SISTEMAS OPERATIVOS

Introducción al Lenguaje C

ADVERTENCIA

- Este material es un simple guión de la clase: no son los apuntes de la asignatura.
- El conocimiento exclusivo de este material no garantiza que el alumno pueda alcanzar los objetivos de la asignatura.
- Se recomienda que el alumno utilice los materiales complementarios propuestos.

Bibliografía

- Problemas resueltos de programación en C
  F. García, J. Carretero, A. Calderón, J. Fernández, J. M. Pérez.
  Thomson, 2003.
  ISBN: 84-9732-102-2.

- El lenguaje de programación C. Diseño e implementación de programas
  J. Carretero, F. García, J. Fernández, A. Calderón
  Prentice Hall, 2001

Origen

- 1969-1973: Dennis Ritchie.
- 1973: Núcleo de UNIX se rescribe en C.
- 1983-1989: Estándar ANSI C. → ANSI C
- 1990: Estándar ISO 9899:1990. → C90
- 1999: Estándar ISO/IEC 9899:19999. → C99

Características de C (I)

- Es un lenguaje nativo, no es por scripts, no necesita nada para ejecutar.

- Lenguaje de propósito general ampliamente utilizado.
- Presenta características de bajo nivel:
  - C trabaja con la misma clase de objetos que la mayoría de los computadores (caracteres, números y direcciones)
- Estrechamente asociado con el sistema operativo UNIX:
  - UNIX y su software fueron escritos en C
- Es un lenguaje adecuado para programación de sistemas por su utilidad en la escritura de sistemas operativos.
- Es adecuado también para cualquier otro tipo de aplicación.

Características de C (II)

- Lenguaje pequeño
  - sólo ofrece sentencias de control sencillas y funciones.
- No ofrece mecanismos de E/S (entrada/salida)
  - Todos los mecanismos de alto nivel se encuentran fuera del lenguaje y se ofrecen como funciones de biblioteca (ej. stdio.h)
- Programas portables
  - pueden ejecutarse sin cambios en multitud de computadores.
- Permite programación estructurada y diseño modular.

Palabras reservadas en C (C99)

auto	else	long	typedef
break	enum	register	union
case	extern	return	unsigned
char	float	short	void
const	for	signed	volatile
continue	goto	sizeof	while
default	if	static	_Bool
do	inline	struct	_Complex
double	int	switch	_Imaginary

¿Por qué aprender C si ya sé Java?

- Lenguaje de alto nivel y de bajo nivel.
  - Interacción con el sistema operativo y controladores de dispositivos
- Mayor y mejor control de mecanismos de bajo nivel.
  - P. ej.: asignación de memoria.
- Mejor rendimiento que Java en muchos casos.
  - Java oculta muchos detalles necesarios para la interacción con el sistema operativo.
- Existe mucho código en C (o C++).
- Para ser "multilingüe"

Mucho más rapido, y ocupa menos => Pero más complejo

¿Qué se usa ahí fuera?

- Sistemas Operativos:
  - Windows: C/C++
  - Linux: C
  - MacOS: C/C++
  - Solaris: C
  - HP-UX: C
  - Google Chrome OS: C/C++
- Sistemas Operativos para móviles:
  - Symbian: C++
  - Google Android: C
  - RIM Blackberry: C++
- GUI
  - Microsoft Windows UI: C++
  - Apple Aqua: C++
  - Gnome: C/C++
  - KDE: C++
- Buscadores *Desktop*:
  - Google Desktop: C++
  - Microsoft Win Desktop: C++
  - Beagle: C++

¿Qué se usa ahí fuera?

- Ofimática:
  - Microsoft Office: C++
  - Sun Open Office: C/Java
  - Corel Office: C/C++
  - Adobe: C++
- Bases de datos:
  - Oracle: C/C++. Java
  - MySQL: C++
  - IBM DB2: C/C++
  - SQL Server: C++
  - IBM Informix: C/C++
  - SAP DB: C++
- Servidores Web:
  - Apache: C/C++
  - Microsoft IIS: C++
- Navegadores:
  - Internet Explorer: C++
  - Mozilla: C++
  - Safari: C++
  - Google Chrome: C++
  - Sun Hot Java: Java
  - Opera: C++
  - Mosaic: C
- Motores 3D:
  - DirectX: C++
  - OpenGL: C
  - OGRE 3D: C++

Desventajas

- C es más antiguo que Java.
- C no es orientado a objetos.
- C no es un lenguaje fuertemente tipado.
  - Es bastante permisivo con la conversión de datos.
- El lenguaje es portable, pero muchas bibliotecas son dependientes de la plataforma.
- La gestión de recursos no es automática.
  - Hay que devolver todo lo que se coge (por ej. la memoria)
- Se puede llegar a escribir código incomprensible.
- Sin una programación metódica puede ser propenso a errores difíciles de encontrar.
- La versatilidad de C permite crear programas difíciles de leer.

