

Programación de Sistemas y Concurrency

Tema 2: Programación en el Lenguaje C

Grado en Ingeniería Informática

Grado en Ingeniería del Software

Grado en Ingeniería de Computadores

Índice de contenidos

- El lenguaje C. Introducción
- E/S
- Control de flujo de ejecución
- Tipos de datos:
 - Tipo de datos simples, estructurados y el tipo puntero
- Subprogramas: procedimientos y funciones
- Gestión de memoria dinámica
- Programación modular
- Persistencia de datos: Ficheros
- Operaciones de bajo nivel

Historia de C. Versiones

1969

- Nacimiento del SO UNIX
- Ordenador: PDP7

1978

- The C Programming Language
- K&R C

1999

- ISO C (C99)
 - Tipo complex
 - Arrays de longitud variable
 - ...

1973

- Ordenador: PDP-11
- Lenguaje C
- UNIX escrito en C

1989

- ANSI C (C89)
 - Prototipos de funciones
 - Punteros void
 - ...

2011

- C11
 - Multi-threading
 - Operaciones atómicas
 - ...

Características principales

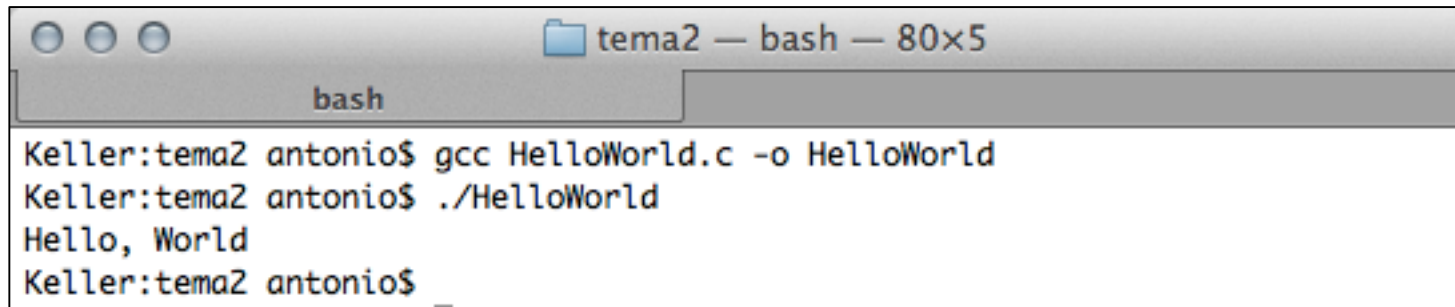
- Lenguaje de alto nivel no orientado a objetos
- Muy eficiente: características de bajo nivel
- Sistema de tipos débil
- Preprocesador (macros, constantes)
- Acceso directo a memoria (punteros)
- Conjunto reducido de palabras clave
- Tipos de datos estructurados: arrays, estructuras y uniones

Ejemplo: HolaMundo.C

```
/*  
 * Program: HelloWorld.c  
 * Description: Implements a program displaying "Hello, World"  
 */  
  
#include <stdio.h>  
  
int main(int argc, char *argv[]) {  
    printf ("Hello, World\n");  
  
    return 0;  
}
```

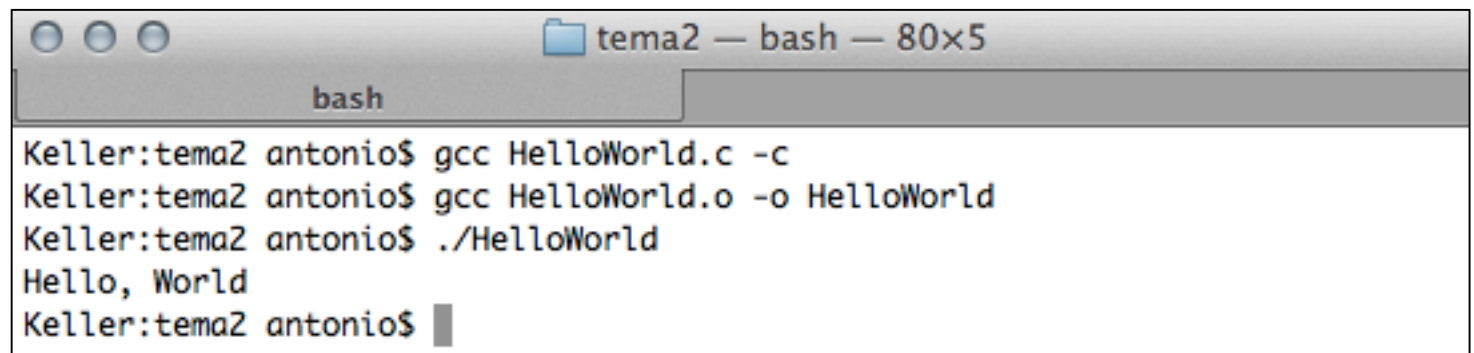
Fases de compilación

- Preprocesado
- Compilación
- Enlazado (linking)



A terminal window titled 'tema2 — bash — 80x5' with a 'bash' tab. It shows the following commands and output:

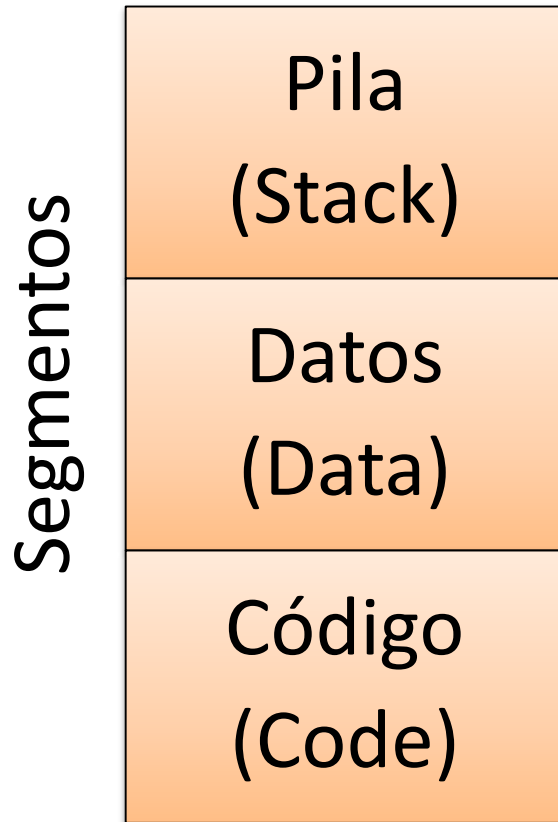
```
Keller:tema2 antonio$ gcc HelloWorld.c -o HelloWorld
Keller:tema2 antonio$ ./HelloWorld
Hello, World
Keller:tema2 antonio$ _
```



