 11 |
| 8 | 12 |
| 12 | 13 |
| 16 | 4 |

- La variable **var1** está asociada a la dirección **4** de la memoria.
- Considere que **var1** es un entero de 4 bytes, por lo tanto **var1 = 11**.
- **pvar1** es apuntador a variables de tipo entero de 4 bytes.
- **pvar1** puede almacenar la dirección **var1.**
- **pvar1** está asociada a la posición de memoria **16** y las direcciones son de 4 bytes.
- Si **pvar1** apunta a **var1**, quiere decir que el contenido de la variable **pvar1** (la posición 16 de memoria) es **4**, ya que 4 es la dirección de **var1**.

# Apuntadores

¿Cómo se declara un apuntador? Se utiliza el operador *.

```
int var1 = 11;
int *pvar1;
```

**MEMORIA**

| DIRECCIÓN | CONTENIDO |
|---|---|
| 0 | 10 |
| 4 (var1) | 11 |
| 8 | 12 |
| 12 | 13 |
| 16 (pvar1) | 4 |

Quiere decir que **pvar1** es una variable que almacenará direcciones de variables de tipo int.

# Apuntadores

¿Cómo se obtiene la dirección de una variable? Se utiliza el operador &.

```
int var1 = 11;
int var2 = 12;
int *pvar1;
pvar1 = &var1;
```

**MEMORIA**

| DIRECCIÓN | CONTENIDO |
|---|---|
| 0 | 10 |
| 4 (var1) | 11 |
| 8 (var2) | 12 |
| 12 (pvar2) | 13 |
| 16 (pvar1) | 4 |

1. ¿Cómo obtener la dirección de var2 y almacenarla en pvar2?

2. ¿Cómo se almacena el contenido de pvar2 en pvar1?

# Apuntadores

- ¿Cómo leer y escribir el contenido de la dirección que está almacenada en el apuntador?

| DIRECCIÓN | CONTENIDO |
|---|---|
| 0 | 0 |
| 4 (x) | 1 |
| 8 (y) | 2 |
| 12 (px) | 4 |

```
int x= 1;
int y = 2;
int *px;

px = &x;
```

px almacena la dirección de x

| DIRECCIÓN | CONTENIDO |
|---|---|
| 0 | 0 |
| 4 (x) | 0 |
| 8 (y) | 0 |
| 12 (px) | 4 |

```
int x= 1;
int y = 2;
int *px;

px = &x;
*px = 0;
y = *px;
```

3

¿Cómo se podría almacenar en la posición de memoria **8** el valor almacenado en la posición de memoria **4** utilizando **px**?

# Apuntadores

- Incrementar o decrementar un apuntador

| DIRECCIÓN | CONTENIDO |
|---|---|
| 0 | |
| 4 (x) | 1 |
| 8 (y) | 2 |
| 12 (px) | 4 |

```
int x= 1;
int *px;
px = &x;
```

| DIRECCIÓN | CONTENIDO |
|---|---|
| 0 | |
| 4 (x) | 1 |
| 8 (y) | 5 |
| 12 (px) | 8 |

```
px = px +1;
*px = 5;
```

4

¿Qué ocurre en este caso?