Ejemplo de código C


```
#define _ -F<00||--F-OO--;
int F=00,OO=00;
main(){F_OO();printf("%1.3fn",4.*-F/OO/OO);}F_OO()
{
                  _-_-_-_
              _-_-_-_-_-_-_-_-_
           _-_-_-_-_-_-_-_-_-_-_-_
         _-_-_-_-_-_-_-_-_-_-_-_-_-_
        _-_-_-_-_-_-_-_-_-_-_-_-_-_-_
        _-_-_-_-_-_-_-_-_-_-_-_-_-_-_
       _-_-_-_-_-_-_-_-_-_-_-_-_-_-_-_
       _-_-_-_-_-_-_-_-_-_-_-_-_-_-_-_
       _-_-_-_-_-_-_-_-_-_-_-_-_-_-_-_
       _-_-_-_-_-_-_-_-_-_-_-_-_-_-_-_
        _-_-_-_-_-_-_-_-_-_-_-_-_-_-_
        _-_-_-_-_-_-_-_-_-_-_-_-_-_-_
         _-_-_-_-_-_-_-_-_-_-_-_-_-_
           _-_-_-_-_-_-_-_-_-_-_-_
              _-_-_-_-_-_-_-_
                  _-_-_-_
}
```

Modelo de compilación de C

Compilación vs. interpretación

- Cuando se compila un programa en Java:
  - Se genera una representación intermedia bytecode.
  - Para ejecutar el programa, la máquina virtual debe interpretar cada instrucción del bytecode y traducir juego de instrucciones del procesador.
- Cuando se compila un programa en C:
  - Se genera un archivo ejecutable con instrucciones máquina.
  - Para ejecutar el programa no hace falta máquina virtual.
  - Para llevarlo a otra plataforma hay que volver a compilar.

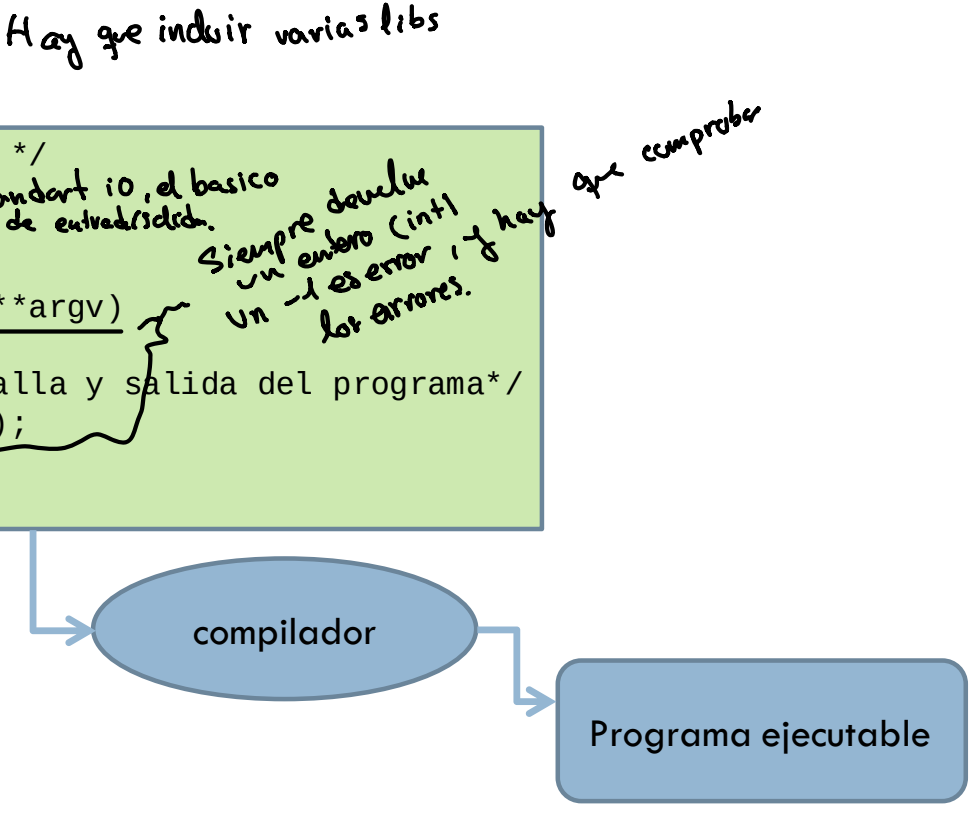
Utilización de bibliotecas

- En C no existe el concepto de paquete Java ni hay clausula import.
  - Las bibliotecas estándar se enlazan por defecto.
  - Hay que incluir en el programa la declaración de los tipos de datos y funciones que se usan.
  - La directiva #include copia el texto de un archivo un el punto de inclusión.
    - #include <archivo.h> → Inclusión de biblioteca del sistema.
    - #include "archivo.h" → Inclusión de biblioteca del usuario.

Hola mundo

hola.c

```
/* Inclusión de archivos */
#include <stdio.h>

/* Función principal */
int main (int argc,char **argv)
{
   /* Impresión por pantalla y salida del programa*/
   printf("Hola mundo\n");
   return 0;
}
```



El programa principal

- Un programa C tiene una única función main() que actúa de programa principal.
  - main devuelve un número entero.
    - 0 → Todo fue OK
    - No 0 → Ha habido un error

Impresión

- La función de biblioteca printf() permite imprimir textos en la pantalla.
  - Se encuentra declarada en el archivo de cabecera <stdio.h>
  - Hay que incluirla como fichero de cabecera

- En general, para cualquier función de cabecera que usemos en nuestros programas:
  - incluir la biblioteca (o librería) donde se encuentra implementada

La función scanf()

- Permite leer datos del usuario.
- La función devuelve el número de datos que se han leído correctamente.
- Formato:
  - scanf(formato, argumentos);
- Especificadores de formato igual que printf().
- Ejemplos:

```
scanf("%f", &numero);
scanf("%c", &letra);
scanf("%f %d %c", &real, &entero, &letra);
scanf("%ld", &entero_largo);
```

- Importante el operador **&.**

Especificadores de formato

Carácter	Argumentos	Resultado
d, i	entero	entero decimal con signo
u	entero	entero decimal sin signo
o	entero	entero octal sin signo
x,X	entero	entero hexadecimal sin signo
f	real	real con punto y con signo
e, E	real	notación exponencial con signo
g, G		
c	carácter	un carácter
s	cadena de caracteres	cadena de caracteres
%		imprime %
p	void	Dependiente implementación
ld,lu,lx,lo	entero	entero largo

Compilación

- Se puede utilizar cualquier compilador ANSI C.
- En clases y prácticas vamos a usar gcc.
- Alternativas:
  - Linux/Unix: Suele estar instalado.
  - Windows: Cygwin
- Ejemplos:
  - 1)

```
gcc -c -Wall hola.c
gcc hola.o -o hola
```

marca los warnings

Pasarlo a ejecutable.