A terminal window titled 'tema2 — bash — 80x5' with a 'bash' tab. It shows the following commands and output:

```
Keller:tema2 antonio$ gcc HelloWorld.c -c
Keller:tema2 antonio$ gcc HelloWorld.o -o HelloWorld
Keller:tema2 antonio$ ./HelloWorld
Hello, World
Keller:tema2 antonio$ █
```

Modelo de memoria de un proceso



- Pila
 - Parámetros de función
 - Variables locales de función
- Datos
 - Datos globales
 - Montículo (heap): memoria dinámica (malloc)
- Código
 - Código ejecutable
 - Segmento de sólo lectura

Índice de contenidos

- El lenguaje C. Introducción
- E/S
- Control de flujo de ejecución
- Tipos de datos:
 - Tipo de datos simples, estructurados y el tipo puntero
- Subprogramas: procedimientos y funciones
- Gestión de memoria dinámica
- Programación modular
- Persistencia de datos: Ficheros
- Operaciones de bajo nivel

E/S estándar

- La mayor parte de las funciones de E/S en C se encuentran en **<stdio.h>**
- La función de salida más utilizada es `printf`
 - Esta función traduce las variables a caracteres

```
int printf(char *format, arg1, arg2, argn)
```

 - `printf` convierte, da formato e imprime los argumentos en la salida estándar bajo el control de `format`.


E/S estándar

- La cadena `format` tiene dos elementos
 - Caracteres ordinarios que se mostrarán tal cual
 - Especificaciones de conversión. Comienzan con % y terminan con un carácter de conversión.
 - Entre % y el carácter de conversión se puede incluir
 - Signo menos -> ajuste a la izquierda del argumento convertido
 - Número que especifica el ancho mínimo del campo
 - Un punto que separa el ancho del campo de la precisión
 - Un número, la precisión, número máximo de caracteres que serán impresos

E/S estándar

- Caracteres de conversión más utilizados

Carácter	Impreso como
d,i,l	Número decimal
O	Número octal
X,x	Número hexadecimal
u	Entero sin signo
c	Carácter
s	Cadena de caracteres
f,e,E,g,G	double



Hay más conversiones
Búscalos!!

E/S estándar

- Ejemplo

```
int s=10;  
printf("El valor de la variable  
s es =%d\n", s),
```

```
char cadena[5]="hola";  
printf("Valor de cadena= %s\n",  
cadena);
```

E/S estándar

- La función `sprintf` realiza las mismas conversiones que `printf` pero almacena la salida en una cadena

```
int sprintf(char *cadena, char *formato,  
            arg, arg2, arg3,...) ;
```

E/S estándar

- La función `scanf` es la entrada análoga a `printf` y proporciona las mismas facilidades de conversión en la dirección opuesta

```
int scanf(char *format, arg1, arg2...);
```


- Los argumentos deben ser punteros a las variables!!!!
- Indican dónde debe almacenarse la entrada correspondiente convertida

E/S estándar

- Ejemplo: Sumador sencillo

```
#include <stdio.h>

int main () {
    double sum,v;
    sum=0
    while (scanf("%lf",&v)==1)
        printf("\t%.2f\n",sum+=v) ;
    return 0;
}
```



Algunas
conversiones y
formatos son
nuevas.
Búscalos!!

E/S estándar

- Hay otras funciones de E/S para usarlas con caracteres: `getchar`, `putchar`

```
main() {  
    int c;  
    while ( (c=getchar()) != '\f' ) {  
        putchar(c);  
    }
```

- ¿Cómo lo mostrarías usando `printf`?

E/S estándar

- La función `fflush(stdout)` permite vaciar el buffer intermedio cada vez que se realiza una llamada de E/S.
- Se puede utilizar cada vez que se realiza una operación de E/S
 - `fflush(stdout);`
- Si no la utilizas, y el programa “muere” o se bloquea, es posible que no veas algunos mensajes previos hechos con `printf`

Un ejemplo curioso

- Ejemplo: Programa Syscall.c

```
/**
 * Syscall.c
 */

#include <stdio.h>
#include <unistd.h>

int main(int argc, char ** argv) {
    printf("Hola, mundo") ; /* funcion C de E/S */
    write(1, "Adios, mundo", 12) ; /* llamada al sistema */
}
```

Índice de contenidos

- El lenguaje C. Introducción
- E/S
- **Control de flujo de ejecución**
- Tipos de datos:
 - Tipo de datos simples, estructurados y el tipo puntero
- Subprogramas: procedimientos y funciones
- Gestión de memoria dinámica
- Programación modular
- Persistencia de datos: Ficheros
- Operaciones de bajo nivel

Estructuras Selectivas

```
if (CondControl) {  
    accionesSI  
}  
accionseguida
```

CondControl es “cierto” si es
distinto de cero

No existen constantes true o
false

```
if (CondControl) {  
    accionesSI  
}  
else {  
    accionesEOC  
}  
accionseguida
```

Estructuras Selectivas

```
if (CondControl) {  
    accionesSI  
}  
else if (CondControl1) {  
    accionesCC1  
}  
else if (CondControl2) {  
    accionesCC2  
}  
else {  
    accioneselse  
}  
accionseguida
```

Estructuras Selectivas

```
switch (expresion) {  
    case exp-const: acciones  
                        break;  
    case exp-const: acciones  
                        break;  
    case exp-const: acciones  
                        break;  
    default           : acciones  
                        break;  
}
```

Estructuras Repetitivas

```
while (expresion) {  
    acciones  
}
```

```
for (expr1;expr2;expr3) {  
    acciones  
}
```

```
do {  
    acciones  
} while (expresion);
```

