# 4. APUNTADORES Y FUNCIONES

## 4.1. DEFINICIÓN DE APUNTADOR

Un apuntador es una variable que es capaz de almacenar una dirección de memoria y no un valor específico. Ésta definición como tal es muy simple pero la manipulación de un apuntador ya no lo es tanto.

Los apuntadores se clasifican en apuntadores con tipo y sin tipo. Un apuntador con tipo es aquel que "apunta" a un tipo de dato que se determina a priori, por ejemplo un apuntador a enteros o un apuntador a caracteres o bien un apuntador a cadenas. Un apuntador sin tipo "apunta" a una determinada dirección de memoria en donde allí puede estar almacenado el valor que sea (es decir, no se especifica de antemano como en el caso de los apuntadores tipados, qué clase de datos se almacenarán allí).
En C, la declaración de una variable de tipo apuntador se define de la siguiente manera:

```
<tipo>    *<nombreApuntador>;
```

Si el <tipo> es *void* entonces se trata de un apuntador sin tipo (genérico), sino entonces se trata de un apuntador con tipo (tipado). En el siguiente código se declara un apuntador a enteros y se hace que apunte a la variable *numero*. De esta manera podemos acceder al dato almacenado en *numero* a través de dicha variable o a través del apuntador lo cual es equivalente:

```
#include <stdio.h>

int main()
   {
    int numero;
    int *p; //apuntador a enteros, es decir, apuntador tipado

    numero=5;
    printf("%\nANTES DE USAR EL APUNTADOR, numero=%d\n",numero);

    p=&numero;
    *p=10;
    printf("%\nDespues de usar el apuntador, numero=%d\n\n",numero);
    printf("\nEsta es la direccion de memoria a donde apunta
              p:%x\n\n",p);
    printf("\nEsta es la direccion de memoria en donde esta almacenado
              p:%x\n\n",&p);
    system("pause");
    return 1;
   }
```

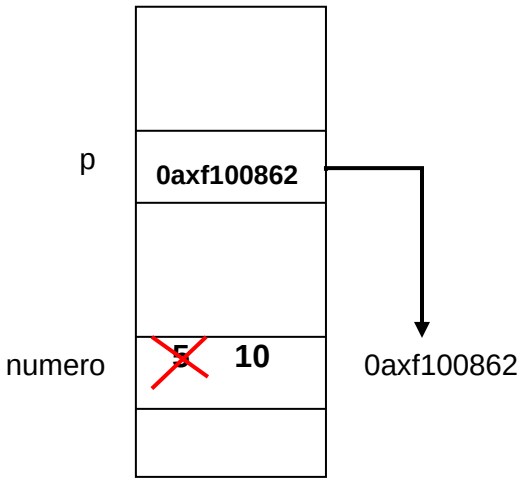
**p=№** Se obtiene la dirección de memoria de la variable numero y se almacena en la variable p que es un apuntador a enteros.

***p=10;** Indirección del apuntador p, es decir, se accede a la dirección de memoria donde apunta p que es aquella donde esta almacenada la variable numero que tiene el numero 5 y en ella se almacena el valor de 10 (sustituye al 5).

**p;** El usar sólo p implica trabajar con la dirección de memoria a donde esta apuntando.

**&p;** Representa la dirección de memoria donde se encuentra almacenado el apuntador p.

La siguiente figura muestra el comportamiento del programa anterior a nivel memoria de la computadora:




Es importante entender la diferencia que existe entre las siguientes instrucciones:

a) ***p=10;** En esta instrucción se asigna el valor de 10 en la localidad de memoria apuntada por el apuntador p.
b) **p=10;** Aquí, se sobrescribe el contenido del apuntador p, es decir, ahora quedaría apuntando a la dirección 10 de memoria. Asignar un valor absoluto a un apuntador es motivo de error de programación en la mayoría de los casos!!!...

### 4.1.1. Manejo de memoria dinámica

Los apuntadores vistos en la sección anterior apuntan a variables estáticas, es decir, el apuntador *p* apunta a la variable estática *numero* (es estática porque en el momento de la compilación el compilador le asigna a la variable *numero* la cantidad de casillas de memoria necesarias para su almacenamiento y éstas son reservadas para dicha variable y

nadie más las puede utilizar en toda la ejecución del programa). Sin embargo la potencialidad del uso de los apuntadores radica en el uso y manejo de memoria dinámica.