Lo que sale la salida será hola

  - 2)

```
gcc hola.c -ansi -Wall -o hola
```

Errores

Intentar eliminar todos los warning, pueden desembocar en errores
En declaraciones ponerles valor

- Cuando se obtienen muchos errores de compilación unos dependen de otros.
  - gcc devuelve el número de línea donde aparece el error y una descripción
  - Arreglar en orden (primero el primer error) y volver a compilar.
- Si hay advertencias (warnings) pero no errores, gcc genera un ejecutable.
  - Peligroso. Un programa correcto no debe tener "warnings".

Errores comunes

- Problemas con las mayúsculas y minúsculas.
- Omisión del punto y coma.
- Comentarios incompletos.
- Comentarios anidados.
- Uso de variables no declaradas.

Sensible a?

↳ Siempre acaban con ;

Una línea es: //
Un bloque de coment: /* ... */

Declarar siempre y donde sea necesaria, dentro del bloque que la necesita (cuando la traiga a caché la trae cerca)

Comentarios

- Cualquier secuencia entre /* y */.
- Los comentarios del tipo // no son válidos en ANSI C.
- No se pueden anidar comentarios.

Tipos de datos numéricos

- Numéricos enteros:
  - char → 1 byte
  - int → 1 palabra (típico 4 bytes)
  - short, long, long long
  - Se admiten prefijos unsigned y signed.
    - Por defecto, todos son con signo.

- Numéricos reales:
  - float → Simple precisión.
  - double → Doble precisión.

En C no hay booleanos (C90)

- Se utilizan valores de los tipos numéricos enteros.
- Interpretación:
  - 0 → Falso
  - Cualquier otro valor (típicamente 1)→ Cierto
- Cuidado:

```
#include <stdio.h>
int main() {
  int x, y;
  x=1;          /* cierto */
  y=2;          /* cierto */
  if (x=y) { printf("Iguales\n"); }
  return 0;
}
```

argc = El propio comand.
argv[0] = El propio comando
argv[1] = los argumentos del comand.
2
:

Esto es una asignación y NO una comparación!

Declaración de variables

- Una declaración asocia un tipo de datos determinado a una o más variables.
- El formato de una declaración es:
  - `tipo_de_datos  var1, var2, ..., varN;`
- Ejemplos:

```
int a, b, c;
float numero_1, numero_2;
char letra;
unsigned long entero;
```

- Deben declararse todas las variables antes de su uso.

Constantes

- Dos alternativas
  - Utilizando el preprocesador (directiva #define)
    - Realiza una sustitución de texto en el archivo fuente antes de compilar.
  - Declarando una variable como constante (const)
    - `const int A = 4;`
    - Permite comprobación de tipos.

fuera de las funciones y main

Dentro de una función o main.

- La segunda opción es preferible, pero existe mucho código ya escrito que sigue usando la primera.

Constantes


```
#define MAX1 100
const int MAX2 = 100;

int main() {
  float v[MAX1];
  float w[MAX2]
  ...
  return 0;
}
```

Operadores

x++ x–	Postincremento/Postdecr.
++x --x +x -x	Postincremento/Postdecr.
!	No lógico
(tipo)expr	Creación o conversión de tipo
* / %	Mult.,división, módulo
+ -	Suma, resta
<< >>	Desplazamiento binario
< <= >= >	Comparaciones de orden
== !=	Test de igualdad
&	AND binario
|	OR binario
&&	AND lógico
||	OR lógico
= += -= *= /= %= &= |= <<= >>= >>>=	Asignación

Ejemplo: Errores comunes


```
#include <stdio.h>
#define PI    3.141593

      /* Programa que calcula el area de
         un circulo
int main()
{
   float radio;

   printf("Introduzca el radio: ")
   scanf("%f",&radio);

   area = PI * radio * Radio;
   printf("El area del circulo es %5.4f \n", area);
}  return 0;
```

Conversión de tipos

- En C un operador se puede aplicar a dos variables o expresiones distintas.
  - Los operandos que difieren en tipo pueden sufrir una conversión de tipo.
  - **Norma general**: El operando de menor precisión toma el tipo del operando de mayor precisión.

Conversión de tipos o *casting*

- Se puede convertir una expresión a otro tipo:

```
(tipo datos) expresion
```

- Ejemplo:

```
( (int) 5.5 % 4)
```

Operadores de asignación

- Forma general:

```
identificador = expresion
```

- El operador de asignación = y el de igualdad == son **distintos**
- Asignaciones múltiples:

```
id_1 = id_2 = ... = expresion
```

- Las asignaciones se efectúan de derecha a izquierda.
- En i = j = 5
  - A j se le asigna 5
  - A i se le asigna el valor de j

Reglas de asignación

- *Si en una sentencia de asignación los dos operandos son de tipos distintos, entonces el valor del operando de la derecha será automáticamente convertido al tipo del operando de la izquierda*. Además:
  - Un valor en coma flotante se puede truncar si se asigna a una variable de tipo entero.
  - Un valor de doble precisión puede redondearse si se asigna a una variable de coma flotante de simple precisión.
  - Una cantidad entera puede alterarse si se asigna a una variable de tipo entero más corto o a una variable de tipo carácter.
- Es importante en C utilizar de forma correcta la conversión de tipos.