Gestión de errores

- En C, la gestión de errores la ha de hacer el programador
 - Típicamente, comprobando los códigos de error de las funciones a las que llamamos

Ej:

```
if ((fent = fopen("fichero.txt", "rt")) == NULL) {  
    perror("Error abriendo fichero.txt");  
}
```

Primera aparición
de NULL (0)



- La variable `errno` contiene un número indicando el último error registrado (en funciones de E/S o llamadas al sistema)
 - Función `perror(char * mensaje)`
 - Imprime el mensaje seguido del error asociado a la variable `errno`

Índice de contenidos

- El lenguaje C. Introducción
- E/S
- Control de flujo de ejecución
- Tipos de datos:
 - Tipo de datos simples, estructurados y el tipo puntero
- Subprogramas: procedimientos y funciones
- Gestión de memoria dinámica
- Programación modular
- Persistencia de datos: Ficheros
- Operaciones de bajo nivel

Tipos de datos simples

Tipo	Descripción	Tamaño (bytes)
char	Byte	1
int	Entero	4
float	Flotante en single precisión	4
double	Flotante en doble precisión	8
short/long int	Entero corto/largo	2/8
unsigned char	Número positivo	1
signed char	Número con signo	1
unsigned int	Entero positivo	4
long double	Flotante con precisión extendida	12

Tipos de datos simples

- Si las variables no se inicializan, su valor es indefinido
- La declaración de variables debe realizarse al inicio de los bloques de instrucciones
- Función `sizeof` para obtener el tamaño de los tipos en un sistema específico
 - Ej: `sizeof(int)`

Tipos de datos estructurados

Arrays

- Declarando arrays

- Unidimensionales

Tipo nombre [tam];

Ej: double saldo[10];

- Varias dimensiones

Tipo nombre [tam1][tam2]...[tamN];

Ej: double tresd[5][10][4];

Tipos de datos estructurados

Arrays

- Inicializando arrays

Ej: `double p[3]={1.0, 2.0, 3.0};`

Se puede omitir el tamaño inicial:

`double p[]={1.0, 2.0, 3.0};`

Ej: `int c[3][2]={ {0,0}, {1,1}, {2,2} };`

- Accediendo a los elementos
 - Comienzo en 0 y hasta el tamaño del array-1
 - **C no hace ninguna comprobación sobre el acceso a posiciones fuera de los límites del array**
 - Ej: `printf("%d", p[2]);`
`printf("%d", c[2][1]);`

Tipos de datos estructurados

Estructuras

- Estructuras (registros)

Nombre (tag) de la estructura: Optativo

```
struct Nombre{  
    Tipo1 miembro1;  
    Tipo2 miembro2;  
    ...  
} var1, var2, var3;
```

Miembros de la estructura
(pueden ser a su vez estructuras)

Definición de variables del tipo de la estructura
Si no hay variables, declaración posterior:
struct Nombre var;

Las estructuras pueden ser copiadas, asignadas, pasadas a funciones y retornadas por funciones

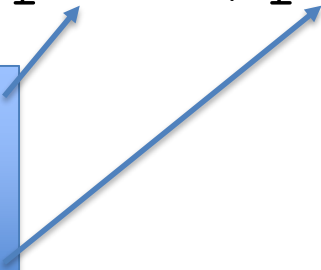
Tipos de datos estructurados

Estructuras

- Ej: Punto

```
struct Punto {  
    int x, y;  
};  
  
struct Punto p1;  
printf("%d %d", p1.x, p1.y);
```

Utilizamos el “.” para acceder a los miembros de la estructura



Tipos de datos estructurados

Estructuras

- Ej: Rectángulo

```
struct Rectangulo {  
    struct Punto p1, p2;  
};
```

typedef
para definir
nuevos
tipos

Alternativamente:

```
typedef struct Punto Punto;  
struct Rectangulo {  
    Punto p1, p2;  
};
```


Tipos de datos estructurados

Estructuras

- Inicializando y asignando estructuras

```
struct Punto p={2,3};
```

```
struct Punto p2=p;
```

```
printf("%d %d", p2.x, p2.y);
```

- Arrays de estructuras

```
struct Punto p[10];
```

```
p[0].x=12; p[0].y=14;
```

Tipos de datos estructurados

Uniones

- Una unión (union) es un tipo estructurado de C que permite almacenar datos de diferentes tipos en la misma localización de memoria
 - Una unión tiene varios miembros como una estructura
 - Pero sólo un miembro tiene un valor válido en un momento dado
- Ej: union Data {
 int i;
 float f;
 char str[20];
} data;

Tipos de datos estructurados

Uniones

- Accediendo a los miembros de la unión
 - Usamos el operador ‘.’ como en las estructuras
 - Sólo el último valor asignado será válido
- Ej:

```
union Data data;
```

```
data.i = 10;
```

```
data.f = 220.5;
```

```
printf( "data.i : %d\n", data.i );
```

```
printf( "data.f : %f\n", data.f );
```

Tipos de datos estructurados

Enumerados

- Un enumerado es una lista de valores constantes (enteros)
 - Ej: `enum months { JAN, FEB, MAR, APR, MAY, JUN, JUL, AUG, SEP, OCT, NOV, DEC };`
 - Declaración de variables
 - `enum months m;`
 - Asignación de valores
 - `m = FEB;`
 - Comparación
 - `if (m==FEB) { ... }`
 - El primer valor (si no se indican explícitamente) es 0
- Nota: Se pueden declarar constantes individuales con la palabra clave **const**
 - Ej: `const double e = 2.71828182845905;`

Tipos de datos estructurados

Cadenas de caracteres

- En C, una cadena de caracteres (string) es un array de caracteres terminado por el carácter ASCII 0 (`'\0'`)
- Definición de una cadena
 - `char cad[TAMCAD];`
 - C no controla el acceso a posiciones fuera de la cadena
- Manejo de la cadena utilizando el nombre del array
 - Ej: `printf("%s", cad);`
- Acceso a posiciones individuales, con sintaxis de array
 - Ej: `printf("%c", cad[0]);`

