

UNIDAD 4.

Introducción a la POO. Funciones.



Índice

1. Ficha unidad didáctica.....	1
2. Contenidos.....	2
2.1. Introducción.....	2
2.2. Programación estructurada y funciones.....	2
2.2.1 Concepto.....	2
2.2.2 Declaración y uso.....	4
2.2.3 Paso de parámetros.....	7
2.2.4 Recursividad.....	13
3. Actividades y ejercicios.....	15



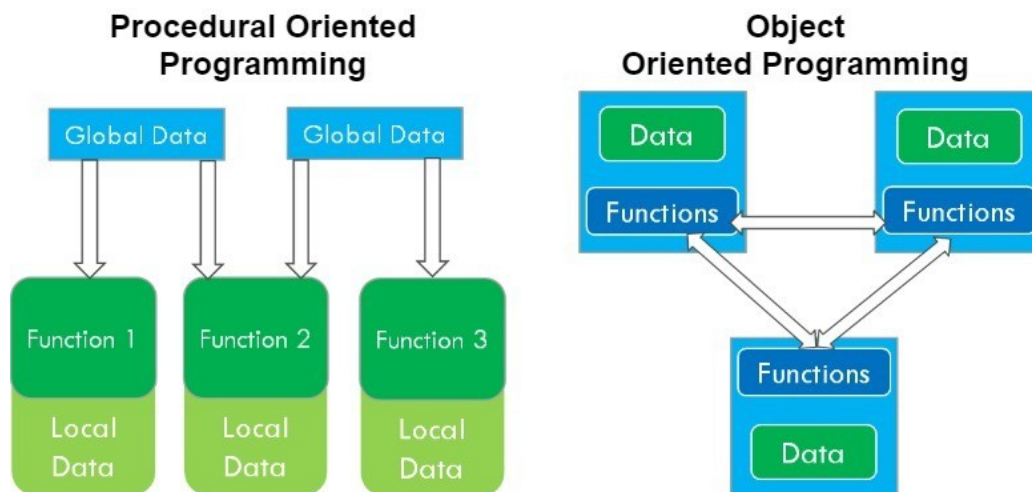
1. Ficha unidad didáctica.

Temporalización: De la semana 12 a la semana 21 del curso (navidades). 42 horas.	
OBJETIVOS DIDÁCTICOS	
<p>OD1: Evaluar las ventajas de la OO.</p> <p>OD2: Analizar la estructura básica de clases y objetos.</p> <p>OD3: Utilizar e instanciar objetos a partir de clases.</p> <p>OD4: Reconocer los diferentes ámbitos de alcance de métodos y atributos.</p> <p>OD5: Elaborar programas con creación de clases, instanciación y uso de los mismos.</p> <p>OD6: Analizar y modificar clases con métodos y atributos con diferente visibilidad.</p> <p>OD7: Comprender la importancia de la protección de datos .</p> <p>EV2. Concienciar de la importancia de emplear hábitos respetuosos con el medio ambiente y control del gasto energético de las instalaciones informáticas</p> <p>EV3. Reconocer de la importancia del trabajo en grupo en el ámbito empresarial.</p> <p>RL1. Utilizar de forma adecuada y ergonómica el mobiliario de oficina, evitando posturas incorrectas que conlleven lesiones.</p> <p>TIC1. Usar Internet para obtener información técnica.</p> <p>ID1. Habituar a la lectura de documentación técnica en inglés.</p>	
RESULTADOS DE APRENDIZAJE	RA4
CONTENIDOS	
<p>Paradigma OO.</p> <p>Estructura básica de clases.</p> <p>Uso de clases, objetos y métodos.</p> <p>Paso de parámetros.</p> <p>Librerías y paquetes.</p> <p>Estructura interna de clase.</p> <p>Definición de parámetros en métodos.</p> <p>Objetos y gestión de memoria.</p> <p>Empaquetado de clases. Organización de las clases en paquetes.</p> <p>Documentación.</p> <p>Creación y uso de constructores avanzados.</p> <p>Encapsulación y visibilidad.</p> <p>Análisis de otros lenguajes OO.</p>	
ORIENTACIONES METODOLÓGICAS	
<p>El planteamiento y desarrollo de las actividades potencian el desarrollo de la habilidad de resolver problemas de forma metódica.</p> <p>Se planifica el incremento gradual de la dificultad de las actividades.</p> <p>Desarrollar la capacidad de trabajo en grupo a través de actividades que la potencien.</p> <p>Las actividades de introducción se planifican para descubrir el uso y la tecnologías y despertar el interés por la materia.</p>	
CRITERIO DE EVALUACIÓN	2a, 2b,2c,2d,2e,2f,2g,2h, 2i,4a, 4b, 4c, 4d, 4e, 4f, 4g, 4h, 4i, 4j.