Sentencias de control

- Sentencia if
- Sentencia if-else
- Sentencia for
- Sentencia while
- Sentencia do-while
- Sentencia siwtch
  switch

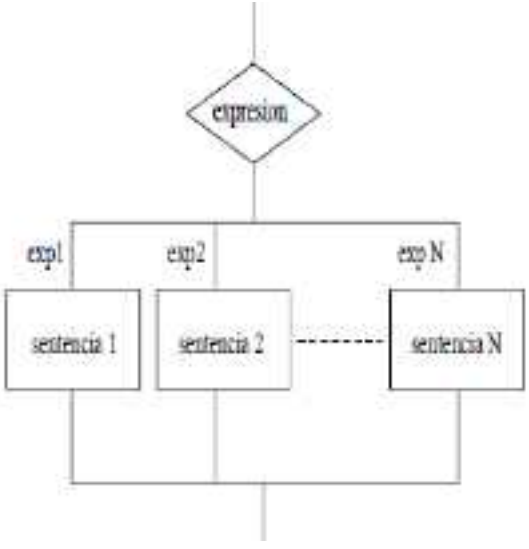
Estructuras de control

- Estructuras de evaluación de condición: if y switch
  - Las condiciones son expresiones de tipo entero.
  - Cuidado → if (x=1) { printf("siempre\n"; }
    ==
- Estructuras de repetición: while, do-while, for
  - El índice for se debe declarar como otra variable más:
    no dentro del for

```
int main() {
  int i;
  for (i=0;i<10;i++) {
    printf("hola\n");
  }
  return 0;
}
```

Sentencia switch

```
switch (expresion)
{
   case exp 1:
      sentencia 1;
      sentencia 2;
         .
         .
      break;

   case exp N:
   case exp M:
      sentencia N1;
      sentencia N2;
         .
         .
      break;

   default:
      sentencia D;
         .
}
```



Arrays

- Un array es un conjunto de datos del mismo tipo a los que se da un nombre común.
- La declaración de variable es muy similar:
  - `int v[10];` → v es un `array` de 10 enteros (de 0 a 9)
  - `double x[10][20]` → x es una matriz de 10x20 double.
  - El índice solo puede variar entre 0 y n-1 (n=elementos del array)
- Diferencias:
  - En C no hay `new` (existe `malloc`) ni `length`.
  - El tamaño del `array` debe ser un valor conocido antes de compilar el programa.
  - No hay comprobación de límites.

```
v[11] = 1; /* Error, pero puede no ser detectado */
```

Arrays en acción


```
int main() {
  int v[100];
  double w[] = { 1.0, 3.5, 7.0 };          /* Tamaño = 3 */
  float y[];                    /* ERROR. Necesita tamaño */

  v[0] = 3;
  v[10] = v[0];
  v[-1] = 0; /* Comp. No definido */
  v[100]= 17; /* Comp. No definido */

  return 0;
}
```

Tipos enumerados

- Tipos definidos por el usuario con una enumeración finita de valores.
  - Tratamiento similar a los enteros.
  - No se pueden imprimir.

```
enum color { rojo, verde, azul };

int main() {
  color c = rojo;
  c = azul;

  return 0;
}
```

Estructuras

- Lo más parecido a una clase (pero sin métodos).
- Una estructura tiene un conjunto de campos con un tipo asociado a cada uno.

```
struct punto {
  double x;
  double y;
};
struct circulo {
  struct punto origen;
  double radio;
};
```

creación
campo 1
campo 2

```
int main() {
  struct circulo c1;
  c1.origen.x = 0.0;
  c1.origen.y = 5.0;
  c1.radio = 10.0;
  ...
  return 0;
}
```

asignación de una estructura
Asignación de valor para un parametro de la struct

Definición de tipos

- Se puede definir un sinónimo de un tipo mediante typedef.

```
typedef float medida;
medida x = 1.0;

struct punto {
  double x;
  double y;
};
typedef struct punto punto_t;

struct circulo {
  punto_t origen;
  double radio;
};
typedef struct circulo circulo_t;
```

Funciones

- En C no hay clases ni métodos, solamente hay funciones.
  - Una función acepta un conjunto de parámetros [0,n] y devuelve un resultado [0,1].
  - El paso de parámetros es siempre por valor (se copian).
  - No hay tratamiento de excepciones.
  - No se puede definir una función dentro de otra.
  - Una función tiene que haberse declarado antes de usarse.
  - No se pueden sobrecargar los nombres de las funciones.

Definición de una función


```
tipo nombre(tipo1 arg1, ..., tipoN argN)
{
   /* CUERPO DE LA FUNCION */
}
```

- Una función devuelve un valor de tipo `tipo`:
  - Si se omite tipo se considera que devuelve un `int`.
  - Si no devuelve ningún tipo: `void`.
- Una función acepta un conjunto de argumentos:
  - Si no tiene argumentos: `void` (o simplemente ())
  - `Ej: void explicacion(void)`
- La última sentencia de una función es `return valor;`
  - finaliza la ejecución y devuelve valor a la función que realizó la llamada.

Prototipo de una función

- Un prototipo de una función es la declaración de la función.