Tipos de datos estructurados

Cadenas de caracteres

- Inicialización
 - En la propia declaración:
 - `char cad[TAMCAD]="hola";`
- No está definido el operador de asignación
 - `cad="hola"; /* ERROR */`
- Utilizar `strcpy` para asignar un valor a una cadena
 - `strcpy(cad, "hola");`

Tipos de datos estructurados

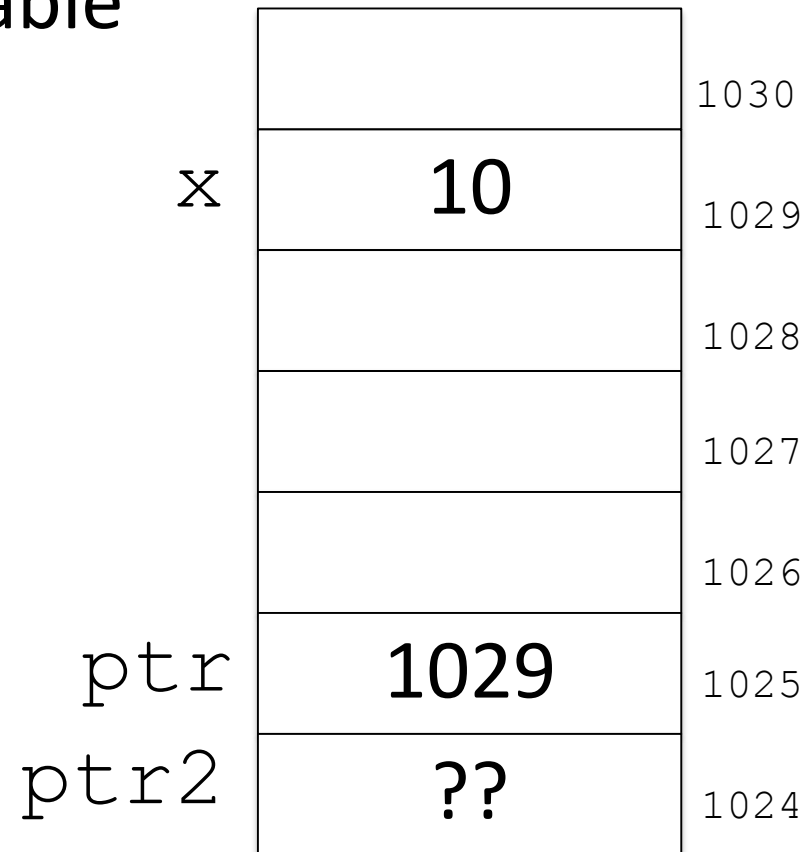
Cadenas de caracteres

- Algunas funciones útiles <string.h>
 - strcpy(s1, s2);
 - Copia s2 en s1 (sin controlar el tamaño de s1)
 - strlen(s1);
 - Retorna la longitud de s1 (se supone terminado en \0)
 - strcat(s1, s2);
 - Añade s2 a s1
 - strcmp(s1, s2);
 - Compara s1 y s2, devuelve 0 si son iguales; menor que 0 si s1<s2 y mayor que 0 si s1>s2

Tipo Puntero

- Un puntero es una variable que contiene la dirección de otra variable

```
int x = 10;  
  
int *ptr ;  
ptr = &x ;  
  
double * ptr2 ;
```



Punteros en C

- Un puntero es una variable que contiene la dirección de otra variable

```
int x = 10;
int y = 20;
int z[4] ;

int *ptr ;    // ptr es un puntero a un entero

ptr = &x ;    // ptr apunta a x
y = *ptr ;    // *ptr devuelve el contenido del
puntero (y vale 10)
*ptr = 0 ;    // el valor de x es 0
ptr = &z[0] ; // ptr apunta a z[0]
```

Punteros en C

- Un puntero está restringido a apuntar a un tipo de terminado
 - Punteros a enteros, doubles, estructuras, etc.
- La excepción son los punteros `void`
 - Un puntero `void` puede apuntar a cualquier objeto

```
int x = 10;  
  
void *ptr ;  
ptr = &x ;
```

Operaciones con punteros

- Indirección (*deferencing*)
 - Operador unario *
 - Devuelve el contenido apuntado por el puntero

```
int x = 10;  
  
int *ptr ;  
ptr = &x ;  
  
int y = *ptr // deferencing
```

Punteros y arrays

- En C, los punteros y los arrays están estrechamente relacionados
- Toda operación sobre un array se puede hacer con punteros
 - Ventaja: mayor control
 - Inconveniente: código más complejo para los que no dominan el lenguaje

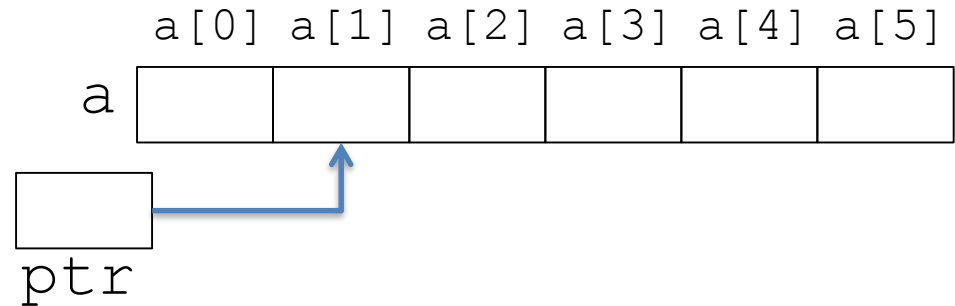
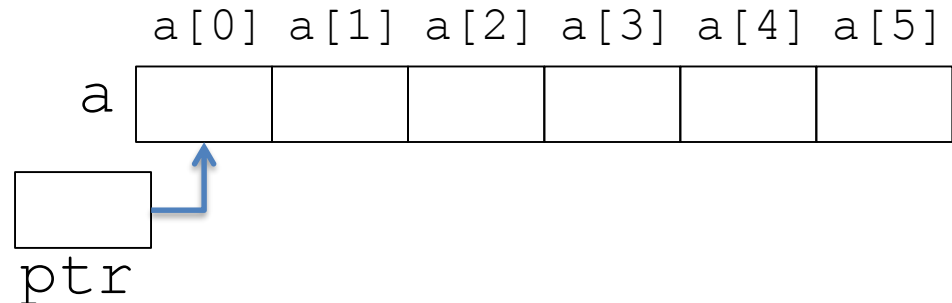
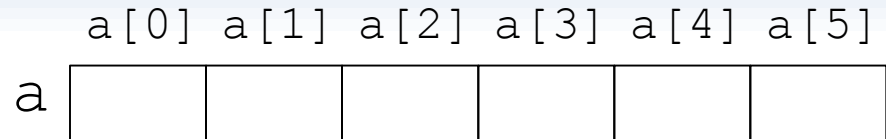
Punteros y arrays

```
int a[6] ;
```

```
int *ptr ;
```

```
ptr = &a[0] ;
```

```
ptr += 1 ;
```