2. Contenidos.

2.1. Introducción.

En los años 80 el software comienza a crecer de forma exponencial y la programación funcional y/o estructurada no consigue crear software mantenible ni manejable para el nivel de complejidad requerido, además de problemas en la fase de desarrollo. Para intentar solventar este problema se recurre al paradigma OO, que une en una misma entidad los datos y la funcionalidad haciendo independientes del resto y pudiendo colaborar entre las mismas entidades, de forma similar a como se realiza en el mundo real.



A pesar de que la mayor parte de los lenguajes son OO la programación funcional sigue utilizándose en la actualidad y los lenguajes más usados permiten definir y usar funciones.

En este tema se trata el concepto de función, su declaración, uso y paso de parámetros, para a continuación introducir el concepto y uso de clases, objetos, métodos y constructores entre otros para finalizar con la creación de clases sencillas con atributos y métodos.

2.2. Programación estructurada y funciones.

2.2.1 Concepto.

Existen tareas que se repiten una y otra vez en diferentes puntos del programa, además son tareas que se pueden utilizar en diferentes programas. Reescribir una y otra vez el

mismo código parece una tarea un poco inútil, que puede facilitar que se comentan errores y que dificulta el mantenimiento del código.

Para solucionarlo se define el concepto de función que permite crear código reutilizable y ser usado en diferentes lugares. El concepto se hereda de la funciones matemáticas, por ejemplo coseno, \sum o \prod , y siguiendo con la analogía, una función en programación tiene:

- Un nombre, por ejemplo sumatorio.
- Un conjunto de parámetros, en la definición llamados **formales**, que tiene un tipo y un nombre que los identifica para su posterior uso, en el sumatorio puede ser un vector de enteros a sumar. Al utilizarse esos parámetros se denominan actuales o reales.
- Un cuerpo, instrucciones que realizan la acción o el cálculo y que utilizan los parámetros pasados a la función.
- Un valor de retorno. Al igual que una función matemática, una función en programación ha de devolver algún valor, es necesario en los lenguajes tipados indicar el tipo de dato que devuelve la función. En el sumatorio el valor devuelto será la suma y el tipo de retorno un entero o entero largo.

En pseudocódigo:

```
funcion sumatorio( elementos:entero[]) devuelve entero_largo:  
    var entero suma=0;  
    var entero i=0;  
    para i=0 i<longitud(elementos);i=i+1 hacer  
        suma=suma+ elementos[i]  
    fin para  
    devuelve suma  
fin funcion
```

Para utilizarlo simplemente escribir el nombre de la función y pasar las variables necesarias, y si es necesario asignar a otra variable para almacenar le resultado:

```
var entero[] numeros= nuevo entero[10]  
var entero i=0  
var entero resultado  
resultado=sumatorio(numeros)
```

Es posible volver a llamar a la función con otros parámetros en cualquier parte del programa.



Relacionadas con las funciones se tienen los procedimientos, aunque no en muchos lenguajes se encuentran implementados y no es muy popular su uso. Un procedimiento es similar a una función con la salvedad de que no devuelve ningún valor, pero en los parámetros de la misma se ha de indicar junto con el nombre y el tipo si es de entrada, salida o entrada salida.

2.2.2 Declaración y uso.

En Java no existen funciones como tal, los fragmentos de código van unidos a una clase (se trata posteriormente), pero se puede simular su declaración y uso haciendo las llamadas dentro de la misma clase, pero los conceptos y comportamiento son iguales a los de un lenguaje estructurado como C.