La *memoria dinámica* es aquella que el programa solicita y libera durante su ejecución en función de la demanda (es decir, cuando sea necesario hacerlo). En C tanto la petición como liberación de memoria se lleva a cabo mediante las siguientes funciones que se encuentran dentro del archivo de encabezado **<stdlib.h>**:

`void *malloc(size_t size);` Recibe el tamaño en bytes del espacio de memoria que se requiere y devuelve un apuntador a void al primer byte de ese bloque de memoria o bien **NULL** sino encuentra la cantidad de memoria contigua solicitada.

`void free(void *ptr);` Recibe un apuntador y libera el bloque de memoria dinámica al que apunta.

A continuación se muestra un ejemplo del uso de las funciones *malloc y free* para usar memoria dinámica y almacenar en ella un valor entero ( y entonces poder acceder a esta porción de memoria dinámica mediante un apuntador).

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

int main()
   {
    int numero;
    int *p;   //apuntador (con tipo) a enteros

    p=malloc(sizeof(int)); //Asignacion de memoria dinamica

    if(p==NULL)
       {
        printf("\nNo se pudo asignar memoria al apuntador!\n");
        return 1;
       }

    *p=50; // Indireccion del apuntador y asignacion de un valor
    printf("\nEl apuntador apunta al valor %d\n", *p);

    free(p); //liberación de la memoria dinamica

    system("pause");
    return 1;
}
```

Un detalle importante es que *malloc* reserva una cantidad arbitraria de bytes, es decir, si se solicitan 100 bytes, esta porción de memoria puede ser utilizada para almacenar por ejemplo 25 enteros de 4 bytes cada uno o bien 100 caracteres, cada uno de un byte o posiblemente 25 valores de punto flotante. Dicho de otra manera, la porción de memoria dinámica asignada por *malloc* es una Proción "genérica" capaz de almacenar tantos datos como quepan en ella.

### 4.1.2. Operaciones básicas con apuntadores y arreglos

#### 4.1.2.1. Manejo de arreglos a través de apuntadores

Parte muy importante del uso de memoria dinámica es que como *malloc* puede reservar una cantidad arbitraria de bytes que es genérica para almacenar cualquier tipo de datos que quepan en ella, podemos entonces construir arreglos dinámicos. En el siguiente código se crea un arreglo dinámico de enteros que se inicializa y se muestra en pantalla.

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

#define TAM_ARRAY 10
#define LIM_INF  50
#define LIM_SUP  70

int main()
   {
    int *numeros,i; //Se declara un apuntador a enteros

    numeros=malloc(TAM_ARRAY*sizeof(int));
    //Asignacion de memoria dinamica

    if(numeros==NULL)
       {
        printf("\nNo se pudo asignar memoria al apuntador!\n");
        return 1;
       }

    srand(time(NULL));
    for(i=0;i<TAM_ARRAY;i++)
       numeros[i]=rand()%(LIM_SUP-LIM_INF+1)+LIM_INF;
       // generacion de num aleatorios entre LIM_INF y LIM_SUP

    for(i=0;i<TAM_ARRAY;i++)
       printf("%d, ",numeros[i]);
    printf("\n");

    free(numeros); //liberación de la memoria dinamica

    system("pause");
```

```
    return 1;
}
```

Un punto importante en el código anterior es que para trabajar un arreglo de enteros sólo necesitamos un apuntador a enteros. Esto es así porque no importa el tamaño que tenga el arreglo, la manera en que accedemos a él siempre es a partir del primer elemento de éste para poder acceder a los elementos que siguen.

#### 4.1.2.2. Aritmética de apuntadores

Aun cuando un apuntador represente una dirección de memoria, finalmente los apuntadores a nivel de lenguaje de programación son valores enteros. Es por eso que es posible realizar sobre ellos operaciones aritméticas de suma o resta.

Dado un apuntado *p* que apunta a datos de tipo *T*, sumar una unidad a *p* significa que el apuntador se incrementará en el tamaño del tipo *T*. Es decir, si *P* es un apuntador a enteros de 4 bytes, incrementarlo en una unidad significa que la dirección almacenada en *p* se incremente en 4. De esta manera el uso de apuntadores como índices de arreglos es fácil pues si el apuntador *p* apunta al elemento *i* de un arreglo de enteros, la operación *p+1* incrementa la dirección de memoria de donde aunta *p,* 4 bytes y de esta manera se consigue que *p* apunte al elemento *i+1* del arreglo.

