Índice

1. Repaso de C:
   - Tipos de variables.
   - Declaración de variables
   - Sentencias de control
   - Funciones. Paso de argumentos a una función.
   - Punteros, operaciones con punteros, punteros y arrays.

Repaso de lenguaje C

OBJETIVOS

Será necesario adquirir el **hábito** de realizar programas que tengan las siguientes características:

Rapidez: se deberá tener en cuenta el tiempo necesario para la programación.

Claridad: El programa esté escrito de forma clara. Añadir comentarios que describen los pasos realizados.

Modularidad: El programa debe estar separado en funciones, cada una de ellas con un cometido en particular. Esta característica permitirá que el programa sea además **ampliable**.

Portabilidad: El programa pueda ser compilado y ejecutado en diferentes computadores, funcionando con diferentes sistemas operativos (p.e. Windows y Linux).

Repaso de lenguaje C: Tipos básicos

- **Tipos de datos**: El lenguaje C nos permite utilizar una serie de datos básicos (primitivos). La selección del tipo de dato a utilizar dependerá de las necesidades del problema y del tipo de datos a almacenar.

Tipo de dato	Descripción	Tam.memoria
int	Entero	2 bytes
char	Caracter	1 byte
unsigned int	Entero positivo	2 bytes
float	Número real	4 bytes
double	Número real de doble precision	8 bytes
...	...	...

C permite definir tipos de datos más complejos a partir de estos tipos básicos →Estructuras.

Diferente rango, dependiendo del tamaño. P.e. int: [-32768 32767] y [0-65535] unsigned

El tamaño de cada tipo de dato en memoria depende del procesador, del compilador y del S.O. Ya que existen diferentes compiladores para procesadores diferentes.

Repaso de lenguaje C

Declaración de variables:

- Antes de utilizar una variable en C se debe declarar (se reserva un área en la memoria para almacenar un tipo de dato)

```
int num;
```

- El valor inicial de la variable es aleatorio, por lo que deberemos asignarle un valor antes de utilizarla.

```
num = 42;
```

- Se pueden declarar al tiempo varias variables de tipo entero.

```
int num, Num=0, num_2, 2num;
```

Repaso de lenguaje C: Operadores

Operador Aritmético	Nombre	Descripción
=	Asignación	Var. A la izquierda toma valor de la derecha
+	Adición	Suma dos operandos
-	Sustracción	Resta dos operandos
*	Producto	Multiplica dos operandos
/	División	Divide dos operandos
%	Resto	Devuelve el resto de la división entera

Operador Comparación	Nombre	Descripción
==	Igualdad	A==B
!=	Desigualdad	A!=B
>	Mayor que	A > B
<	Menor que	A<B
>=	Mayor o igual	A>=B
<=	Menor o igual	A<=B
&&	AND	AND lógico
||	OR	OR lógico

Repaso de lenguaje C: Bifurcaciones

```
if(expresion)
{
    sentencia;
}
```

```
if(expresion)
{
    sentencia_1;
}
else
{
    sentencia_2;
}
```

```
if(expresion1)
{
    sentencia_1;
}
else if(expresion2)
{
    sentencia_2;
}
else if(expresion3)
{
    sentencia_3;
}
else
{   sentencia_3;
}
```

```
switch(expresion)
{
    case 1: sentencia_1;
        break;
    case 2: sentencia_2;
        break;
    ...
    case N: sentencia_N;
        break;
    default: sentencia;
        break;
}
```


Repaso de lenguaje C: Bucles

```
while(expresion)
{
    sentencia;
}
```

```
for(inicializacion; expresion_control; actualizacion)
{
    sentencia;
}
```

```
inicializacion;
while(expresion_control)
{
    sentencia;
    actualizacion;
}
```


Repaso de lenguaje C: Errores comunes

Ejercicio 1.- Descubrir errores

```
int i = 0;
while(i<100)
{
    printf("%d\n", i);
    i++;
    if(i=50)
        break;
}
printf("Acabado el bucle while");
```

Ejercicio 2.- Descubrir errores

```
int i = 0;
while(i<10)
{
    if(i=!0)
        printf("%d\n", i);
    i++;
}
printf("Acabado el bucle while");
```

```
1                   1
2                   1
3                   1
4                   1
5 ...               1 ...
```

Solución errores comunes

- Ejemplo 1.
  - Dentro del if, ponemos i=50, cuando debemos poner i==50. Qué pasa cuando ponemos i=50? Se asigna 50 a i, se evalúa i. Cómo i es mayor que cero, la expresión del if resulta ser cierta, con lo que se ejecuta break y sale del bucle.
- Ejemplo 2.
  - Esperamos que imprima todos los números del 0 al 10 menos el 0. Pero resulta que imprime muchos 1. Dentro del if, otra vez. Estamos negando el cero, asignándolo a i y ejecutando la sentencia dentro del if.