En C/C++ una función se define como:

```
tipo_de_retorno nombre_de_la_funcion (tipo_parametro nombre_parametro1...){  
    //instrucciones  
    return variable //tiene que ser del mismo tipo que tipo_de_retorno  
}
```

Un ejemplo de código que indica si un texto es palíndromo o no:

```
int is_palindrome(char*);  
void copy_string(char*, char*);  
void reverse_string(char*);  
int string_length(char*);  
int compare_string(char*, char*);  
  
int main()  
{  
    char string[100];  
    int result;  
  
    printf("Input a string\n");  
    gets(string);  
    result = is_palindrome(string);  
    if (result == 1)  
        printf("\"%s\" is a palindrome string.\n", string);  
    else  
        printf("\"%s\" isn't a palindrome string.\n", string);  
    return 0;
```

```
}

int is_palindrome(char *string)
{
    int check, length;
    char *reverse;
    length = string_length(string);
    reverse = (char*)malloc(length+1);
    copy_string(reverse, string);
    reverse_string(reverse);
    check = compare_string(string, reverse);
    free(reverse);
    if (check == 0)
        return 1;
    else
        return 0;
}

int string_length(char *string)
{
    int length = 0;
    while(*string)
    {
        length++;
        string++;
    }
    return length;
}

void copy_string(char *target, char *source)
{
    while(*source)
    {
        *target = *source;
        source++;
        target++;
    }
    *target = '\0';
}

void reverse_string(char *string)
{
    int length, c;
    char *begin, *end, temp;
```

```
length = string_length(string);
begin = string;
end = string;
for (c = 0; c < (length - 1); c++)
    end++;
for (c = 0; c < length/2; c++)
{
    temp = *end;
    *end = *begin;
    *begin = temp;
    begin++;
    end--;
}
}

int compare_string(char *first, char *second)
{
    while(*first==*second)
    {
        if (*first == '\0' || *second == '\0')
            break;
        first++;
        second++;
    }
    if (*first == '\0' && *second == '\0')
        return 0;
    else
        return -1;
}
```

Al inicio se declaran, esto se hace para indicar al compilador de que esas funciones no están completas pero se definen más abajo, de forma que se pueda usar antes de disponer del cuerpo, en los lenguajes actuales ya no es necesario.



Observar como se van llamando a las funciones con los parámetros. Esas funciones definidas ¿pueden ser necesarias en otros programas?

2.2.3 Paso de parámetros.

Los parámetros de una función son información que se les facilita a la misma para realizar su cometido. Se ha de diferenciar entre parámetros formales y reales.

Los **parámetros formales** son aquellos que se establecen en la definición de la función, estos parámetros han de poseer un tipo de dato (primitivo o compuesto) y un nombre, que será utilizado en el interior de la función. Por ejemplo en C:

```
#include <stdio.h>
int esprimo( int numero){
    int esprimo=1;
    int i;
    for(i=2;i<numero && esprimo;i++){
        if((numero % i)==0){
            esprimo=0;
        }
    }
    return esprimo;
}
```



A partir de la versión del año 99, el lenguaje C soporta los booleanos, anteriormente se utilizan los enteros, 0 → false, diferente de 0 → true. Se puede consultar algunos de los cambios en <https://en.wikipedia.org/wiki/C99>

El orden de los parámetros (en especial los tipos de datos ha de ser el correcto en la llamada).

Los **parámetros reales** son aquellos que se facilitan a la función en cada llamada, siguiendo con el ejemplo anterior:

```
int main() {
    int primo1=5,primo2=10;
    int resultado;
    //llamada a la función con parámetro real primo1
    resultado=esprimo(primo1);
    printf("%d ",esprimo(primo1));
    //llamada a la función con parámetro real primo2
    resultado=esprimo(primo2);
}
```