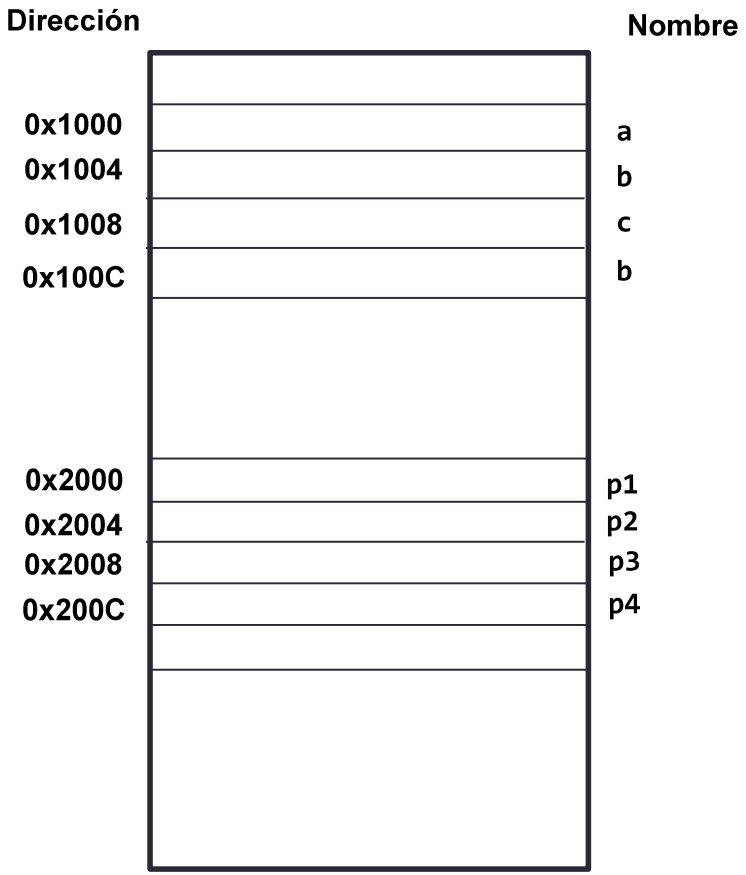
```
px = px - 1;
*px = 5;
```

# Apuntadores

**5**

Suponga que se tienen las siguientes instrucciones. Cuando se realizo la declaración de las variables se asignaron las direcciones para cada variable tal y como se muestra en la figura. ¿Cuáles son los valores finales de las variables después de la ejecución de las instrucciones?

```
// instrucciones
int a,b=3,c=8, d;
int *p1 = &a, *p2, *p3 = &c;
*p1 = 2;
p2 = p3;
*p2 = *p1-b;
d = (*p2)*(*p1);
p3 = &d;
b = a + b + c;
```



# Apuntadores

```c
#include <stdio.h>

void swap(int d1, int d2);

int x;
int y;

int main(void)
{
   x = 1;
   y = 2;
   printf("Valor de x: %d\n",x);
   printf("Valor de y: %d\n",y);
   swap(x,y);
   printf("Valor de x: %d\n",x);
   printf("Valor de y: %d\n",y);
   return 0;
}


void swap(int d1, int d2)
{
   int temp;
   temp = d1;
   d1 = d2;
   d2 = temp;
}
```

Queremos hacer una función que intercambie el contenido de dos variables. Al ejecutar el programa anterior este es el resultado:

```
juanfh@franco:~/Cexamples/Lab2Examples$ ./Lab2-1
Valor de x: 1
Valor de y: 2
Valor de x: 1
Valor de y: 2
juanfh@franco:~/Cexamples/Lab2Examples$
```

¿Qué salió mal?

En C los parámetros de una función se pasan por valor. Al llamar swap(x,y), se pasan en el **stack** los números 1 y 2.



¿Qué es el **stack**?

# Apuntadores

```c
#include <stdio.h>

void swap(int *d1, int *d2);

int x;
int y;

int main(void)
{
    x = 1;
    y = 2;
    printf("Valor de x: %d\n",x);
    printf("Valor de y: %d\n",y);
    swap(&x,&y);
    printf("Valor de x: %d\n",x);
    printf("Valor de y: %d\n",y);
    return 0;
}


void swap(int *d1, int *d2)
{
    int temp;
    temp = *d1;
    *d1 = *d2;
    *d2 = temp;
}

```

Al ejecutar el programa este es el resultado:

```
juanfh@franco:~/Cexamples/Lab2Examples$ ./Lab2-2
Valor de x: 1
Valor de y: 2
Valor de x: 2
Valor de y: 1
juanfh@franco:~/Cexamples/Lab2Examples$
```

El programa funciona correctamente. En este caso estamos pasando a **swap** la dirección de las variables x, y. A esto se le conoce como paso de parámetros por **referencia**.

# Apuntadores

```
1 #include <stdio.h>
2
3 void swap(int *d1, int *d2);
4
5 int x;
6 int y;
7
8 int main(void)
9 {
10    x = 1;
11    y = 2;
12    printf("Valor de x: %d\n",x);
13    printf("Valor de y: %d\n",y);
14    swap(&x,&y);
15    printf("Valor de x: %d\n",x);
16    printf("Valor de y: %d\n",y);
17    return 0;
18 }
19
20
21 void swap(int *d1, int *d2)
22 {
23    int temp;
24    temp = *d1;
25    *d1 = *d2;
26    *d2 = temp;
27 }
28
```

**7**

Analice y compare los códigos en los cuales los llamados a funciones fueron hechos por valor y por referencia y responda las siguientes preguntas:

- ¿Por qué cuando se hace el llamado por referencia en la función se pasan los parámetros con &?
- Suponga que en la línea 12 (del programa de la izquierda)se introduce la siguiente instrucción **int *p = &y;** la invocación (resaltada en el cuadro azul) se puede reemplazar por:

**a.swap(x,*p);**
**b.swap(x,&p);**
**c.swap(x,p);**
**d.Ninguna de las anteriores.**

```
1 #include <stdio.h>
2
3 void swap(int d1, int d2);
4
5 int x;
6 int y;
7
8 int main(void)
9 {
10    x = 1;
11    y = 2;
12    printf("Valor de x: %d\n",x);
13    printf("Valor de y: %d\n",y);
14    swap(x,y);
15    printf("Valor de x: %d\n",x);
16    printf("Valor de y: %d\n",y);
17    return 0;
18 }
19
20
21 void swap(int d1, int d2)
22 {
23    int temp;
24    temp = d1;
25    d1 = d2;
26    d2 = temp;
27 }
28
```

Declaración (prototipo)

Invocación (Llamado de la función)

Definición

# Apuntadores

8

•Compile el código mostrado a continuación. Si este presenta errores corríjalos. Explique brevemente por que ocurren estos errores.
•Ejecute el programa, observe la salida y diga el por que se observa esta.

```
#include <stdio.h>

void f1(int ,int *, int *);

int main() {
  int d1 = 3, d2 = 1, d3 = 4;
  printf("d1=%d, d2=%d, d3=%d\n",d1,d2,d3);
  f1(--d2,d3,d1);
  printf("d1=%d, d2=%d, d3=%d\n",d1,d2,d3);
  return 0;
}

void f1(int x,int *y, int *z) {
  *y = x + z;
  *z = --(*y);
  x = *y - *z;
}
```

# arreglos

# Arreglos

- Permiten construir vectores cuyos elementos son del mismo tipo de dato.

- Ejemplo: **int vec[4] = {1,2,3,4};**
- Seclara un arreglo de 4 elementos de tipo entero. Si se asume que cada int ocupa 4 bytes en memoria y que **vec** comienza en la dirección 0, se tiene que:

| DIRECCIÓN | CONTENIDO |
|---|---|
| 0 (vec[0]) | 1 |
| 4 (vec[1]) | 2 |
| 8 (vec[2]) | 3 |
| 12 (vec[3]) | 4 |

Considere:

int a = vec[0] o también (a = 1)
int a = *vec (a = 1)

# Arreglos

| DIRECCIÓN | CONTENIDO |
|---|---|
| 0 (vec[0]) | 1 |
| 4 (vec[1]) | 20 |
| 8 (vec[2]) | 30 |
| 12 (vec[3]) | 40 |

```
int a = *(vec + 1);
Int b = *vec + 1;
```

9

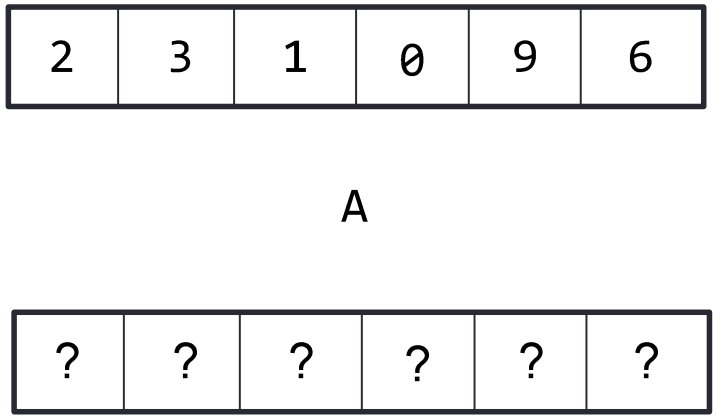
En el programa anterior, ¿Cuál es el valor de las variables a y b?

# Arreglos

**10**

¿Cómo quedará el arreglo después de que se ejecutan las siguientes instrucciones?

```
...
int A[6]={2,3,1,0,9,6};
int *ptr1;
int *ptr2 = &A[5];
ptr1 = A;
ptr1+=2;
*ptr1=5;
ptr2--;
*(ptr2)=*ptr1+*(ptr2-1)+*(ptr2+1);
...
```



# Alimento para el pensamiento 1

11

**Realice un programa que calcule el promedio de lo valores de un arreglo de 100 posiciones.**

**Nota: El programa debe generar el arreglo automáticamente y luego llamar una función que calcule el promedio.**

# Arreglos multidimensionales

- Un arreglo multidimensional se puede entender como una arreglo de una dimensión cuyos elementos son arreglos.