El siguiente código muestra el uso de la aritmética de apuntadores para recorrer secuencialmente una cadena de caracteres.

```
#include <stdio.h>

int main()
  {
   char *p="Hola Mundo!";

   for(p; *p!='\0'; p++)
      printf("%c",*p);
   printf("\n");

   system("pause");
   return 1;
  }
```

Algo interesante es que el nombre de un arreglo es también la dirección de memoria del primer elemento de éste, por tanto el nombre de un arreglo puede utilizarse como un apuntador:

```
#include <stdio.h>

int main()
{
 char cadena[]="Hola Mundo!";
 int i;

 for(i=0; *(cadena+i)!='\0'; i++)
    printf("%c",*(cadena+i));
 printf("\n");

 system("pause");
 return 1;
}
```

### 4.1.3. Apuntadores no tipados (sin tipo)

Un apuntador no tipado o sin tipo es un apuntador declarado con *void* el cual se convierte en un apuntador genérico capaz de apuntar a cualquier tipo de dato. Su sintaxis es:

```
void  *<nomApun>
```

El ejemplo siguiente muestra un apuntador genérico que se utiliza para almacenar una cadena de caracteres y después almacena un número entero.

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

#define TAM_CADENA 20

int main()
   {
    void *p; //apuntador genérico o no tipado

    p=malloc(TAM_CADENA*sizeof(char));
    // Asig. de mem dinamica para una cadena

    if(p==NULL)
       {
        printf("\nNo se pudo asignar memoria al apuntador...\n");
        return 1;
       }
    strcpy(p,"Hola Mundo!");
    printf("Valor apuntado por el apuntador: %s\n", (char *)p);
    free(p);
```

```
    p=malloc(sizeof(int)); // Asig. de mem dinamica para una cadena
    if(p==NULL)
       {
        printf("\nNo se pudo asignar memoria al apuntador...\n");
        return 1;
       }
    *(int *)p=123;
    printf("Valor apuntado por el apuntador: %d\n", *(int *)p);
    free(p);

    system("pause");
    return 1;
   }
```

## 4.2. FUNCIONES

En el lenguaje C, el código fuente de un programa se puede modularizar (dividir en módulos de código de programa que resuelven subproblemas del problema general) a través del uso de funciones, que cooperarán entre sí para dar al usuario final la solución de un problema. La sintaxis de una función en C es la siguiente:

```
<tipo_de_retorno> <nombreFuncion>(<lista_parametros>)
   {
     <cuerpo_de_la_funcion>
   }
```

Por ejemplo, el siguiente código muestra una función que recibe como parámetros dos números enteros y retorna como resultado la suma de ambos números:

```
int suma(int a, int b)
   {
     int resultado;
     resultado=a+b;
    return resultado;
   }
```

Una función siempre retorna un solo valor, salvo en los casos en los cuales no regresa ninguno; en este caso, la función se declara de tipo de retorno *void*.

Dependiendo de la forma de programar, es usual que las funciones se definan después de la función principal *main()*, en cuyo caso será necesario para cada una de ellas, escribir su prototipo o firma antes del *main()* tal como lo muestra el siguiente código:

```
#include <stdio.h>

int suma(int a, int b); //Prototipo o firma de la funcion suma

int main()
   {
    int operando1, operando2, resultado;

    operando1=5;
    operando2=10;