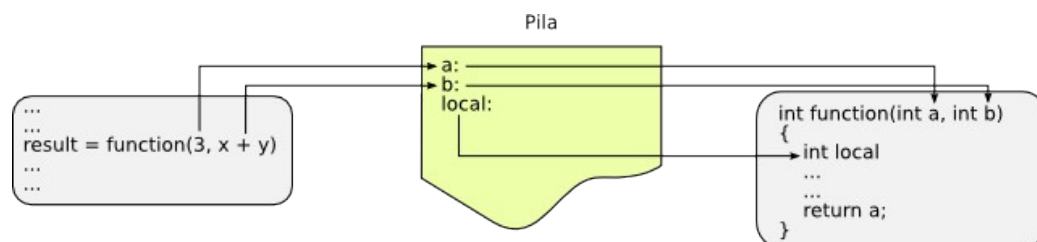
```
printf("%d ",esprimo(primo2)) ;
return 0;
}
```

```
my_function(10, 20, 30);           // function call

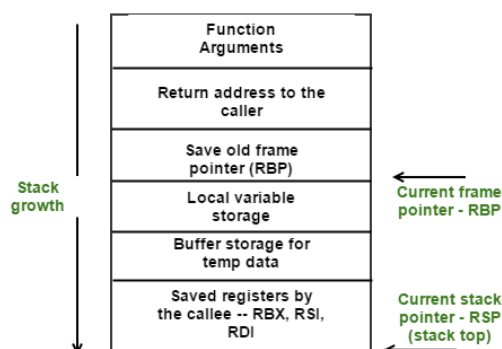
void my_function(int a, int b, int c) // function definition
{
    .
    .
    .
}
```

Otro de las características de las funciones es el **paso de parámetros por valor o por referencia**, que marcan si las modificaciones de las variables pasadas como parámetros no tienen efecto fuera de la función o si los cambios producidos son permanentes.

Para entender el funcionamiento de los dos tipos es importante saber como se pasan dichos parámetros a las funciones, al realizar una llamada a la función los parámetros formales se apilan en una estructura denominada pila.

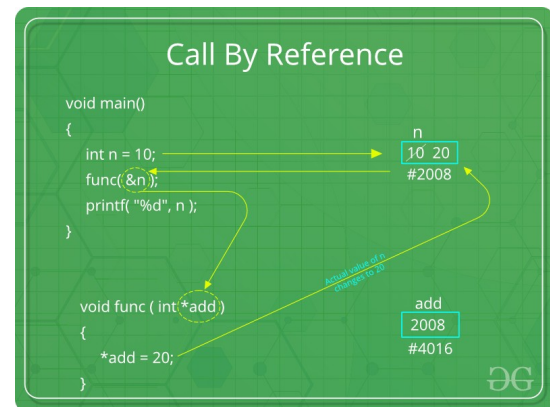
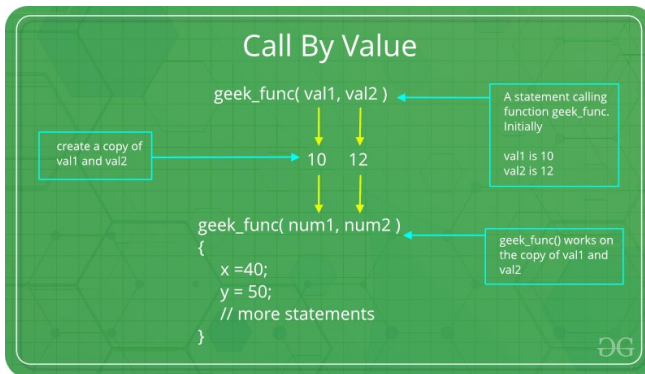


En esta pila se almacena diferente información como son los argumentos reales, la dirección de retorno (return), el estado del contador de programa antes de la llamada (SP) o las variables locales entre otra información:



En este caso se trabaja con direcciones de memoria, y puede ser la dirección

de memoria de la variable que se pasa como parámetro real o una copia de la memoria que ocupa dicha variable, en el primer caso se está pasando parámetros por referencia, ya que se pasa la dirección de memoria de la variable, y en el segundo por valor, ya que se pasa una copia de la variable.



En el caso de pasar valores por referencia **las modificaciones que se realizan dentro de la función se están realizando en la variable que se pasa como parámetro real, al salir de la función los cambios persisten, en cambio en el paso por valor los cambios no son persistentes ya que se ha pasado una copia de la variable.**

En el lenguaje C se utilizan punteros para pasar parámetros por referencia, un puntero almacena la dirección de memoria de una variable, para indicar que se recibe una dirección de memoria se utiliza el operador * en el parámetro formal, y en la llamada no se le pasa la variable sino la dirección que ocupa esta, con el operador &. Un ejemplo en C de paso por valor y paso por referencia:

Paso por valor:

```

#include <iostream>
using namespace std;
void func(int a, int b)
{
  a += b;
  cout << "In func, a = " << a << " b = " << b << endl;
}
int main(void)
{
  int x = 5, y = 7;
  // Passing parameters
  func(x, y);
  cout << "In main, x = " << x << " y = " << y;
}
  
```

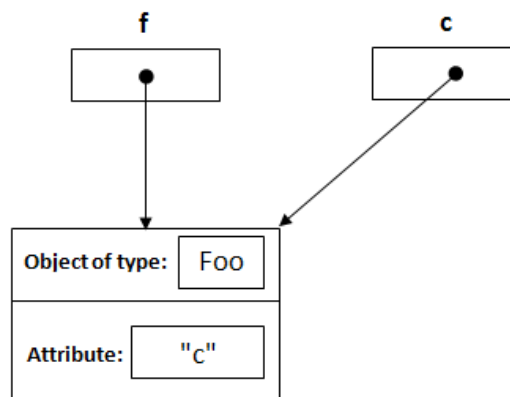
```
return 0;
}
```

Paso por referencia (punteros):

```
#include <stdio.h>
void swapnum(int* i, int* j)
{
    int temp = *i;
    *i = *j;
    *j = temp;
}

int main(void)
{
    int a = 10, b = 20;
    // passing parameters
    swapnum(&a, &b);
    printf("a is %d and b is %d\n", a, b);
    return 0;
}
```

En el caso de C++ se pueden utilizar punteros pero también aparece el concepto de **referencia**, también usado en Java, **una referencia es una alias de una variable (los dos señalan a la misma dirección de memoria)** que simplifica el proceso.



En este caso únicamente se ha de indicar en la definición de la función que se recibe una referencia, al llamar a la función se crea una referencia y esta es la que se pasa a la función, con lo que los cambios dentro de la función se realizan sobre el elemento real.

Un ejemplo que suma dos estructuras:

```
#include "iostream"
```

```
using namespace std;
struct Coordenada{
    int x;
    int y;
};
//Suma la dos coordenadas por valor no tiene efectos fuera de la función
void porvalor(Coordenada uno, Coordenada dos){
    uno.x+=dos.x;
    uno.y+=dos.y;
    cout<<"Dentro de la función al pasar por valor: uno.x vale "<<uno.x<<" y
uno.y vale "<<uno.y<<endl;
}
//Suma la dos coordenadas por valor no tiene efectos fuera de la función
void porreferencia(Coordenada &uno, Coordenada &dos){
    uno.x+=dos.x;
    uno.y+=dos.y;
    cout<<"Dentro de la función al pasar por referencia: uno.x vale "<<uno.x<<"
y uno.y vale "<<uno.y<<"\n";
}
int main() {
    struct Coordenada real_uno;
    struct Coordenada real_dos;
    //se asignan valores
    real_uno.x=5;
    real_uno.y=2;
    real_dos.x=1;
    real_dos.y=4;
    cout<<"Antes de llamar a las funciones: uno.x vale "<<real_uno.x<<" y uno.y
vale "<<real_uno.y<<endl;
    cout<<"LLamando por valor"<<endl;
    porvalor(real_uno,real_dos);
    cout<<"Al salir de por VALOR: uno.x vale "<<real_uno.x<<" y uno.y vale
"<<real_uno.y<<endl;
    cout<<"LLamando por referencia"<<endl;
    porreferencia(real_uno,real_dos);
    cout<<"Al salir de por REFERENCIA: uno.x vale "<<real_uno.x<<" y uno.y vale
"<<real_uno.y<<endl;
    return 0;
}
```

El resultado de la ejecución del programa anterior es:

```
Antes de llamar a las funciones: uno.x vale 5 y uno.y vale 2
LLamando por valor
```

Dentro de la función al pasar por valor: uno.x vale 6 y uno.y vale 6
Al salir de por VALOR: uno.x vale 5 y uno.y vale 2
LLamando por referencia
Dentro de la función al pasar por referencia: uno.x vale 6 y uno.y vale 6
Al salir de por REFERENCIA: uno.x vale 6 y uno.y vale 6



[Paso de parámetros por valor y por referencia. UPV](#)



Es curioso que al pasar un array o vector (ya sea de uno o más dimensiones), internamente un array se representa por la dirección de memoria de la primera posición, al llamar a una función con parámetro por valor, realmente se pasa una copia de la dirección de memoria del primer elemento, que implica que a efectos prácticos se realice un paso por referencia de arrays o vectores.