  La crear la cabecera, pero no el contenido.

  - Permite la comprobación de errores entre la llamada a una función y la definición de la función correspondiente.

```
float potencia (float x, int y); /* prototipo */
float potencia (float x, int y)  /* definicion */
{
   int i;
   float prod = 1;
   for (i = 0; i < y; i++)
      prod = prod * x;
    return(prod);
}
```

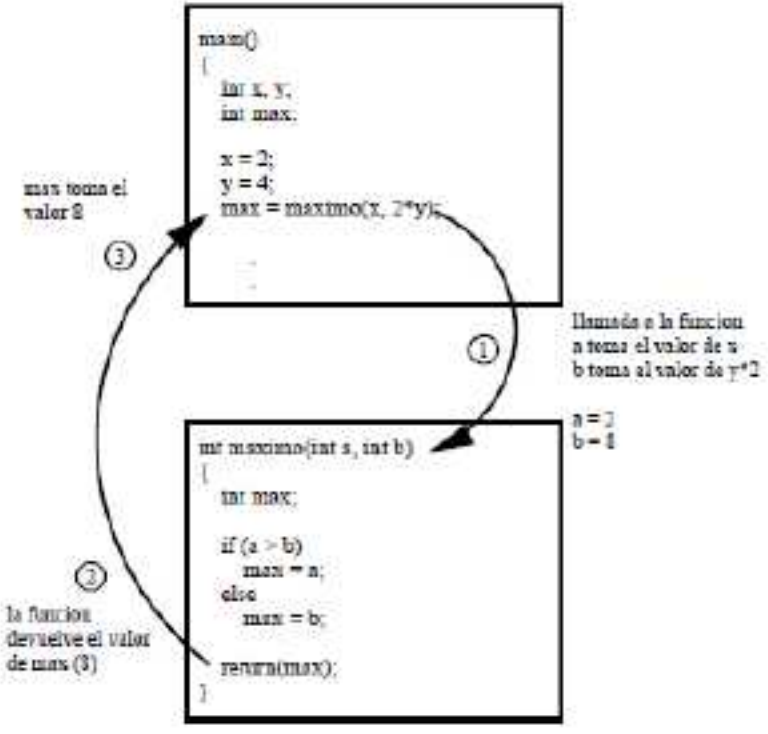
Ejemplo: función suma


```
#include <stdio.h>

int suma (int a, int b);   /* Prototipo */

int main() {
  int x, y;
  x=3;
  y = suma(x,2);           /* Llamada a la función */
  printf("%d + %d = %d\n", x, 2, y);
  return 0;
}

int suma (int a, int b) {  /* Definición */
  return a+b;
}
```

Parámetros de una función

- Parámetros formales:
  - aparecen en la definición de la función.
  - Se puede modificar el argumento formal dentro de la función, pero el valor del argumento real no cambia
    - **paso de argumentos por valor**
- Parámetros reales:
  - Se pasan en la llamada a la función.
  - Los parámetros reales pueden ser:
    - Constantes.
    - Variables simples.
    - Expresiones complejas
- Cuando se pasa un valor a una función **se copia** el argumento real en el argumento formal.

Proceso de una llamada



Variables locales y globales

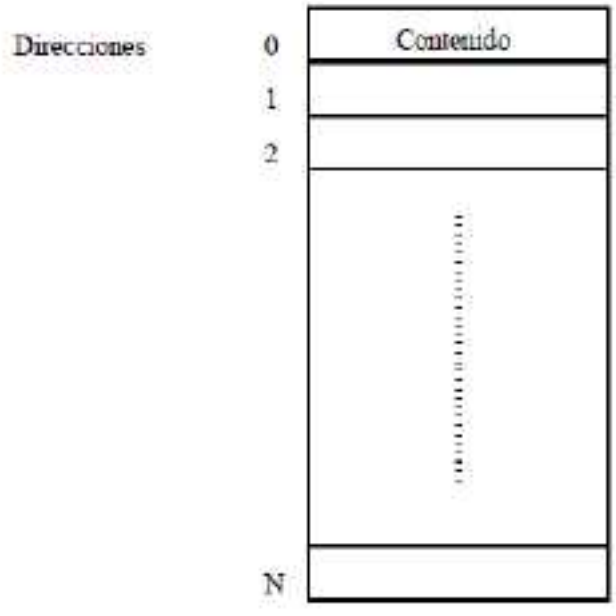
- Las variables que se declaran dentro de una función, son locales a esa función.
  - Existen solamente durante la ejecución de la función.
  - Se almacenan en la pila.

- Las variables que se declaran fuera de las funciones, son globales.
  - Se puede acceder a ellas desde cualquier función.
  - Poco deseable, pero necesario a veces.

Punteros. Introducción

- La memoria del computador se encuentra organizada en grupos de bytes que se denominan palabras.
- Dentro de la memoria cada dato ocupa un número determinado de bytes:
  - Un char: 1 byte.
  - Un int: 4 bytes (típicamente)
- A cada byte o palabra se accede por su dirección.
- Si X es una variable que representa un determinado dato el compilador reservará los bytes necesarios para representar x
  - 4 bytes si es de tipo int