    /* llamado de la funcion suma. Se le invoca con su nombre
       y sus argumentos entre paréntesis */
    resultado= suma(operando1,operando2);

    printf("\n%d + %d = %d\n",operando1,operando2,resultado);
    system("pause");
    return 1;
   }

int suma(int a, int b)
   {
    int resultado;
    resultado=a+b;
    return resultado;
   }
```

Una función puede ser llamada desde cualquier parte del programa tantas veces como sea necesaria, una vez que ésta se haya definido. Se le invoca con su nombre, seguida de una lista opcional de argumentos (tantos como parámetros tenga la función). Los argumentos van entre paréntesis y si hubiera más de uno entonces van separados por comas. Una función puede no requerir de argumento alguno, será entonces porque no lleva parámetros, en cuyo caso sólo existirán los paréntesis vacíos.

### 4.2.1. Ámbito de las declaraciones de las variables

A las declaraciones de variables y constantes dentro del cuerpo de una función (las cuales se pueden utilizar dentro de cualquier punto de ése bloque) se les denomina *variables locales*, pues su existencia y alcance están limitados por la función que las contiene, es decir, sólo existen para dicha función y fuera de ella no se pueden utilizar. Por otro lado, cuando las declaraciones se realizan fuera de cualquier función, se les considera variables y/o constantes globales y su alcance es el de todas las funciones del programa, es decir, pueden ser utilizadas por cualquier función pues son variables y/o constantes compartidas. A continuación se muestra el código del ejemplo anterior pero ahora utilizando variables globales:

```
#include <stdio.h>

int suma(); //Prototipo o firma de la funcion suma

int operando1, operando2, resultado; //Variables globales

int main()
   {
    operando1=5;
    operando2=10;
    resultado= suma(); //Llamado a la funcion suma sin aparametros

    printf("\n%d + %d = %d\n",operando1,operando2,resultado);
    system("pause");
    return 1;
   }

int suma()
   {
    int resultado; //variable local de la función suma
    resultado=operando1+operando2;
    return resultado;
   }
```

Ahora otro ejemplo pero ahora del uso de variables locales:

```
#include <stdio.h>

int obten_Maximo(int, int, int);
void imprime_numeros(int, int, int);

int main()
  {
   /* variables locales de main() */
   int n1,n2,n3, maximo;

   printf("\nIngrese tres numeros enteros: ");
   scanf("%d %d %d", &n1, &n2, &n3);

   imprime_numeros(n1,n2,n3);
   maximo=obten_Maximo(n1,n2,n3);
   printf("El maximo es: %d\n", maximo);


   printf("\n");
   system("pause");
   return 1;
  }
```

```
int obten_Maximo(int a, int b, int c)
   {
    int mayor; //variable local de obten_Maximo(...)

    mayor=a;
    if (b>mayor)
        mayor=b;
    if (c>mayor)
        mayor=c;
    return mayor;
   }

void imprime_numeros(int a, int b, int c)
   {
    printf("Numeros ingresados: %d %d %d \n",a,b,c);
   }
```

### 4.2.2. Paso de parámetros

Como ya se ha podido observar, un parámetro no es lo mismo que un argumento. El parámetro es el nombre de la variable que utiliza la función en forma interna para hacer uso de cada valor que le está pasando el que la llamó. El argumento es el valor en sí mismo. Cuando se llama a una función los argumentos deben ser coherentes en cantidad, tipo y orden respecto de los parámetros especificados en la declaración de la función.

Por tanto el paso de parámetros es la copia de los argumentos de una función a una zona de memoria de la propia función o stack. Existen diferentes formas de llevar a cabo el paso de parámetros en la llamada a una función:

**Paso de parámetros por valor:** Los argumentos son copiados en las variables que conforman la lista de parámetros de la función. De esta manera, si la función modifica el contenido de sus parámetros, el argumento original no es alterado.

**Paso de parámetros por referencia:** En lugar de una copia se pasa la dirección de memoria de un argumento (es decir, una referencia), de manera que la función podría modificar el valor original. Esto puede ser útil si se quiere que la función retorne más de un valor a través de sus parámetros.

En el siguiente ejemplo, se muestra el uso de una función utilizando el paso de parámetros por referencia para intercambiar el contenido de dos variables:

```
#include <stdio.h>

void intercambiar(char *a, char *b);

int main()
  {
   char letra1, letra2;

   letra1='A';
   letra2='B';

   printf("%c %c \n",letra1,letra2);
   intercambiar(&letra1, &letra2);
   printf("%c %c \n",letra1,letra2);

   printf("\n");
   system("pause");
   return 1;
  }

void intercambiar(char *a, char *b)
   {
    char temp;
    temp=*a;
    *a=*b;
    *b=temp;
   }
```

### 4.2.3. Argumentos en línea de comandos

De la misma manera en que una función puede recibir argumentos como parámetros, la ejecución de un programa también puede recibir datos externos en forma de argumentos (a través del *main()*) pasándoselos en línea de comandos.

En C, el paso de parámetros por línea de comandos se lleva a cabo indicando una lista de parámetros a la función *main()*, que es el programa principal, es decir, es la función que siempre se ejecutará primero en un programa. Su sintaxis es:

```
int main(int argc, char *argv[])
  {
   . . .
   return 1;
  }
```

***argc*** indica la cantidad de argumentos que recibe el programa, considerando el nombre del programa como el primero de ellos.

***argv*** es una arreglo de cadenas de caracteres que son los argumentos en sí que se pasarán en línea de comandos.

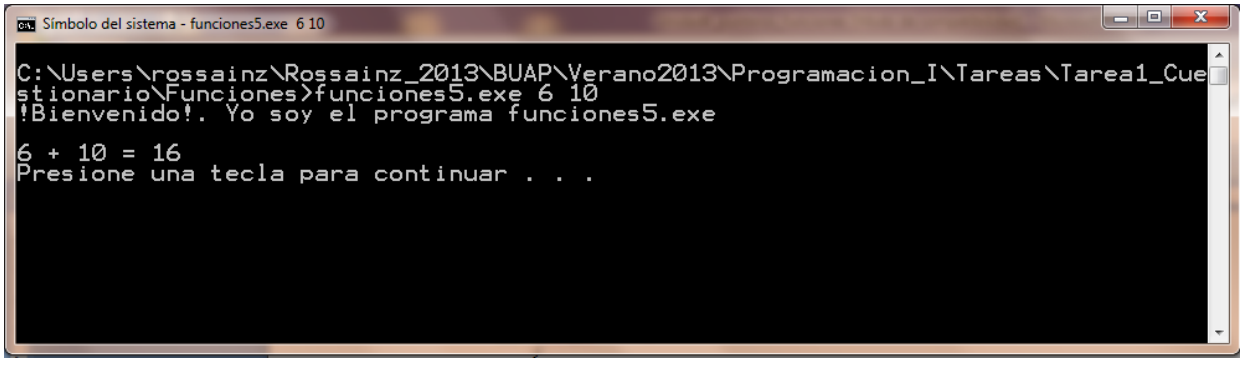
El siguiente ejemplo muestra un programa que recibe dos enteros en línea de comandos como argumentos del programa llamado funciones5.exe, los suma y muestra su resultado.

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

int main(int argc, char *argv[])
   {
    int num1, num2;

    if (argc !=3)
       {
        printf("Modo de uso: %s <entero> <entero>\n",argv[0]);
        system("pause");
        return 1;
       }
    printf("!Bienvenido!. Yo soy el programa %s\n\n", argv[0]);
    num1=atoi(argv[1]);
    num2=atoi(argv[2]);

    printf("%d + %d = %d\n",num1,num2,num1+num2);
    system("pause");
    return 1;
   }
```




### 4.2.4. Arreglos como parámetros de funciones

El lenguaje C también soporta el paso de arreglos como parámetros. Se debe recordar que el nombre de un arreglo y la dirección de memoria del primer elemento de éste son equivalentes, por tanto, en un paso de parámetros con arreglos, lo que se pasa es la dirección de memoria del primer elemento del arreglo (es decir, un apuntador) y no el contenido del arreglo. Lo que estamos haciendo entonces es un *paso de parámetros por*

*referencia* lo que permite que la función que recibe como argumento el arreglo altere el contenido de éste y dichos cambios se perciban en el ámbito de quien llamó a la función. A continuación un ejemplo:

```
#include <stdio.h>
#define N 5

void cargar_arreglo(int numeros[], int n);
float promediar(int numeros[], int n);

int main()
{
 int numeros[N];
 cargar_arreglo(numeros,N);
 printf("Promedio = %.2f\n", promediar(numeros,N));

 printf("\n");
 system("pause");
 return 1;
}

void cargar_arreglo(int numeros[], int n)
   {
    int i;
    for(i=0;i<n;i++)
      {
       printf("Ingrese un entero: ");
       scanf("%i", &numeros[i]);
      }
   }

float promediar(int numeros[], int n)
   {
    int i, sumatoria=0;
    for(i=0;i<n;i++)
       sumatoria+=numeros[i];
    return (float)sumatoria/n;
   }
```

Nótese en el ejemplo que no se especifica el tamaño del arreglo en el listado de parámetros de cada función, sin embargo, sí es necesario indicar a la función la longitud del arreglo que se está pasando por medio de un parámetro ya que de otra forma, la función no tendría medios para determinar cuáles son los límites del arreglo.