- ¿Cómo se almacena el arreglo “nombres” en memoria si los char ocupan 1 byte?

- char nombres[3][10] = {"fulano", "mengano", "perano"};

# Arreglos multidimensionales

| Dirección | Contenido |
|---|---|
| 0 | ‘f’ |
| 1 | ‘u’ |
| 2 | ‘l’ |
| 3 | ‘a’ |
| 4 | ‘n’ |
| 5 | ‘o’ |
| 6 | 0 |
| 7 | 0 |
| 8 | 0 |
| 9 | 0 |

| Dirección | Contenido |
|---|---|
| 10 | ‘m’ |
| 11 | ‘e’ |
| 12 | ‘n |
| 13 | ‘g’ |
| 14 | ‘a’ |
| 15 | ‘n’ |
| 16 | ‘o’ |
| 17 | 0 |
| 18 | 0 |
| 19 | 0 |

| Dirección | Contenido |
|---|---|
| 20 | ‘p’ |
| 21 | ‘e’ |
| 22 | ‘r’ |
| 23 | ‘a’ |
| 24 | ‘n’ |
| 25 | ‘o’ |
| 26 | 0 |
| 27 | 0 |
| 28 | 0 |
| 29 | 0 |

# Alimento para el pensamiento 2

12

**Explique cómo funciona el siguiente programa y que resultado produce.**

```c
#include <stdio.h>

char nombres[3][10] = {"fulano","mengano", "perano"};

int main(void)
{
    char i;
    char *a;
    char (*b)[10];

    a = (char *)nombres;
    printf("el nombre es %s \n",a);

    b = (char (*)[10])nombres[0];
    for(i = 0;i < 3; i++)
    {
        printf("el nombre[%d] es %s \n",i,(char *)(b+i));
    }
    return 0;
}
```

# Memoria dinámica

# Asignación dinámica de memoria

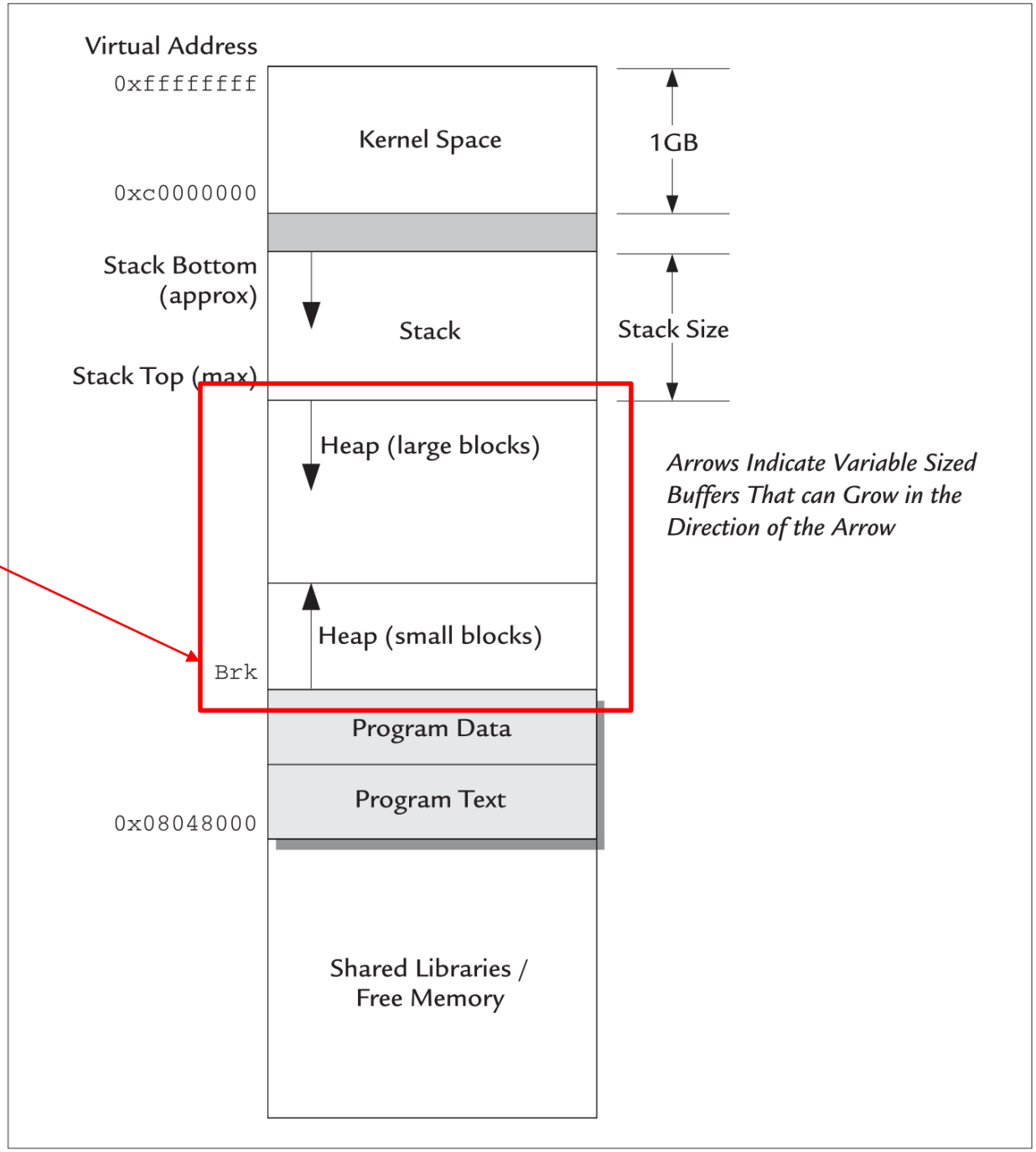
- En **lenguaje C** las variables se puede asignar en memoria de tres formas: **estáticamente**, **automáticamente** (en el stack), **dinámicamente** (en el heap) (algunas se almacenan también en registros del procesador).

