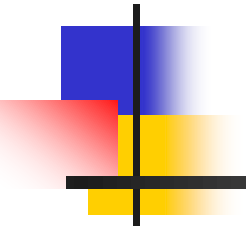


PROGRAMACIÓN ESTRUCTURADA

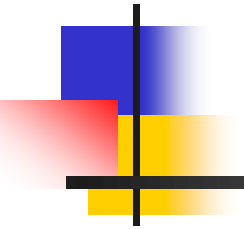
UNIDAD 3.- MODULARIDAD

COMPETENCIA:

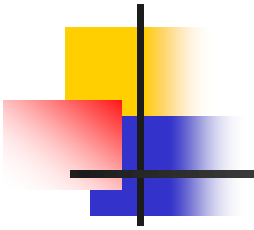
Desarrolla programas computacionales en forma modular, mediante el empleo de funciones.

SECUENCIA DE CONTENIDOS:

1. Introducción a la modularidad.
2. Funciones definidas por el usuario.
3. Paso de parámetros a funciones.
4. Ámbito de las variables y clases de almacenamiento.
5. Funciones de biblioteca.



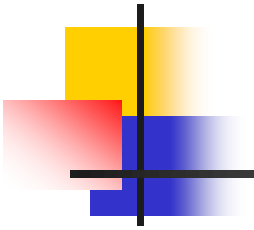
1. Introducción a la modularidad



Modularidad

- La **modularidad** es la capacidad que tiene un sistema de ser estudiado, visto o entendido como la unión de varias partes que interactúan entre sí y que trabajan para alcanzar un objetivo común, realizando cada una de ellas una tarea necesaria para la consecución de dicho objetivo.

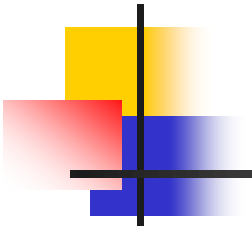
- Idealmente un módulo debe poder cumplir las condiciones de caja negra, es decir, ser independiente del resto de los módulos y comunicarse con ellos (con todos o sólo con una parte) a través de unas entradas y salidas bien definidas.



Modularidad

- Muchos lenguajes están diseñados para permitir la Programación Modular. La modularidad significa dividir un programa en varios módulos pequeños en vez de uno solo grande.

- Una función o módulo es una sección de código a la que se le da un nombre. Un módulo también es llamado:
  - segmento
  - subprograma
  - subrutina
  - función
  - procedimiento

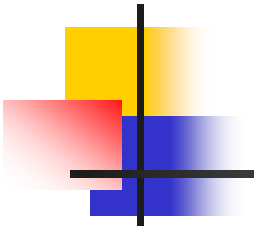


Modularidad

- Un módulo es un conjunto de instrucciones que de manera ordenada y lógica realizan una tarea específica. A esta unidad, se le podrá invocar cuantas veces sea necesario desde el programa principal o desde otros módulos.

- La idea de contar con módulos es permitir ver segmentos de código como una unidad de software con características especificas y definidas.

- Esta forma de programar es muy poderosa pues permite solucionar problemas complejos y difíciles de manejar, por medio de la división del problema en subproblemas más específicos.



Modularidad: beneficios

- Una vez definido un módulo y asignado un nombre, su código puede ser llamado el número de veces que se requiera.
- Segmenta y divide en tareas menores la solución de un problema extenso o complejo.
- Permiten distribuir y estructurar de una mejor manera el código de programas complejos.
- Permiten aislar los problemas.
- Al modularizar un programa es más legible, sencillo y permite un mejor mantenimiento y depuración.

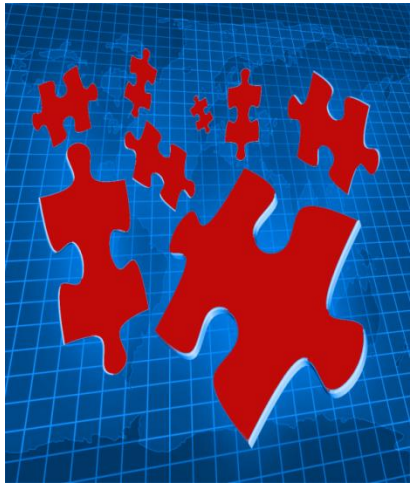
Criterios para la modularización

- Un programa grande que resuelva un problema complejo siempre ha de dividirse en módulos para ser más manejable. Aunque la división no garantiza un sistema bien organizado será preciso encontrar reglas que permitan conseguir esa buena organización.