Punteros y arrays

- Algunos casos concretos:
 - El identificador de un array es un puntero a la primera posición del array: `&a[0] == a`
 - `* (ptr + 1) = a[1]`
 - `ptr++`: mueve el puntero a la siguiente posición del array
 - `ptr += i`: mueve el puntero `i` posiciones en el array

Punteros a estructuras

- Se pueden definir punteros a estructuras como a cualquier otro tipo
 - Ej: `struct Punto *ppunto;`
 - O con `typedef`
`typedef struct Punto *PPunto;`
`PPunto ppunto;`
- El acceso a los miembros puede hacerse de dos formas
 - Sintaxis 1: `(*ppunto).x=20;`
 - Sintaxis 2: `ppunto->x=20;`
 - La segunda opción es más legible que la primera

Punteros a funciones

- En C, el nombre de una función es un puntero a dicha función
- Ejemplo:

```
int (*comp)(void *, void *)
```

- No confundir con:

```
int *comp(void *, void *)
```


Punteros a funciones

- En C, el nombre de una función es un puntero a dicha función
- Ejemplo:

```
int esMayorInt (int *a, int *b) { return *a > *b ; }

int esMayorDouble (double *a, double *b) { return *a > *b ; }

int esMayor(void *a, void *b, int (*f)(void *, void *)) {
    return f(a, b) ;
}
```

Índice de contenidos

- El lenguaje C. Introducción
- E/S
- Control de flujo de ejecución
- Tipos de datos:
 - Tipo de datos simples, estructurados y el tipo puntero
- Subprogramas: procedimientos y funciones
- Gestión de memoria dinámica
- Programación modular
- Persistencia de datos: Ficheros
- Operaciones de bajo nivel

Subprogramas

- C define funciones que pueden:
 - Devolver “nada” (void) -> Procedimientos
 - Devolver valores de algún tipo
- Argumentos de las funciones
 - Todos se pasan por valor pero....
 - Es posible pasarlos por referencia.
 - En la llamada se debe proporcionar la dirección de la variable
 - En la definición formal el parámetro debe ser declarado como puntero a una variable del tipo correspondiente.

Punteros y argumentos de funciones

- Los parámetros de las funciones en C se pasan por valor
- El paso por referencia se hace con punteros

```
swap (a, b)
...
void swap(int x, int y) {
    int temp;
    temp = x;
    x = y;
    y = temp;
}
```

Incorrecto

```
swap (&a, &b)
...
void swap(int* x, int* y) {
    int temp;
    temp = *x;
    *x = *y;
    *y = temp;
}
```

Correcto

Arrays y funciones

- Arrays como parámetros de funciones
 - Se pasan con sintaxis de punteros:
 - `void mifuncion(int *param);`
 - Ej: `void mostrar(int *elem, int n) {
 int i;
 for (i=0; i<n; i++)
 printf("%d", elem[i]);
}`
- Retornando arrays: no es posible, se pueden devolver punteros (cuidado con esto!)
 - Muchas funciones de cadenas de caracteres hacen esto
 - Ej:
`char *strcpy(char *destination, const char *source);`

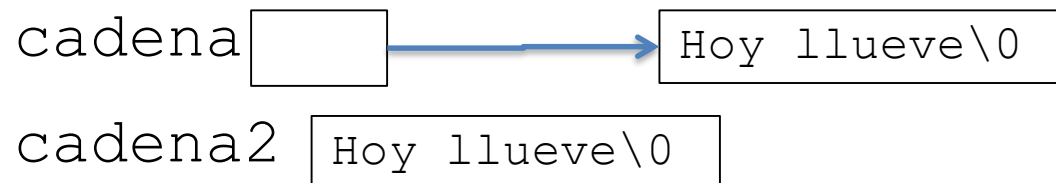
Arrays y funciones

- Si necesitamos pasar un array como parámetro de un función **SIEMPRE** se le pasa la dirección de comienzo de éste
 - Nunca se pasan por valor
- Ejemplo: Implementar una función que copie el contenido de una cadena en otra.

Arrays y funciones

- Representación de cadenas en memoria:

```
char *cadena = "hoy llueve" ; // puntero a constante  
char cadena2[] = "hoy llueve" ; // array  
  
char * ptr = cadena ;
```



Arrays y funciones

- Ejemplo: copiar cadenas

```
/* strcpy: copiar t a s; version con arrays */
void strcpy(char *s, char *t) {
    int i = 0;
    while ((s[i] = t[i]) != '\0')
        i++;
}
```

```
/* strcpy: copiar t a s; version con punteros*/
void strcpy(char *s, char *t) {
    while ((*s = *t) != '\0') {
        s++;
        t++;
    }
}
```


Arrays y funciones

- Ejemplo: copiar cadenas

```
/* strcpy: copiar t a s; version 2 con punteros*/  
void strcpy(char *s, char *t) {  
    while ((*s++ = *t++) != '\0') ;  
}}
```

```
/* strcpy: copiar t a s; version 3 con punteros*/  
void strcpy(char *s, char *t) {  
    while (*s++ = *t++) ;  
}
```

Estructuras y funciones

- Usando estructuras en llamadas a funciones

```
void mostrar(struct Punto p) {  
    printf("%d %d", p.x, p.y);  
}
```

Cuando se realiza la llamada anterior, la estructura tiene que ser **copiada**: muy ineficiente en el caso de estructuras de gran tamaño (mejor paso con punteros)