Un ejemplo de paso de arrays:

```
#include "iostream"
using namespace std;
void f_vector(int vector[]){
    for(int i=0;i<10;i++){
        vector[i]=vector[i]+1;
    }
}
int main() {
    int vector[]={0,1,2,3,4,5,6,7,8,9};
    for(int i=0;i<10;i++)
        cout<<vector[i]<<" ";
    cout<<endl;
    f_vector(vector);
    cout<<"El valor al terminar es\n";
    for(int i=0;i<10;i++)
        cout<<vector[i]<<" ";
    cout<<endl;
    return 0;
}
```

La salida del programa es:

```
0 1 2 3 4 5 6 7 8 9
El valor al terminar es
1 2 3 4 5 6 7 8 9 10
```



2.2.4 Recursividad.

Las funciones permiten resolver problemas relativamente complejos de forma sencilla, por ejemplo los algoritmos de backtranking, que consiste en buscar entre todas las posibles soluciones de un problema para encontrar la mejor o mejores soluciones (pudiendo cortar y podar posibles caminos en caso, por ejemplo los mejores movimientos para ganar una partida de ajedrez o la solución a un sudoku, o el algoritmo alfa-beta en juegos como Age of Empires (aunque en el caso del ajedrez el dominio de búsqueda es tan grande que no es posible calcular todas las posibilidades en un tiempo razonable).

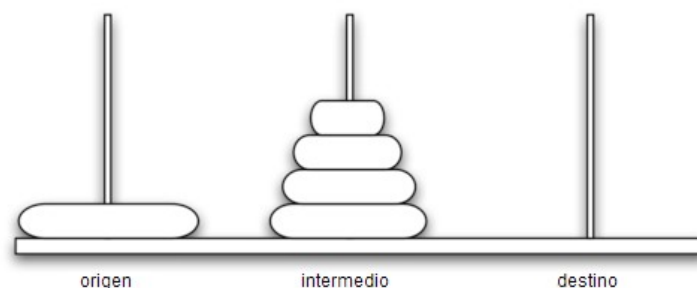
 [Problema de las N-Reinas con backtraking UPV.](#)

La recursividad consiste en resolver un problema realizando llamadas a una función dentro de si misma, por supuesto, tiene que existir una modificación en los parámetros que se pasan a la función, ya que en caso contrario se llega a un stackoverflow o desbordamiento de la pila (se queda sin memoria en la que apilar las llamadas).

Dentro de una función recursiva se tienen dos elementos básicos:

-  **Caso base:** Cuando se cumple finalizan las llamadas y provoca que se vayan resolviendo las llamadas previas en el orden inverso (desapilar).
-  **Caso recursivo:** Se puede tener más de uno en la función, es una llamada a la misma función dentro de esta, con la modificación de algunos parámetros. La función que llama se queda esperando a que la función llamada termine y devuelve un valor si así se ha definido.

El ejemplo clásico son las Torre de Hanoi:



Las reglas son:

- Solo se puede mover un disco cada vez y para mover otro los demás tienen que

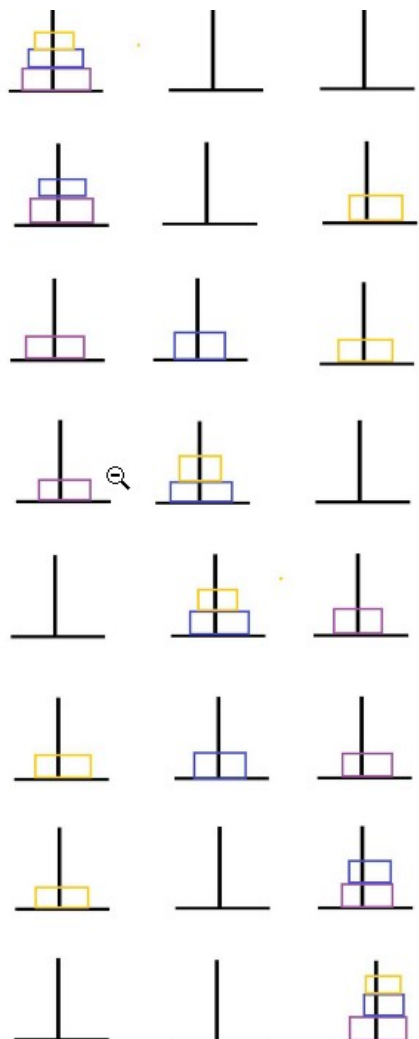
estar en postes.