Organización de la memoria



Punteros

- Un puntero es una variable que almacena la dirección de otro objeto (ej. una variable, una función, etc.)

```
int *a;
```

a ahora será un puntero y para acceder a su valor se hara con *a

```
int a
```

a es el valor y su dirección es &a

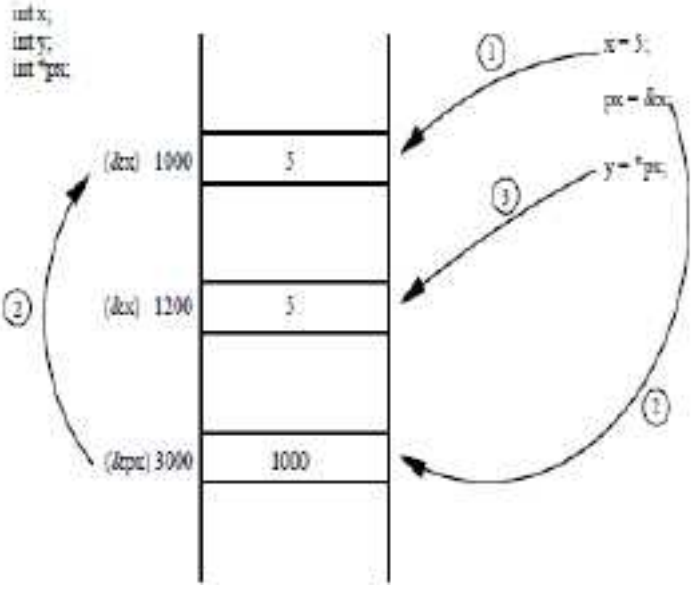
Declaración de punteros


```
tipo_dato *variable_ptr;
```

- `variable_ptr` es el nombre de la variable puntero.
- `tipo_dato` se refiere al tipo de dato apuntado por el puntero.
- `variable_ptr` sólo puede almacenar la dirección de variables de tipo `tipo_dato`.
- Para usar un puntero se debe estar seguro de que apunta a una dirección de memoria correcta (en otro caso, error).
- Ejemplos:
  - `int *numero;`
  - `float *p;`
  - `char *letra;`

Ejemplo


```
#include <stdio.h>

main()
{
   int x;     /* variable de tipo entero */
   int y;     /* variable de tipo entero */
   int *px;   /* variable de tipo
                 puntero a entero */

   x = 5;
   px = &x; /* asigna a px la direccion de x */
   y = *px; /* asigna a y el contenido de la
               direccion almacenada en px */

   printf("x = %dn", x);
   printf("y = %dn", y);
   printf("*px = %dn", *px);
}
```



Punteros

- Si px es un puntero:
  - *px representa el contenido almacenado en la dirección de memoria a la que apunta px
    - * es el **operador de indirección**
- Si x es una variable:
  - &x representa la dirección de memoria de x.
    - & es el **operador de dirección**.

- Un puntero contiene la dirección de memoria de la variable y NO su valor:

Declaración de punteros

- En:

```
char *p;
char letra;
letra = 'a';
p = &letra;
*p = ‘b’;
```

  - ¿Cuántos bytes ocupa la variable p?
  - ¿Cuántos bytes ocupa la variable letra?
  - ¿Es correcto el código anterior?

Ejemplo

```
float n1;
float n2;
float *p1;
float *p2;

n1 = 4.0;
p1 = &n1;

p2 = p1;
n2 = *p2;
n1 = *p1 + *p2;
```

- ¿Cuánto vale `n1` y `n2`?

Puntero NULL

- Cuando se asigna 0 a un puntero, este no apunta a ningún objeto o función.
- La constante simbólica NULL definida en stdlib.h tiene el valor 0 y representa el puntero nulo.
- Es una buena técnica de programación asegurarse de que todos los punteros toman el valor NULL cuando no apuntan a ningún objeto o función.
  - `int *p = NULL;` → Antes de darle valor, ponerlo a null (0)
- Para ver si un puntero no apunta a ningún objeto o función:

```
if (p == NULL)
      printf("El puntero es nulo\n");
   else
      printf("El contenido de *p es\n", *p);
```

Punteros y paso de parámetros

- Los punteros se pueden usar para simular el paso de parámetros por referencia:
  - No se pasa una copia sino la dirección del dato al que apunta
  - Parámetro por referencia → Se declara de tipo puntero
  - Dentro de la función se usa el puntero.
  - Fuera de la función se pasa la dirección de la variable.

Así se puede modificar directamente el contenido de la dir

- El uso de punteros como argumentos de funciones permite que el dato sea alterado globalmente dentro de la función.

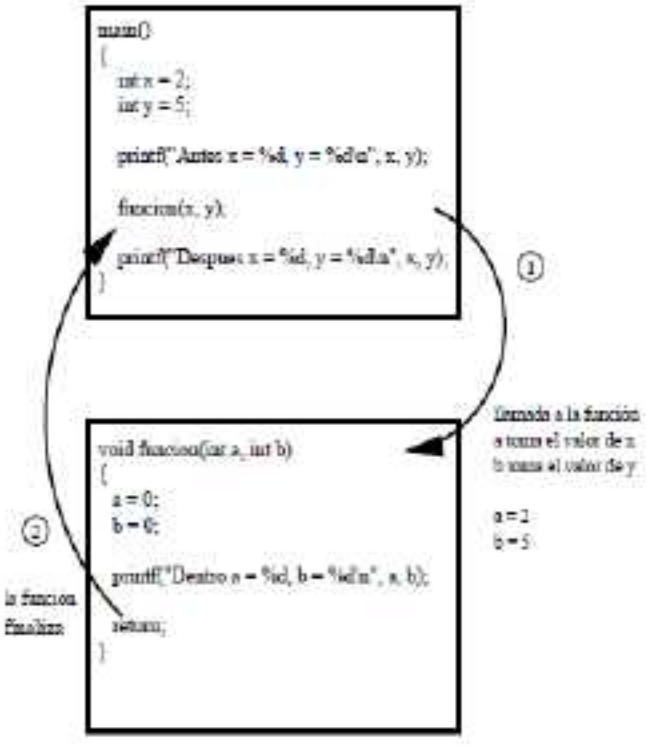
Punteros y paso de parámetros


```
#include <stdio.h>

void acumula(int * s, int x) {    /* Puntero s*/
  *s += x;
}

int main() {
  int suma = 0;
  int i;
  for (i=0;i<10;i++) {
    acumula(&suma,x);      /* Dirección de suma */
  }
  printf("La suma de 1 a 9 es %d\n",suma);
}
```