De acuerdo con esto, las funciones del código anterior podrían sobreescribirse como se muestran a continuación:

```
void cargar_arreglo(int *numeros, int n)
   {
    . . .
   }


float promediar(int *numeros, int n)
   {
    . . .
   }
```

Por extensión se puede deducir que el paso de un arreglo multidimensional como parámetro de una función se lleva a cabo tal como lo muestra el ejemplo siguiente.

```
#include <stdio.h>

#define M 5
#define N 5

void cargar_matriz(char matriz[][N], int filas, int columnas);
void imprime_matriz(char matriz[][N], int filas, int columnas);


int main()
{
 char matriz_letras[M][N];

 cargar_matriz(matriz_letras,M,N);
 imprime_matriz(matriz_letras,M,N);

 printf("\n");
 system("pause");
 return 1;
}

void cargar_matriz(char matriz[][N], int filas, int columnas)
{
 int i,j;

 for(i=0; i<filas; i++)
    {
     for(j=0; j<columnas;j++)
        {
         printf("\nIngrese el caracter [%d][%d]: ",i,j);
         scanf("%c", &matriz[i][j]);
         while(getchar()!='\n');
        }
    }
}
```

```
void imprime_matriz(char matriz[][N], int filas, int columnas)
{
 int i,j;

 for(i=0; i<filas; i++)
    {
     for(j=0; j<columnas;j++)
        {
         printf("%c\t",matriz[i][j]);
        }
     printf("\n");
    }
}
```

### 4.2.5. Registros como parámetros de funciones

Los registros también pueden pasarse como argumentos de funciones. Este paso de parámetros se realiza por copia, aunque también puede utilizarse la dirección de memoria o apuntador donde se aloja la estructura si fuese necesario para modificar alguno de sus campos dentro de la propia función.

El siguiente ejemplo muestra cómo se implementa el paso de una estructura a una función tanto por copia como por referencia.

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

typedef struct
{
 char nombre[30];
 int edad;
 long mat;
 int nrc;
 char nombre_carrera[20];
} t_alumno;

void cargar_alumno(t_alumno *alumno);
void mostrar_alumno(t_alumno alumno);

int main()
{
 t_alumno alumno;

 cargar_alumno(&alumno);
 mostrar_alumno(alumno);
```

```
 printf("\n");
 system("pause");
 return 1;
}

//paso de la struct por referencia
void cargar_alumno(t_alumno *alumno)
   {
    printf("Ingrese los datos del alumno\n\n");
    printf("Nombre: ");
    fgets(alumno->nombre,30,stdin);
    if (alumno->nombre[strlen(alumno->nombre)-1]=='\n')
       alumno->nombre[strlen(alumno->nombre)-1]='\0';
    else while(getchar()!='\n');

    printf("Edad: ");
    scanf("%d", &(alumno->edad)); //equivale a (*alumno).edad
    while(getchar()!='\n');

    printf("Matricula: ");
    scanf("%ld", &(alumno->mat));
    while(getchar()!='\n');

    printf("NRC de la carrera: ");
    scanf("%d", &(alumno->nrc));
    while(getchar()!='\n');

    printf("Nombre de la carrera: ");
    fgets(alumno->nombre_carrera,20,stdin);
    if (alumno->nombre_carrera[strlen(alumno->nombre_carrera)-1]=='\n')
       alumno->nombre_carrera[strlen(alumno->nombre_carrera)-1]='\0';
    else while(getchar()!='\n');
   }

//paso de la struct por valor
void mostrar_alumno(t_alumno alumno)
   {
    printf("\nDATOS DEL ALUMNO:\n\n");
    printf("Nombre %s\n",alumno.nombre);
    printf("Edad %d\n",alumno.edad);
    printf("Matricula %ld\n",alumno.mat);
    printf("NRC de la carrera %d\n",alumno.nrc);
    printf("Nombre de la carrera %s\n",alumno.nombre_carrera);
   }
```