- Uno de los criterios clave en la división es la independencia, esto es, el acoplamiento de los módulos; otro criterio es que cada módulo debe ejecutar una sola tarea, una función relacionada con el problema.

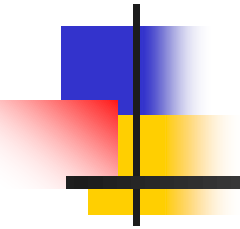
Cohesión de módulos

- El acoplamiento se refiere al grado de independencia entre módulos.
- El grado de acoplamiento se puede utilizar para evaluar la calidad de un diseño de sistema.
- Es preciso minimizar el acoplamiento entre módulos, es decir, minimizar su interdependencia.
- El criterio de acoplamiento es una medida para evaluar cómo un sistema ha sido modularizado. Este criterio sugiere que un sistema bien modularizado es aquel en que las interfaces sean claras y sencillas.

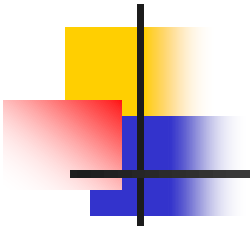


Cohesión de módulos

- La cohesión es la propiedad que se refiere a la fortaleza interna de un módulo, es decir, lo fuertemente relacionados que están entre sí las parte de un módulo.
- Este criterio se utiliza para juzgar un diseño mediante el examen de cada módulo del sistema buscando determinar la fuerza de enlace dentro del módulo.
- Un modelo cuyas partes estén fuertemente relacionadas con cada uno de los otros se dice que es fuertemente cohesivo.



2. Funciones definidas por el usuario



Función

- Una función es un segmento de programa que realiza una determinada tarea. Todo programa C consta de una o más funciones. Una de estas funciones se debe llamar **main()**. Todo programa comienza su ejecución en la función **main().**

- Las funciones en C no se pueden anidar, es decir, no puede haber una función dentro de otra función.

- En C todas las funciones son globales o externas, es decir, se pueden llamar desde cualquier punto del programa.