Ejemplo


```
#include <stdio.h>

void funcion(int a, int b); /* prototipo */

main() {
   int x = 2;
   int y = 5;

   printf("Antes  x = %d, y = %d\n", x, y);

   funcion(x, y);

   printf("Despues  x = %d, y = %d\n", x, y);
}

void funcion(int a, int b){
   a = 0;
   b = 0;

   printf("Dentro  a = %d, b = %d\n", a, b);
   return;
}
```

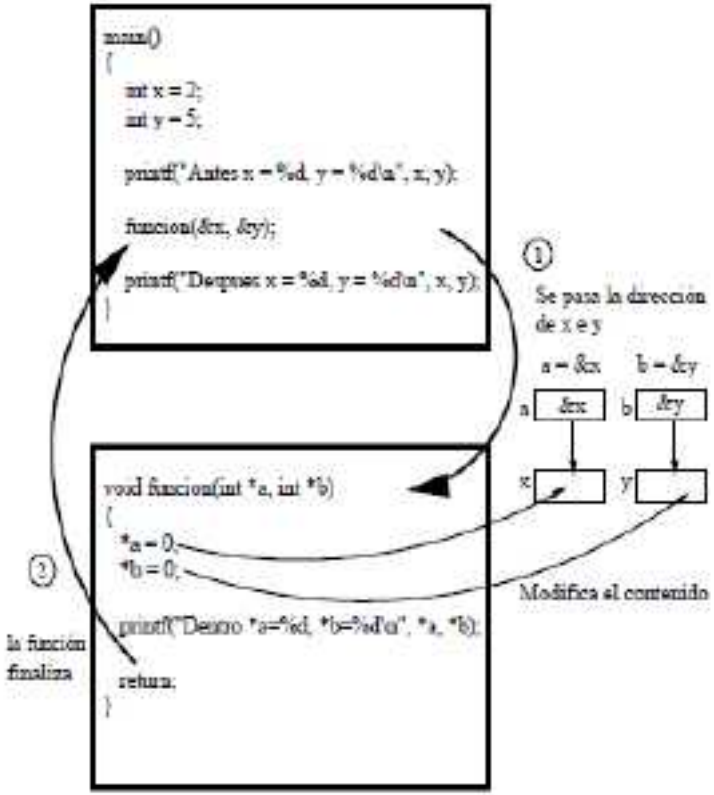
Proceso de llamada



Paso de parámetros por referencia


```
#include <stdio.h>

void funcion(int *a, int *b); /* prototipo */

main(){
   int x = 2;
   int y = 5;

   printf("Antes  x = %d, y = %d\n", x, y);

   funcion(&x, &y);

   printf("Despues  x = %d, y = %d\n", x, y);
}

void funcion(int *a, int *b){
   *a = 0;
   *b = 0;

   printf("Dentro  *a = %d, *b = %d\n", *a, *b);

   return;
}
```

Proceso de llamada



Consideraciones

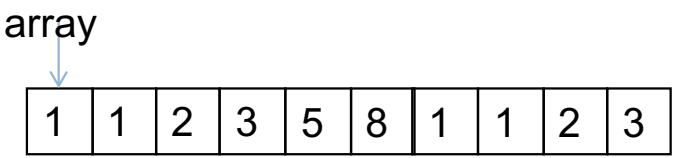
- Un puntero no reserva memoria
  - La declaración de un puntero solo reserva memoria para la variable de tipo puntero
- En:

```
int *p;
*p = 5;
```

  - ¿A qué dirección de memoria apunta p?
  - ¿Dónde se almacena el valor 5?

Punteros y vectores

- Un array de una dimensión es equivalente a un puntero que apunta a la primera posición del array.
  - int array[10]
  - &array[0]= array
- Típicamente una función que trabaja con un array necesita tomar un puntero y una variable entera que indique su tamaño.



Suma de un array


```
#include <stdio.h>

double suma_array(double * v, int n) {
  int i;
  double s = 0.0;
  for (i=0;i<n;i++) {
    s += v[i];
  }
  return s;
}

int main() {
  double w[10];
  for (int i=0;i<10;i++) { w[i] = 1.0 * i; }
  printf("suma %lf\n", suma_array(w,10));
  return 0;
}
```

Aritmética de punteros

- Se pueden hacer operaciones aritméticas entre un puntero y un entero.
  - Suma o resta del puntero con el entero multiplicado por el tamaño del tipo base.

```
int v[10];
int * p = v; /* equivale a p = &v[0] */
p = p + 2;   /* p apunta a v[2] */
p+= 3;      /* p apunta a v[5] */
p--;         /* p apunta a v[4] */
*p = 5;      /* v[4] = 5 */
```

Suma de un array (toma 2)


```
#include <stdio.h>

double suma_array(double * v, int n) {
  double * fin = v + n;
  double s = 0.0;
  for (;v!=fin;v++) {
    s += *v;
  }
  return s;
}

int main() {
  double w[10];
  for (int i=0;i<10;i++) { w[i] = 1.0 * i; }
  printf("suma %lf\n", suma_array(w,10));
  return 0;
}
```