- Un disco de mayor tamaño no puede estar sobre uno más pequeño que él mismo.
- Solo se puede desplazar el disco que se encuentre arriba en cada poste.

La resolución de este problema sin usar la recursividad es compleja, en cambio usando la recursividad:

```
void moverDiscos (int n, int origen, int destino, int aux){  
    if (n > 0){  
        moverDiscos (n-1, origen, aux, destino);  
        printf("Muevo un disco de %d a %d\n", origen, destino);  
        moverDiscos (n-1, aux, destino, origen);  
    }  
}
```

Un ejemplo de ejecución con 3 discos:





Diseñar un algoritmo que calcule el sumatorio de un número de forma recursiva.

Indicar en primer lugar el caso recursivo y el caso base.

3. Actividades y ejercicios.

- ? ¿Cómo se define una función?
- ? Indicar las diferencias entre funciones y procedimientos.
- ? Definir una función en Java, C o Pseudocódigo que eleve un número a otro y lo devuelva.
- ? ¿Java posee funciones? ¿Cómo simular las funciones en Java?
- ? Indicar las diferencias entre un programa en Java y el ejemplo del número primo.
- ? ¿Qué son los parámetros formales? ¿Y los reales?
- ? En el siguiente código indicar los parámetros formales con su tipo, los reales y si la llamada es correcta o no.
- ? ¿Qué consecuencias tiene pasar parámetros por valor? ¿Y por referencia?
- ? Explicar cómo se pasan parámetros por referencia en C.
- ? ¿Qué es un puntero? ¿Y una referencia?
- ? Explicar cómo se pasa parámetros en C++.
- ? En el siguiente código, escrito en el lenguaje C++:

```
#include "iostream"
#include <string>
using namespace std;
struct Producto{
    string nombre;
    float precio;
};

int ejercicio(int a, int &b, struct Producto &p1, struct Producto p2){
    a++;
    b++;
}
```

```
p1.nombre="Sandia";
p1.precio=3.33;
p2.nombre="Cerezas";
p2.precio=5.1;
return a+b;
}
int main() {
    struct Producto producto1,producto2;
    producto1.nombre="Tomate";
    producto1.precio=2.1;
    producto2.nombre="Manzana";
    producto2.precio=4.6;
    int uno,dos;
    ejercicio(uno,dos,producto1,producto2);
    return 0;
}
```

- Qué tipo de dato devuelve la función ejercicio? ¿Qué valor devuelve?
- Al salir de la función ¿Qué valor tendrá a? ¿Y b?.
- La variable a ¿se pasa por valor o por referencia?
- La variable b ¿se pasa por valor o por referencia?
- Al salir de la función ¿Qué vale el nombre de producto1?
- Al salir de la función ¿Qué vale el nombre del producto2?

? Crear una función en C++ que calcule el sumatorio de un número, por ejemplo del número 5 es 5+4+3+2+1.

? Explicar de forma textual y gráfica de lo que sucede al pasar un array como parámetro a una función.

? Indicar en las Torres de Hanoi el caso base y el caso recursivo.

? Realizar un algoritmo recursivo en pseudocódigo o c/c++ que indique si un número es primo o no (ojo con las soluciones de Internet, en caso de no entender recursividad y cambiarlo por cualquier otro problema....)

? Se tiene el siguiente código:

```
#include "iostream"
```

```
using namespace std;
int f(char c[],int b, int l){
    if (b>=l )
        return 1;
    else{
        if (c[b]==c[l])
            return f(c,b+1,l-1);
        else
            return 0;
    }
}
int main() {
    char mensaje[]={ 'h', 'd', 'o', 'h' };
    int resultado=f(mensaje,0,3);
    cout <<resultado;
    return 0;
}
```

- Indicar el caso base y el caso recursivo.
- Realizar una traza con la entrada que aparece.
- Realizar una traza con el valor Orejero
- ¿Qué hace la función f?