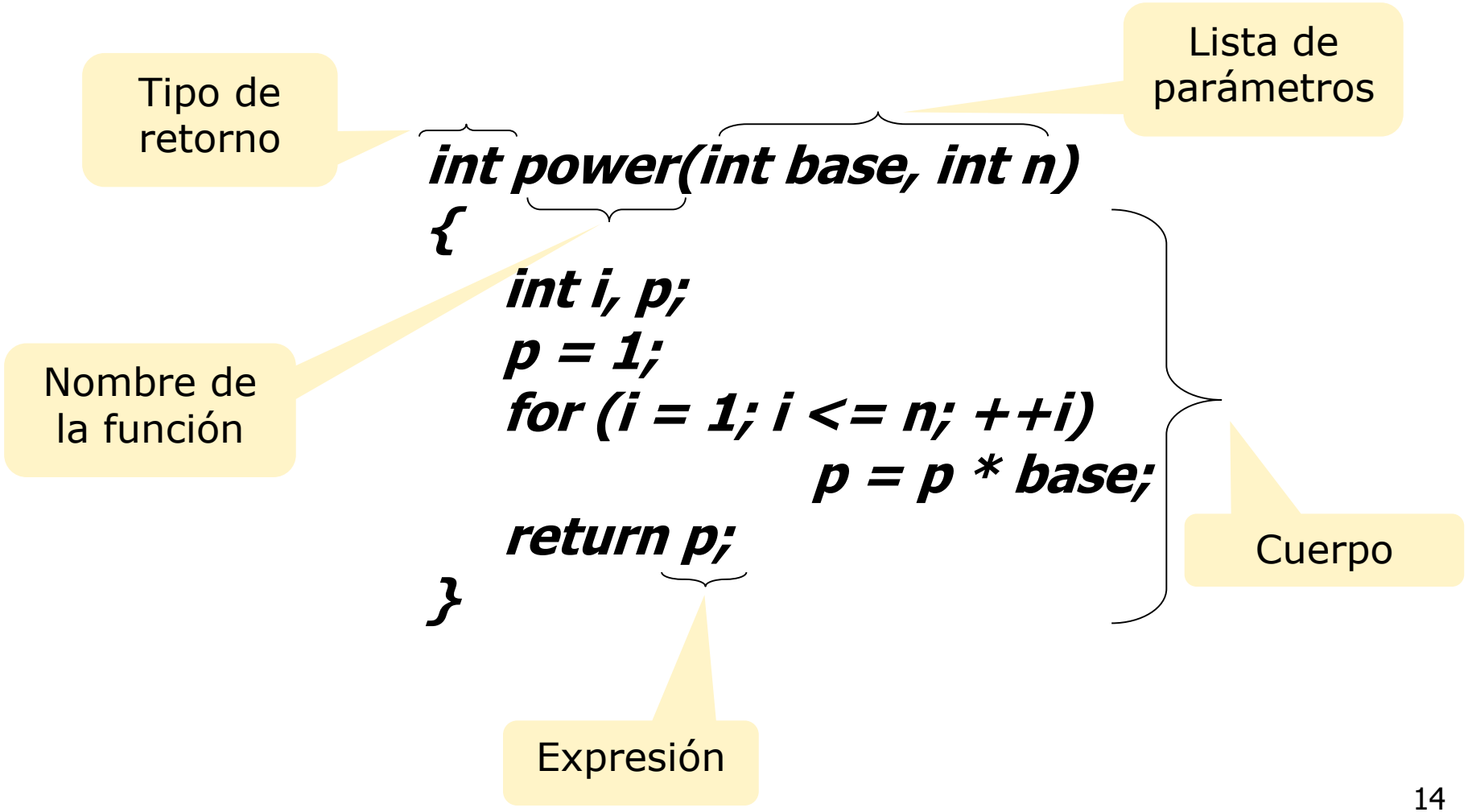
- El uso de funciones en C permite la descomposición y desarrollo modular. Permite dividir un programa en componentes más pequeños y permite la reutilización de código.

Sintaxis de una función

```
Tipo_retorno nombre_función(Lista_parámetros)
{
    sentencias;
    return Expresión;
}
```

- Donde:
  - *Tipo_retorno*: Tipo de dato del valor devuelto por la función (void si la función no devuelve ningún valor).
  - *nombre_función*: nombre de la función.
  - *Lista_parámetros*: Lista de declaraciones de los parámetros de la función (separados por comas).
  - *Expresión*: Valor que devuelve la función.

Ejemplo: pow(x,y)



Declaración de prototipos de funciones

- No es obligatorio pero si aconsejable.
- Van al inicio del archivo, antes de la utilización de las mismas.
- Permite la comprobación de errores entre las llamadas a una función y la definición de la función correspondiente.
- Verifica los tipos de datos de entrada y salida en el proceso de compilación.
- Sintaxis:

Tipo_retorno nombre_función (Lista_parámetros);

Llamadas a funciones

- Para llamar a una función se especifica su nombre y la lista de argumentos, sin poner el tipo de dato.

nombre_funcion (valor1, valor2,…,valorN);

- En una llamada habrá un argumento real por cada argumento formal, respetando el orden de la declaración.

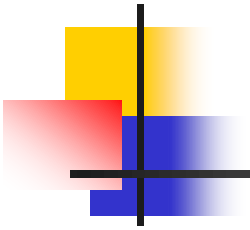
- Los **parámetros reales** pueden ser:
  - Constantes.
  - Variables simples.
  - Expresiones complejas.