Funciones

Repaso de lenguaje C: Funciones

Calcular el máximo de dos números introducidos por teclado

Ejemplo 3:
Sin funciones

```
void main(void)
{
      int x, y;
      printf("\nIntroduzca x");
      scanf("%d", &x);
      printf("\nIntroduzca y");
      scanf("%d", &y);
      if(x>y)
            printf("\nEl maximo es x=%d", x);
      else
            printf("\nEl maximo es y=%d", y);

      printf("\nIntroduzca x");
      scanf("%d", &x);
      printf("\nIntroduzca y");
      scanf("%d", &y);
      if(x>y)
            printf("\nEl maximo es x=%d", x);
      else
            printf("\nEl maximo es y=%d", y);
}
```


Repaso de lenguaje C: Funciones

- Inconvenientes del código anterior:
  - Es bastante largo.
  - si queremos extender su funcionalidad es difícil de modificar.
- Las funciones permiten dividir un programa en módulos, más pequeños y manejables (subprogramas o funciones que son llamadas por el programa principal.)

Repaso de lenguaje C: Funciones

Ventajas de las funciones

- **Modularidad**:
  - A cada función se le asigna una misión concreta.
  - No debe tener un número de líneas excesivo.
  - Recibe unos parámetros de entrada, realiza una serie de cálculos y ofrece unos resultados.
- **Reutilización de código**: Cada función puede llamarse muchas veces desde un programa. Con lo que se reduce el número de líneas de código del programa. Además, se puede reutilizar en diferentes programas.

A la hora de escribir un programa en C, deberemos dividir el problema general en problemas más pequeños, y programarlos separadamente en funciones. La división de un programa en funciones no es única y depende del estilo de cada persona

Repaso de lenguaje C: Funciones

Calcular el máximo de dos números introducidos por teclado

Ejemplo 3:

Con funciones

```
#include <stdio.h>

void HallarMaximo(void); //Declaración, prototipo

void main(void)
{
        HallarMaximo(); //Llamada
        HallarMaximo();
}

void HallarMaximo(void) //Definición, implementación
{
        int x, y;
        printf("\nIntroduzca x");
        scanf("%d", &x);
        printf("\nIntroduzca y");
        scanf("%d", &y);
        if(x>y)
                printf("\nEl maximo es x=%d", x);
        else
                printf("\nEl maximo es y=%d", y);
}
```


Repaso de lenguaje C: Funciones, ejemplos

Ejemplo 4

Ver si 'n' es primo y multiplicar 2 números en coma flotante entre ellos

```
#include <stdio.h>

//Declaración, prototipo
int Primo(int n);
float Multiplica(float x, float y);


void main(void)
{
    int n = 7, res1;
    float a = 1.5;
    float b = 2.0;
    float c;

    res1=Primo(n);
    if(res1)
        printf("\nEl número es
    primo");

    c=Multiplica(a, b);
}
```

```
int Primo(int n)
{
    int i;

    for(i=2;i<n;i++)
    {
        if (n%i==0)
                return 0;
    }
    return 1;
}

float Multiplica(float par1, float par2)
{
    return par1*par2;
}
```


Paso de argumentos a una función

- C permite pasar la dirección de una variable como argumento de una función.
- Esto posibilita cambiar la variable dentro del cuerpo de la función.
- Paso de argumentos a una función:
  - Paso por valor: Se le pasa el valor de la variable. Se altera una copia de la variable, no la variable original.
  - Paso por dirección: El contenido de la dirección puede ser cambiado. Se cambia la variable original.
- El paso por dirección permite a una función devolver más de un dato (***Paso por dirección Vs return***)

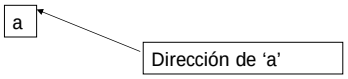
Punteros

Punteros: *Concepto y utilidad*

ReCoRDaR: *El valor de cada variable está almacenado en un lugar de la memoria, caracterizado por una dirección*

- **Variable a → dirección: 0023:0001**

- **Puntero**: Variable que almacena la dirección de una variable en memoria.



Punteros: *Concepto y utilidad*

- **¿Por qué usar punteros?**
  - *Para que una función devuelva más de un valor*
    - Recordar que con "return" solo podemos devover un valor
  - *Para un mejor uso de la memoria dinámica*
    - ayuda a ahorrar memoria y por consiguiente, hace más efectivo el uso y administración de la misma.

Se debe distinguir entre la dirección de memoria a la que apunta la variable puntero y el contenido de esa dirección.

- En esa zona de memoria tendremos almacenado un dato.
- El número de bytes que ocupa una variable en memoria depende de su tipo.
  - ej. un caracter ASCII ocupa un byte. El compilador se encarga de estos detalles.