Estructuras y funciones

- Retornando estructuras

```
struct Punto crear(int x,int y) {  
    struct Punto res;  
    res.x=x; res.y=y;  
    return res;  
}
```

...

```
struct Punto origen;  
origen = crear(0,0);
```

Estructuras y funciones

- Ejemplo estructuras con sintaxis punteros

```
void mostrar(struct Punto *p) {  
    printf("%d %d", p->x, p->y);  
}
```

Variables static

- Son variables cuyo valor permanece entre llamadas a funciones

```
/* Static.c */

#include <stdio.h>

void add2() {
    static int var = 1 ;
    printf("%d\n", var+=2) ;
}

int main (int argc, char ** argv) {
    add2() ;
    add2() ;
    add2() ;
}
```

Índice de contenidos

- El lenguaje C. Introducción
- E/S
- Control de flujo de ejecución
- Tipos de datos:
 - Tipo de datos simples, estructurados y el tipo puntero
- Subprogramas: procedimientos y funciones
- **Gestión de memoria dinámica**
- Programación modular
- Persistencia de datos: Ficheros
- Operaciones de bajo nivel

Gestión de memoria dinámica

- Una de las características más importantes de C es la gestión dinámica de memoria
- Resumen de funciones:

Tipo	Descripción
<code>void *malloc(size_t n)</code>	Asigna un número de bytes de memoria
<code>void *calloc(size_t n, size_t size)</code>	Igual que malloc, pero inicializa a ceros
<code>free(void *ptr)</code>	Devuelve memoria previamente solicitada
<code>void *realloc(void *ptr, size_t size);</code>	Incrementa el tamaño del bloque especificado, reubicando si es necesario

Gestión de memoria dinámica

- Las funciones malloc, calloc y realloc
 - Devuelven un puntero a la zona de memoria asignada
 - En caso de error (memoria insuficiente) devuelven NULL
- Estas funciones devuelven un tipo `void *`
 - Luego hay que usar conversiones explícitas de tipo para asignar el puntero devuelto

Gestión de memoria dinámica

```
// a block of 20 ints is requested.
int * ptr1 = (int *)malloc(20*sizeof(int)) ;
if (ptr1 == NULL) {
    fprintf(stderr, "Error requesting memory for 20 int\n") ;
    exit(-1) ;
}

// a block of 20 doubles is requested.
double *ptr2 = (double *)calloc(20, sizeof(double)) ;

// the block of 20 ints is increased to 40 ints
int * ptr3 = (int *)realloc(ptr1, 40) ;

//free(ptr1) ;
free(ptr2) ;
free(ptr3) ;
```

Listas Enlazadas

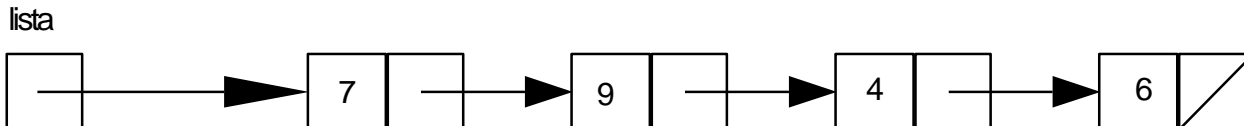
Tipos

```
typedef struct NodoNum *ListaNum;  
struct NodoNum {  
    int num;  
    ListaNum sig;  
};
```

Variables

```
ListaNum lista;
```

Ejemplo: lista con 7, 9, 4 y 6



Listas Enlazadas

Consideraciones:

- A cada elemento (estructura) se le denomina nodo.
- En general, un nodo de una lista enlazada puede contener **toda la información que deseemos** (todos los campos que queramos), **más un campo de tipo Puntero**, que apuntará al siguiente registro.
- El puntero “lista” apunta al primer nodo.
- Todos los nodos son variables dinámicas que han ido añadiéndose a la lista.
- El último nodo contiene un puntero NULO (**NULL**) que será utilizado por los algoritmos que manipulen la lista para detectar que se trata del último nodo de la misma.

Operaciones básicas

- Recorrido de una lista
- Recorrido condicional de una lista
- Insertar un nodo al principio
- Eliminar el primer nodo
- Insertar un nodo en una lista enlazada ordenada
- Eliminar un nodo en una lista enlazada
- Crear una lista

Operaciones básicas

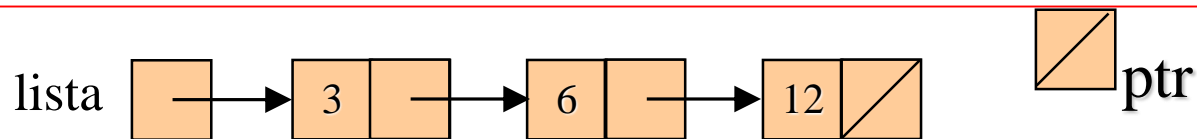
Recorrido de una Lista

```
ptr = lista;  
while (ptr != NULL) {  
    // Procesar *ptr  
    ptr = ptr->sig;  
}
```

Operaciones básicas

Mostrar una lista

```
void mostrar(ListaNum lista) {  
    ListaNum ptr;  
  
    ptr = lista;  
    while (ptr != NULL) {  
        escribir(ptr->num);  
        ptr = ptr->sig;  
    }  
}
```



Recorrido condicional

Recorrido condicional

```
ptr = lista;
```

```
while ((ptr != NULL) && (!cond)) {  
    ptr = ptr->sig;  
}
```

```
// o bien ptr apunta al primer nodo que cumple cond
```

```
// o bien ningún nodo cumple cond y ptr vale NULL
```