Ejemplo:

```
#include <stdio.h>

/*Declaración de las funciones*/
int leer_dato(void);
void compara_datos(int,int);

int main() {
        int a,b,c;

        /*Llamadas a las funciones*/
        a=leer_dato();
        b=leer_dato();
        compara_datos(a,b);
        return 0;
}

/*Funciones*/
int leer_dato(){
        int dato;
        puts("Proporcionar dato:");
        scanf("%d: ", &dato);
        return dato;
}

void compara_datos(int x, int y){
        if (x > y) printf("a=%d es mayor que b= %d",x,y);
        else if (y > x) printf("b=%d es mayor que a=%d",y,x);
        else printf("a=%d es igual a b=%d",x,y);
        return;
}
```

Argumentos reales

Argumentos formales



Ejemplo:

- Se tienen dos funciones, cuadrado() y suma()
  - La función cuadrado() calcula y escribe los cuadrados de números enteros sucesivos a partir de un número dado (n) hasta obtener un cuadrado mayor de 1000.
  - La función suma() calcula la suma de un número determinado de elementos leídos desde el teclado.

Continuación ...

```
void cuadrado( int );
float suma(int );

int main(){
   cuadrado(3);
   float k = suma(3);
   printf("La suma es: %.3f", k);
}
```

Prototipo de las funciones

Llamada a las funciones

Continuación…

- Implementación de la función cuadrado

```
void cuadrado(int n) {
    int q=0;
    while(q<=1000) {
                q = n*n;
                printf("El cuadrado de: %d es %d \n",n,q);
                n++;
    }
}
```

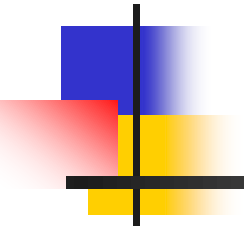
Continuación…

- Implementación de la función Suma

```
float suma(int num_ele){
    int i;
    float total = 0.0;
    printf("\n\t Introduce %d numeros reales\n", num_ele);
    for (i=0;i< num_ele;i++) {
                float x;
                scanf("%f",&x);
                total+=x;
    }
return total;
}
```

Tipo de dato de retorno

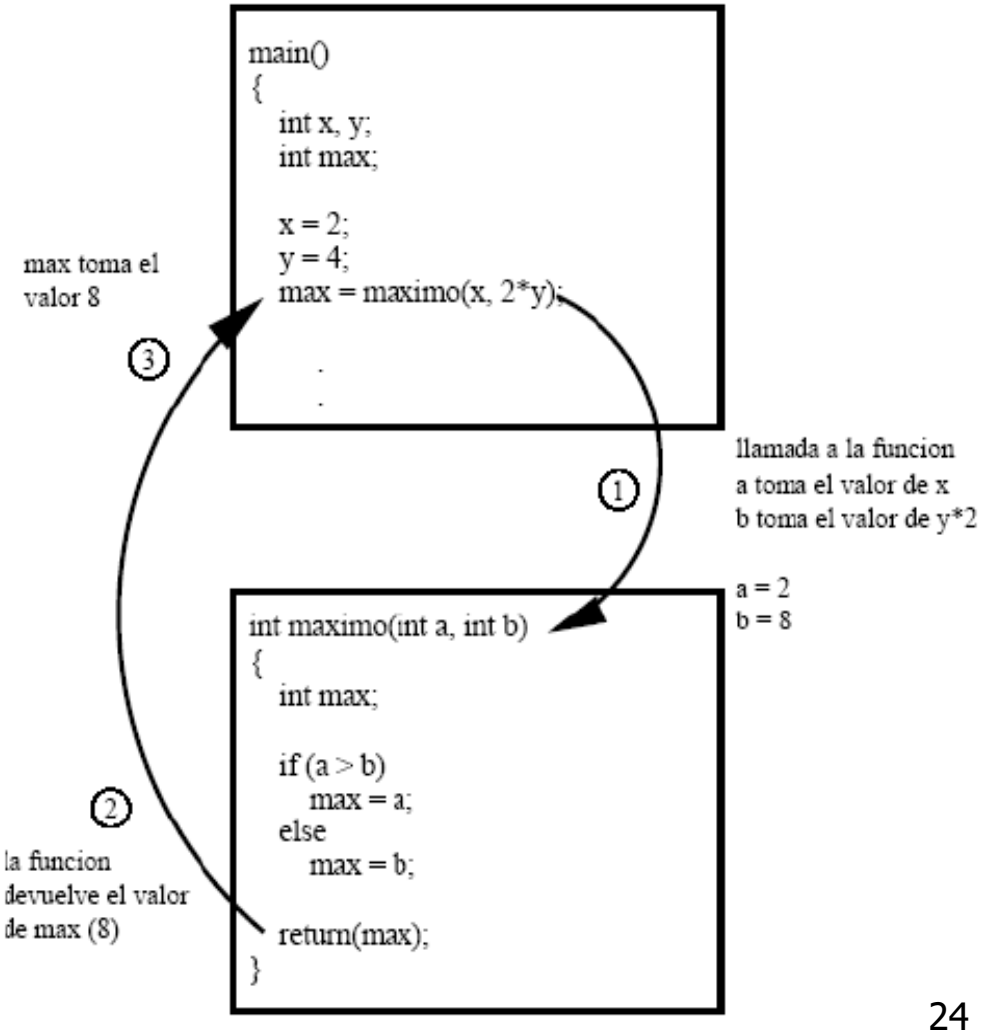
```
int max(int x, int y);                  //devuelve un tipo int
double media(double x, int y);          //devuelve un tipo double
float func0(int n);                     //devuelve un tipo float
char func1(void);                       //devuelve un tipo char
int func2(void)                         //devuelve un tipo int
void visualizardatos()                  //no devuelve datos
void visualiza(int a, float b)          //no devuelve datos
```