Punteros: *Declaración*

- Declaración de un puntero.

tipo-dato *pvar;

pvar: es el nombre de la variable puntero.
tipo-dato: es el tipo de variable al que puede apuntar (int, float, uint...).

```
int *pentero;
```

Un puntero puede apuntar a variables ordinarias, arrays, funciones o a otros punteros

Punteros: *Operadores & y **

Operadores

- **Operador dirección**: &
  - Da la dirección de una variable.
- **Operador indirección**: *
  - Permite acceder al contenido de una dirección a la que apunta un puntero.

Ejercicio 3 y ejercicio 4
hoja 1 problemas resueltos

Punteros: Errores de sintaxis de punteros

- Errores comunes de sintaxis:
  - `a=&12;` `// las constantes no tienen dirección`
  - `b=&(i+4);` `// las expresiones no tienen dirección`
  - `&i=c;` `// las direcciones de las variables no se pueden cambiar`
  - `d=12345;` `// no se puede asignar una dirección absoluta directamente (tenemos que recurrir al casting): d=(int *)12345;`

Punteros: Asignación e inicialización

- Asignaciones entre punteros sin casting
  - Declaramos 2 punteros que apuntaran a objetos enteros
    - int *p,*q;
  - Asignamos también al puntero q la dirección almacenada en p
    - q=p;
- Inicialización de punteros
  - Es posible asignarle un valor inicial (preferentemente nulo) tal y como haríamos con una variable
    - float *p = NULL;
      - Null está definido en stdlib.h y representa la dirección 0 (0x0) de memoria. Para operaciones lógicas = false

Punteros: punteros a void

- No es posible asignaciones directas, sin casting, entre punteros que apuntan a distintos tipos de variables.
- Para resolver el problema: Punteros a void
  - Ejercicio 5 hoja 1 problemas resueltos

Punteros: Aritmética

- Aritmética de punteros

- Ejemplo 9: Hallar el valor de las variables a, b, c al final del programa

```
void main(void){
  int a,b,c;
  int *p1,*p2;

  p1 = &a;
  *p1 = 1;
  p2 = &b;
  *p2 = 2;
  p1 = p2;
  *p1 = 0;
  p2 = &c;
  *p2 = 3;
}
```

Resultado final:

a=1

b=0

c=3

Aritmética de punteros

- **1.** `ip+10;`
  - Produce otro puntero;
  - Si ip es un puntero al elemento m[j] de una matriz (dicho de otro modo: si `*ip == &m[j]`), el nuevo puntero apunta a otro elemento m[j+10] de la misma matriz, con lo que `*(ip+10) == &m[j+10]`
- **2.** `y = *ip+10;`
  - Añade 10 al objeto *ip (objeto referenciado por ip) y lo asigna a y
- **3.** `*ip += 1;`
  - Equivale a: *ip = *ip + 1. Incrementa en 1 el valor (Rvalue) del objeto referenciado por ip.
- **4.** `++*ip;`
  - Igual que el anterior: incrementa en 1 el valor del objeto referenciado por ip

Aritmética de punteros

- **5.** `++ip;`
  - El resultado es otro puntero. Equivale a ip = ip+1, es decir, incrementa en 1 el valor del puntero, con lo que el nuevo puntero señala a otra posición (ver caso 1.)
- **6.** `ip++;`
  - Igual que el caso anterior. Incrementa en 1 el valor del puntero.
- **7.** `(*ip)++`
  - Igual que el caso 4. Dado que el paréntesis tiene máxima precedencia y asocia de izquierda a derecha, incrementa en 1 el valor del objeto referenciado por ip. Observe que el paréntesis es necesario; sin él iríamos al paso 8

Aritmética de punteros

- **8.** `*ip++` o `*(ip++)`
  - Es decir, se realiza ip++;. Si ip es un puntero al elemento m[j] de una matriz, el resultado es un puntero al elemento m[j+1]. Después se tomaría la indirección, es decir, el resultado final sería el valor del elemento m[j+1].
- **9.** `*ip+1;`
  - El valor resultante es el de incrementar en 1 el valor del objeto apuntado por ip

Punteros: ejercicio propuesto 1

```
#include <stdio.h>

void main( void )
{
    int u = 3, v;
    int *pu; //puntero a entero
    int *pv; //puntero a entero

    pu = &u;
    v = *pu;
    pv = &v;
    printf("\nu=%d &u=%X  pu=%X *pu = %d", u, &u, pu, *pu);
    printf("\nv=%d &v=%X  pv=%X *pv = %d", v, &v, pv, *pv);
}
```

Entregar la salida por pantalla asignando el valor de memoria hex para &u=FF56 y &v=F67A