Cadenas de caracteres

- Las cadenas en C, se representan como un `array` de `char`.
  - Un carácter en cada posición.
  - Marca de fin de cadena: carácter con código 0 (representado por `'\0'`).
  - Tamaño: Número de caracteres más uno.
  - Funciones más usuales (incluidas en string.h):
    - `strlen()`: Para conocer el tamaño de una cadena
    - `strcat()`: concatena dos cadenas
    - `strcpy()`: copia una cadena en otra
    - `strchr()`: encuentra la primera ocurrencia de un carácter en una cadena
    - `strstr()`: busca una cadena en otra
    - `sprintf()`: permite introducir datos formateados en una cadena (Ej: int => string)
    - `sscanf()`: permite extraer datos de una cadena (Ej: string => int)

Acaban con \0 = null

`char cadena[] = "hola";`

Cadenas

- Se debe distinguir el tamaño de una cadena de su longitud.
  - Tamaño: Número de caracteres que puede almacenar el array.
  - Longitud: Número de caracteres que hay desde el principio hasta la marca de fin.

Ejemplos


```
char c1[5] = { 'h', 'o', 'l', 'a', 0 };
char c2[5] = "hola";
char c3[] = "hola";
char c4[100] = "hola";
char *c5 = c2;      /* c5 referencia la cadena c2 */
c5 = c1;            /* c5 referencia ahora la cadena c1 */
c3 = c2;            /* ERROR */
```

Espacio solo para hola\0

Espacio para 100 caracteres en los que esta hola\0

puntero

puntero

Contar apariciones


```
int contar(char * c, char x) {
  int n=0;
  char * q = c;
  while (q!=NULL) {
    if (*c==x) {
      n++;
    }
   }
   return 0;
}
```

Punteros genéricos

- Existe un tipo de puntero general.
  - void * p;
- Reglas:
  - Se puede copiar un puntero a un puntero genérico.
    - char * q = hola;
    - p = q;
  - No se puede desreferenciar un puntero genérico.
    - *p = 'x'; /* ERROR */
  - Se puede copiar un puntero genérico a un puntero, pero hace falta aplicar una conversión.
    - q = (char*)p;

Memoria dinámica

- Permite reservar memoria en el montículo (heap).

- Operaciones:
  - Reservar memoria (equivalente a new).
    - p = (tipo*) malloc(nbytes);
  - Liberar memoria (en C NO ES AUTOMÁTICO)
    - free(p)

Operador sizeof

- Operador que permite conocer el tamaño (en bytes) de cualquier tipo de datos.
  - sizeof(char) → 1
  - long a;
    sizeof(a) → 4

Ejemplo


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

double promedio(double * w, int n) {
  double s = 0.0;
  int i;
  for (i=0;i<n;i++) {
    s += w[i];
  }
  return s/n;
}
```

```
int main() {
  int n,i;
  double * v;
  printf("nro de valores=");
  scanf("%d", &n);
  v = (double*) malloc(sizeof(double)*n);
  for (i=0;i<n;i++) {
    printf("v[%d]=",i);
    scanf("%lf",&v[i]);
  }
  printf("Promedio=%lf\n", promedio(v,n));
  free(v);
  return 0;
}
```

Una cola de peticiones

- Se desea tener una cola de peticiones que se va a usar en un servidor Web.
- Se consideran peticiones de 2 tipos: GET y PUT.
- Todas las peticiones tienen una URL asociada de longitud arbitraria.

peticion.h


```
#ifndef PETICIONES_H
#define PETICIONES_H

enum tipo_peticion { PET_GET, PET_PUT };
typedef enum tipo_peticion tipo_peticion_t;
struct peticion {
  tipo_peticion_t tipo;
  char * url;
};
typedef struct peticion peticion_t;

peticion_t * crea_peticion(tipo_peticion_t p, char * url);
void destruye_peticion(peticion_t * p);
void copia_peticion(peticion_t * d, peticion_t * o);

#endif
```

peticion.c


```
#include "peticion.h"
#include <stdlib.h>
#include <string.h>

peticion_t * crea_peticion(tipo_peticion_t t, char * url) {
  peticion_t * p;
  p = malloc(sizeof(peticion_t));
  p->tipo = t;
  p->url = malloc(sizeof(char)*(strlen(url)+1));
  strcpy(p->url, url);
  return p;
}

void destruye_peticion(peticion_t * p) {
  free(p->url);
  free(p);
}

void copia_peticion(peticion_t * d, peticion_t * o) {
  d->tipo = o->tipo;
  free(d->url);
  d->url = malloc(sizeof(char)*(strlen(o->url)+1));
  strcpy(d->url,o->url);
}
```

cola.h


```
#ifndef COLAPET_H
#define COLAPET_H

#include "peticion.h"

struct cola_peticiones {
  peticion_t ** cola;
  int tam;
  int inicio, fin;
};
typedef struct cola_peticiones cola_peticiones_t;

cola_peticiones_t * crea_cola(int max);
void destruye_cola(cola_peticiones_t * c);
void pon_peticion(cola_peticiones_t * c, peticion_t * p);
peticion_t * saca_peticion(cola_peticiones_t * c);


#endif
```

cola.c

- Piense en una posible implementación.

Lectura recomendada

- Cómo NO Hacer unas prácticas de programación:
  - http://di002.edv.uniovi.es/~cernuda/pubs/jenui2002-2.pdf

SISTEMAS OPERATIVOS

Introducción a Lenguaje C