3. Paso de parámetros a funciones

Paso de parámetros por valor

- En el paso de parámetros por valor se hace una **copia del valor** del argumento en el parámetro formal.
- La función opera internamente con estos últimos.
- Como las variables locales a una función (y los parámetros formales lo son) se crean al entrar a la función y se destruyen al salir de ella, cualquier cambio realizado por la función en los parámetros formales no tiene ningún efecto sobre los argumentos.



Ejemplo:

```
void local(int);

int main(){
        int n=10;
        printf("antes de la funcion n= %d\n",n);
        local(n);
        printf("despues de la funcion n= %d\n",n);
}

void local(int valor){
        printf("dentro de local, valor = %d\n",valor);
        valor = 999;
        printf("dentro de local, valor = %d\n",valor);
}
```

Paso de parámetros por referencia

- Cuando una función modifica el valor del parámetro pasado y devuelve este valor modificado a la función llamadora, se ha de utilizar el método de paso de parámetros por referencia.

- El compilador pasa la dirección de memoria del valor del parámetro a la función.

- Cuando se modifica el valor del parámetro, este valor queda almacenado en la misma dirección de memoria, por lo que al retornar a la función llamadora la dirección de la memoria donde se almacenó el parámetro contendrá el valor modificado.

Paso de parámetros por referencia

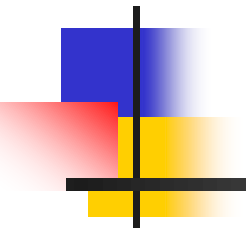
- Para pasar una variable por referencia, se utiliza el símbolo *&* precediendo al nombre de la variable.
- El parámetro variable correspondiente de la función debe declararse como puntero (apuntador).
- C utiliza tipos de datos apuntadores para implementar parámetros por referencia.

x = 1, y = 2;&x = 100 &y = 200	

100		200	
x	1	y	2

Ejemplo

```
void f(int* );

int main() {
  int x = 47;
  printf ("x = %d \n", x );
  printf ("direccion de x = %x \n", &x );
  f(&x);
  printf ("x = %d \n", x );
  return 0;
}
void f(int* r) {
    printf ("r = %d \n", r );
    *r = 5;
    printf ("r = %d \n", *r );
}
```



4. Ámbito de las variables y clases de almacenamiento

Ámbito de las variables

- Las variables declaradas pueden ser "vistas" o no dependiendo del lugar donde fueron declaradas. El ámbito de la variable determina desde que punto del programa las variables pueden ser vistas y utilizadas. Existen cuatro tipos de ámbitos:

  - Ámbito del programa: Son referenciadas por cualquier función en el programa completo; tales variables se llaman variables globales. Se declaran al principio del programa.