Ejercicio propuesto 1. Hoja 1 Ejercicios resueltos

Punteros: ejercicio propuesto 2

```
#include <stdio.h>
void main( void )
{
    int a=5,b,*pa,*pb;
    printf("La direccion de a= %X\n",&a);
    printf("El valor de a= %X\n",a);
    pa=&a; // puntero apunta a la direccion de a
            printf("La dirección del puntero pa es %X\n",&pa);
            printf("El puntero pa apunta a la dirección %X\n",pa);
            printf("El valor de la dirección de memoria a la que apunta el
                  puntero pa es %d\n\n",*pa);
    pb=pa; // el puntero pb apunta a la direccion del puntero pa
            printf("La dirección del puntero pb es %X\n",&pb);
            printf("El puntero pb apunta a la dirección %X\n",pb);
            printf("El valor de la dirección de memoria a la que apunta el
                  puntero pb es %d\n",*pb);
}
```

Entregar la salida por pantalla asignando el valor aleatorio de memoria en hex que creamos oportuno al inicio

Ejercicio propuesto 1. Hoja 1 Ejercicios resueltos

Funciones: Paso por dirección

- Ejercicio 1 : paso por dirección
  Hoja 1 ejercicios resueltos

Arrays

Arrays

- Definición:
  - modo de almacenar una serie de variables de forma consecutiva en memoria bajo un mismo nombre
- Declaración:
  - `tipo nombre[numero_de_elementos]`
    - `Ej: char ciudades[20]`
- Peculiaridades:
  - Los datos que se almacenan en un array deben ser siempre del mismo tipo
  - No se puede operar con todo el vector como entidad única, sino que hay que tratar sus elementos uno a uno a través de bucles (for, ...)
  - Para acceder a una determinada variable dentro de un array, se accede a través de un subindice (se comienza desde 0):
    - a[5]

Arrays: ejemplo 1

- *Ejemplo 1*

```
#include <stdio.h>
void main(void){
     char vector[20]="Madrid";
     printf ("%c", vector[0]);
}
```

El resultado mostrado por pantalla es: “M” ya que el vector quedará almacenado como:

vector [M,a,d,r,i,d]
0 1 2 3 4 5

Punteros y arrays

PUNTEROS Y ARRAYS

- El nombre de un vector es un puntero a la dirección de memoria que contiene el primer elemento del vector
  - Ej:

```
double vect[10];   //vect es un puntero a vect[0]

double *p;
p=&vect[0];      //p=vect
```


Repaso de lenguaje C: Punteros y arrays

Podemos acceder a los elementos del array de la siguiente manera:

Ejemplo 11

```
void main(void){
     int x[3];                        // array de 3 enteros
     int *puntero;
     x[0]=10;
     x[1]=20;
     x[2]=30;

     puntero = x;                     // metodo 1
     puntero = &x[0];                 // metodo 2 son equivalentes

     printf("%d\n\n",puntero[0]);     // Mostramos el elemento 0 del array
     printf("%d\n\n",*puntero);       // Mostramos el elemento 0 del array

     printf("%X\n\n",&puntero);       // Mostramos la posición en memoria del primer elemento del array
     printf("%X\n\n",&puntero[1]);    // Mostramos la posición en memoria del segundo elemento del array

     printf("%d\n\n",puntero[1]);     // Mostramos el segundo elemento del array
     printf("%d",*(puntero+1));       // Mostramos el segundo elemento del array

     printf("%d",*(puntero+2));       // Mostramos el tercer elemento del array


     system("pause");
}
```

Arrays : Arrays de caracteres

- Un array de caracteres se debe entender como un array de elementos de tipo char.
  - char cade[20]="hola";

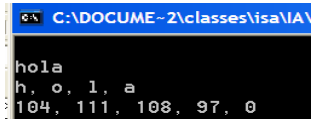
h	o	l	a	'\0'

- Para que el array de caracteres sea válido, su último elemento debe ser el carácter nulo '\0', que es el valor 0 en código ASCII. El compilador hace esto automáticamente.
- En cada uno de los elementos del array se almacena el código ASCII.

Ejemplo de Arrays

- Ejemplo: Obtener el código ASCII

```
char cade[20]="hola";
printf("\n%s", cade);
printf("\n%c, %c, %c, %c", cade[0],cade[1], cade[2],
  cade[3]);
printf("\n%d, %d, %d, %d, %d", cade[0],cade[1], cade[2],
  cade[3], cade[4]);
```



Matrices

- Definición:
  - Son similares a los vectores pero con dos dimensiones (filas y columnas) en lugar de una
- Declaración:

```
tipo nombre[numero_filas] [numero_columnas]
```

ej: char matriz [2] [3]