- El tamaño de las variables asignadas **estáticamente** y **automáticamente** se debe conocer en tiempo de compilación (antes del estándar C99).

- Si el tamaño de la variable únicamente puede ser conocido en **tiempo de ejecución**, la variable debe asignarse de manera **dinámica** en el heap.

Mapa de memoria virtual de un proceso corriendo en una arquitectura IA32 con kernel Linux compilado 3G/1G.

Para declarar variables en el heap se utilizan las funciones **malloc()** y **free()**.



# Malloc

**Permite reservar un bloque de memoria en el heap.**

void **malloc*(size_t size );

- **malloc** retorna la dirección en el heap a partir de la cual se reservó el tamaño de memoria solicitado, o retorna **NULL** si no es posible reservar la cantidad de memoria.

- **void** * indica que la dirección retornada es genérica, es decir, en esa dirección se puede almacenar cualquier tipo de variable.

- **malloc** reservará **size** bytes de memoria siempre que estén disponibles.

# Free

**Permite liberar un bloque de memoria previamente reservado en el heap.**

void *free*(void *pointer);

•**free** recibe la dirección del bloque de memoria a liberar apuntado por pointer.

# Alimento para el pensamiento 3

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

int main(void)
{
  char *buffer;
  char *string1 = "Hola ";
  buffer = malloc(sizeof(char)*30);
  strcpy(buffer, string1);
  strcat(buffer, "Mundo.");
  puts(buffer);
  free(buffer);
  return 0;
}

```

13

**¿Qué hace la función sizeof()?**

14

**Explique cómo funciona el programa y que resultado produce.**

# Alimento para el pensamiento 1

15

**Realice un programa que calcule el promedio de lo valores de un arreglo de n posiciones.**

**Nota: Requisitos del programa**
**-Debe crear el arreglo dinamicamente (malloc) acorde al numero de posiciones**
**-Debe ingresar el numero de posiciones como argumento, asi: ./program 50  (para 50 posiciones)**
**-Debe generar el arreglo automáticamente y luego llamar una función que calcule el promedio.**

# Ejercicio práctico

# Definición




**Realice un programa que permita calcular el promedio ponderado obtenido por un estudiante en el semestre, considerando:**

1.El programa debe solicitar el número de materias vistas en el semestre.
2.Debe preguntar el nombre de la materia.
3.Debe preguntar la nota obtenida.
4.Debe solicitar el número de créditos.
5.Al final, debe mostrar una tabla con tantas filas como materias y con 3 columnas indicando el nombre de la materia, la nota y el número de créditos.
6.También debe reportar el promedio ponderado.

**Nota: use los conceptos aprendidos anteriormente.**