  - Ámbito del archivo fuente: Una variable con ámbito de archivo fuente, se declara fuera de cualquier función y su declaración contiene la palabra reservada ***static***. Están accesibles desde cualquier lugar del archivo de código fuente (posterior a su declaración).

Ámbito de las variables

- Ámbito de función: Una variable con ámbito de función, conocidas como variables locales o automáticas, son definidas y utilizadas sólo dentro de una función o módulo. Los parámetros pasados por valor a las funciones se les considera variables locales.

- Ámbito de bloque: Una variable declarada dentro de un bloque tiene ámbito de bloque y puede ser referenciada desde cualquier parte del bloque.

Ejemplo:

```
#include <stdio.h>

void imprime();

int a=10;        //Ámbito del programa

int main() {
        a +=5;
        printf("a=% d\n", a);
        imprime();
        //printf("%d\n", c);            Error Ámbito de función

        for (int i=1;i<5; i++){          //Ámbito de bloque
                printf("i= %d\n", i);
        }
        //printf("%d\n", i);            Error Ámbito bloque
        if(a==20){
                int j=-1;        //Ámbito de bloque
                printf("j= %d\n", j);
        }
        //printf("%d\n", j);            Error Ámbito bloque
        //printf("%d\n", m);            Error Ámbito archivo fuente
return 0;
}

static int m=0;          //Ámbito de archivo fuente

void imprime(){
        int c=0;          //Ámbito de función
        c+=5;
        printf("c=%d\n", c);

        a +=5;
        printf("%a= d\n", a);

        printf("m= %d\n", m);
        return;
}
```

Clases de almacenamiento

- Los especificadores de clase de almacenamiento permiten modificar el ámbito de una variable. Los especificadores pueden ser uno de los siguientes: ***auto, register, static, extern.***

  - Variables automáticas:
    - Son las variables que se declaran dentro de una función (***auto***), se les asigna la memoria de manera automática y se les libera el espacio cuando salen de la función. La palabra reservada ***auto*** es opcional. Ejemplo:

      auto int total; //Esto es igual a int total;

Clases de almacenamiento

- Variables registro:
  - Cuando la variable se declara de tipo registro (***register***) se le sugiere al procesador que la variable se almacene en uno de los registros hardware del microprocesador. Esto es una sugerencia, más no se le ordena al compilador para que almacene la variable en su registro. La variable debe ser local a una función, no global al programa completo. Sirve para que el acceso a la variable sea más rápido. Ejemplo:

```
register int indice;
for(indice = 0; indice<1000; indice++)…
```

Clases de almacenamiento

- Variables estáticas:
  - Las variables estáticas (**static**) no se borran (no pierden su valor) cuando la función termina, y así, retienen su valor entre llamadas a una función. Este tipo de variables se inicializan una sola vez. Se utilizan normalmente para mantener valores entre llamadas a funciones, ejemplo:

```
float resultadoTotal(float valor){
        static float suma = 0;
        suma = suma + valor;
        return suma;
}
```

Clases de almacenamiento

- Variables externas:
  - Son utilizadas cuando una función necesita utilizar una variable que otra función inicializa. Como las variables locales sólo existen temporalmente mientras se ejecuta su función, no pueden resolver el problema. Cuando una variable se declara externa (***extern***) *se indica al compilador que el espacio de la variable está* definida en otro lugar.

Ejercicios:

1. Hacer la función que calcule el número de fibonacci dado.

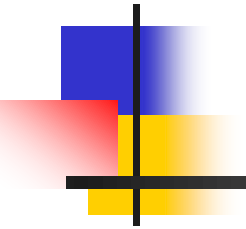
1. Hacer la función factorial y la función potencia. Usar estas funciones para resolver el problema siguiente:

13. Elaborar un programa que lea un valor para X; que utilice una función COSENO que reciba dicho parámetro y calcule el coseno de X, al regresar al algoritmo principal que lo imprima. Con el mismo valor, utilizar la función EXPONENCIAL que haga lo propio, pero calculando el exponencial de X. Utilizando las siguientes fórmulas:

$$\text{Coseno}(X) = 1 - \frac{X^2}{2!} + \frac{X^4}{4!} - \frac{X^6}{6!} + \ldots\ldots\ldots -$$

$$\text{Exponencial}(X) = 1 + X + \frac{X^2}{2!} + \frac{X^3}{3!} + \frac{X^4}{4!} + \ldots\ldots\ldots -$$

hasta que la diferencia entre dos valores sea menor que 0.001.