Operaciones básicas

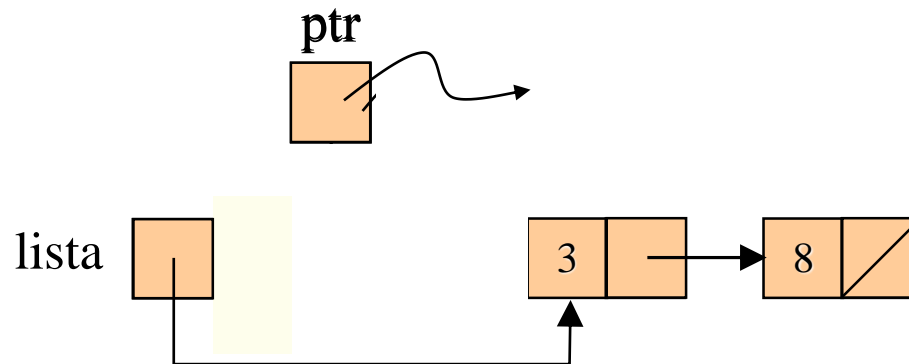
Buscar un nodo en una lista

```
ListaNum buscar_nodo(ListaNum lista,int elem) {  
    ListaNum ptr;  
  
    ptr = lista;  
    while ((ptr != NULL) && (elem != ptr->num)) {  
        ptr = ptr->sig;  
    }  
    return ptr;  
}
```


Operaciones básicas

Eliminar el Primer Nodo

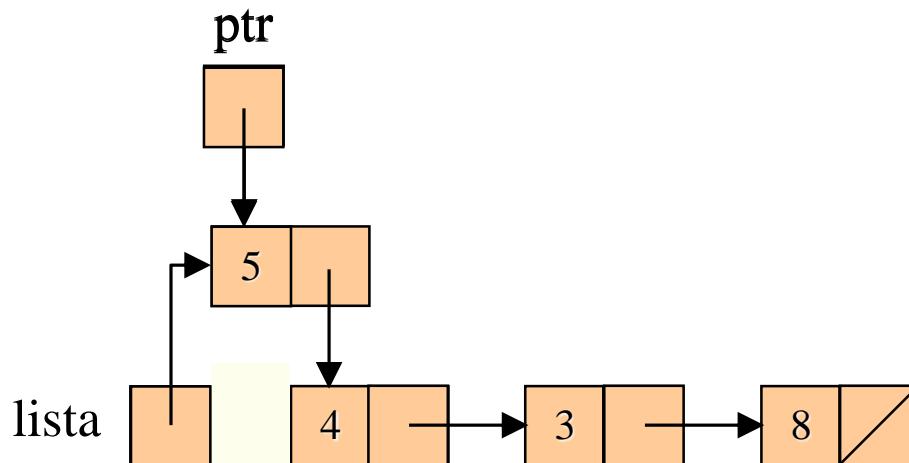
```
void eliminar_primerο(ListaNum *lista) {  
    ListaNum ptr;  
    // Precondición: hay al menos un nodo en la lista  
    ➡ ptr = *lista;  
    ➡ *lista = (*lista)->sig;  
    ➡ free(ptr);  
}
```



Operaciones básicas

Insertar un Nodo al Principio

```
void sumar_principio(ListaNum *lista,int elem) {  
  
    ListaNum ptr;  
  
    ➡ ptr = malloc (sizeof(struct NodoNum));  
    ➡ ptr->num = elem;  
    ➡ ptr->sig = *lista;  
    ➡ *lista = ptr;  
}
```



Operaciones básicas

Insertar un Nodo en una Lista Enlazada Ordenada

```
void ins_ordenado(ListaNum *lista, int elem) {  
  
    ListaNum nuevonodo;  
    ListaNum ant,ptr;  
  
    nuevonodo = malloc (sizeof(struct NodoNum));  
    nuevonodo->num = elem;  
    if (*lista == NULL) { // lista vacia  
        nuevonodo->sig = NULL;  
        *lista = nuevonodo;  
    } else  
        if (nuevonodo->num <= (*lista)->num) {  
            // insertar al principio  
            nuevonodo->sig = *lista;  
            *lista = nuevonodo;  
        }  
}
```

Operaciones básicas

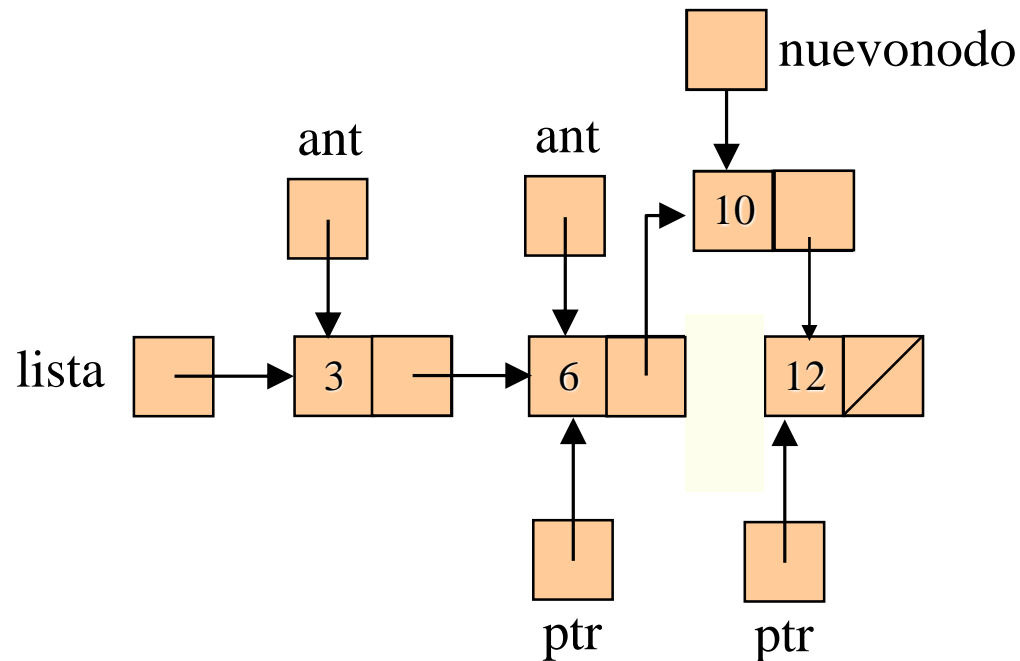
Insertar un Nodo en una Lista Enlazada Ordenada

```
else { // insertar en medio o al final
    ant = *lista;
    ptr = (*lista)->sig;
    while ((ptr != NULL) &&
           (nuevonodo->num > ptr->num)) {
        ant = ptr;
        ptr = ptr->sig;
    }
    nuevonodo->sig = ptr;
    ant->sig = nuevonodo;
}
}
```