5. Funciones de Biblioteca

Funciones de biblioteca

- Todas las versiones del lenguaje C ofrecen una biblioteca estándar de funciones que proporcionan soporte para operaciones utilizadas con más frecuencia. Estas funciones permiten realizar una operación con sólo una llamada a la función (sin necesidad de escribir el código fuente).

- Estas funciones estándar o definidas se dividen en grupos; todas las funciones que pertenecen al mismo grupo se declaran en el mismo archivo de cabecera.

- Ejemplos de archivos de cabecera estándar que se pueden utilizar en los programas:
  - <ctype.h>, <math.h>, <stdio.h>, <stdlib.h>, <string.h>, <time.h>, <conio.h>etc.

Ejercicios

1. Hacer un programa que mediante una función que reciba dos parámetros (a, b) calcule la hipotenusa de un triángulo rectángulo. Regresa el resultado por valor.

   Fórmula: $h = \sqrt{(a^2 + b^2)}$

2. Escriba una función distancia() que calcule la distancia entre dos puntos, dicha función recibe los parámetros enteros: x1, y1 y x2, y2 y regresa el resultado como un valor double. La fórmula para encontrar la distancia entre dos puntos es:

   Distancia = $\sqrt{((x1 - x2)^2 + (y1 - y2)^2)}$.

   Luego desde el programa principal, se desea saber cual es el lado más corto de un triángulo cuyas vértices están en los puntos p1(2, 2) , p2(7, -3) y p3(4, 5). Utiliza la función distancia() para calcular las tres distancias entre los puntos y posteriormente indicar cual de ellos es el más corto mediante un mensaje de texto.

Ejercicio

- Hacer un programa que mediante una función que reciba dos parámetros (a, b) calcule la hipotenusa de un triángulo rectángulo. Regresa el resultado por referencia.

Fórmula: $h = \sqrt{(a^2 + b^2)}$

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

void hipotenusa(float, float, float *)
void leer (float *, float *)

int main(){
    float a, b, h;
    leer (&a,&b);
    hipotenusa(a, b, &h);
    printf("La hipotenusa es %f\n", h);
    return 0;
}
```

```
void hipotenusa(float a, float b, float *h){
    *h = sqrt(pow(a,2) + pow(b, 2));
    return;
}

void leer (float *a, float *b){
    printf("Dame valores a y b:\n");
    scanf("%f %f", *&a, *&b);
    return;
}
```

Bibliografía

- Brian Kernighan; P. J. Plauger (1976), **Software Tools**, Addison-Wesley, pp. 352, ISBN 020103669X.
- Cairó Oswaldo. **Metodología de la programación- Algoritmos, diagramas de flujo y programas-.** Ed. Alfaomega.
- Joyanes Aguilar, Luis. **Fundamentos de programación**. Ed. Mc Graw Hill.
- Joyanes Aguilar, Luis. **Programación en C.** McGraw Hill.
- Schildt, Herbert. **Lenguaje C, Programación Avanzada**. McGraw Hill.
- Bronson, Gary. **Algorithm Development and Program Design Using C**, PWS Publishing Co.; Book and Disk edition (February 15, 1996) ISBN: 0314069879.
- Standish, Thomas. **Data Structures, Algorithms, and Software Principles in C**, Addison-Wesley Pub Co; 1st edition (September 30, 1994), ISBN: 0201591189.
- Linden, Peter. **Expert C Programming**, Prentice Hall, 1994, ISBN 0131774298.
- Reek, Kenneth. **Pointers on C**. Addison Wesley, 1997. ISBN 067399866.
- Kernighan, Brian W. & Dennis, M. Richie. **El lenguaje de programación C**. 2ª Ed. Prentice Hall, 1988.
- http://zinjai.sourceforge.net/