Operaciones básicas

Insertar un Nodo en una Lista Enlazada Ordenada

```
➡nuevonodo=malloc(sizeof(struct NodoNum));  
➡nuevonodo->dato=10;  
➡nuevonodo->sig = NULL;  
➡BuscarPosicion  
➡nuevonodo->sig = ptr;  
➡ant->sig = nuevonodo;
```



Operaciones básicas

Eliminar un Nodo en una Lista Enlazada

```
void eliminar(ListaNum *lista,int elem) {

    ListaNum ptr;  // Usamos dos vbles. pret siempre va un
    ListaNum ant;  // paso por detrás de ptr.

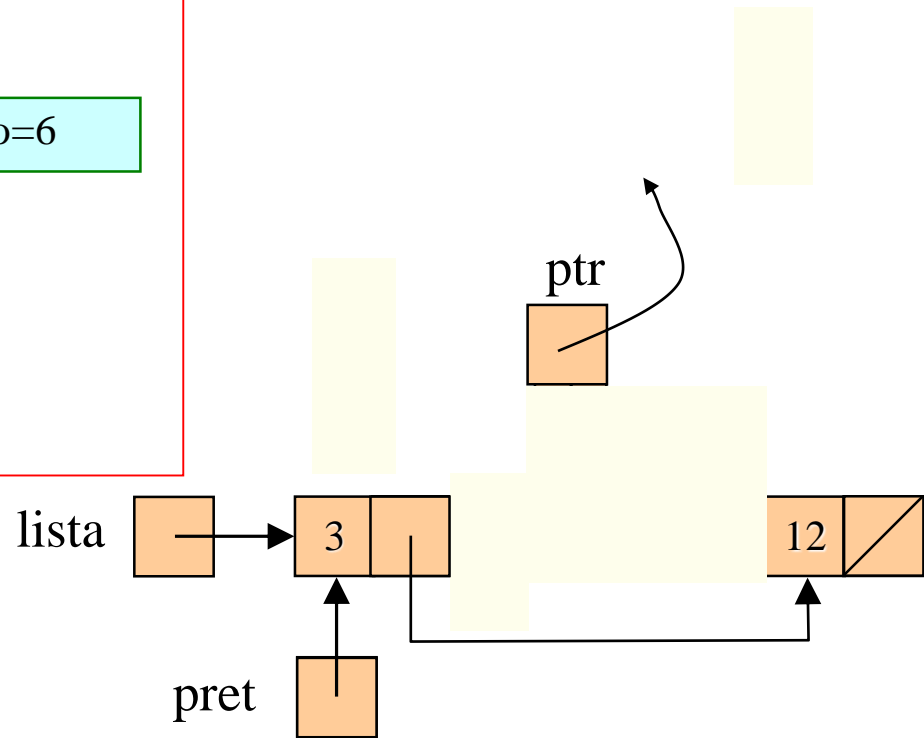
    if (*lista != NULL) // lista no vacia
        if ((*lista)->num == valor)
            eliminar_primerio(lista);
        else { // buscar elem en resto de la lista
            ant = *lista;
            ptr = (*lista)->sig;
            while ((ptr != NULL) && (ptr->num != elem)) {
                ant = ptr;
                ptr = ptr->sig;
            }
            if (ptr != NULL) { // encontrado
                ant->sig = ptr->sig;
                free(ptr);
            }
        }
    }
}
```

Operaciones básicas

Eliminar un Nodo en una Lista Enlazada

```
ant = *lista;  
ptr  = (*lista)->sig;  
while ((ptr != NULL) &&  
        (ptr->num != elem)) {  
    ant = ptr;  
    ptr = ptr->sig;  
}  
if (ptr != NULL) { // encontrado  
    ant->sig = ptr->sig;  
    free(ptr);  
}
```

dato=6



Operaciones básicas

Creación de una lista vacía

```
ListaNum crear_lista() {  
    return NULL;  
}
```


Operaciones básicas

Ejemplo de creación de una lista con elementos

```
ListaNum crear_lista() {  
  
    int dato;  
    ListaNum lista, ptr;  
  
    scanf("%d",&dato);  
    if (!dato {  
        lista = NULL;  
    else {  
        lista = malloc(sizeof(struct NodoNum)); // primer nodo  
        lista->num = dato;  
        ptr = lista;                               // Copiar primer nodo  
        scanf("%d",&dato);  
    }  
}
```

Operaciones básicas

Ejemplo de creación de una lista con elementos

```
while (dato) {  
    ptr->sig = malloc(sizeof(struct NodoNum));  
    ptr = ptr->sig;  
    ptr->num = dato;    // Copiar nodo  
    scanf("%d",&dato);  
}  
ptr->sig = NULL;  
}  
return lista;  
}
```

Índice de contenidos

- El lenguaje C. Introducción
- E/S
- Control de flujo de ejecución
- Tipos de datos:
 - Tipo de datos simples, estructurados y el tipo puntero
- Subprogramas: procedimientos y funciones
- Gestión de memoria dinámica
- Programación modular
- Persistencia de datos: Ficheros
- Operaciones de bajo nivel

Programación modular

- En C, como en muchos otros lenguajes, existe la posibilidad de usar bibliotecas (librerías)
- Típicamente:
 - En un fichero .h se incluyen:
 - Declaraciones de tipos, constantes, variables estáticas, declaraciones de funciones
 - En un fichero .c se incluye:
 - La implementación de todo lo indicado en el .h

Programación modular

- Ejemplo: módulo para implementar una pila mediante lista encadenada de punteros

```
stack.h

#ifndef __STACK_H__
#define __STACK_H__

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

struct stackNode {
    int value ;
    struct stackNode * ptr ;
} ;

struct stack {
    int size ;
    struct stackNode * head ;
} ;

struct stack * stackCreate() ;
int stackPush(struct stack * s, int v) ;
int stackPop(struct stack * s, int * v) ;
int stackSize(struct stack * s) ;
#endif
```

```
stack.c

#include <stack.h>

struct stack * stackCreate() {
    printf("statCreated invoked\n") ;

    struct stack * s ;
    s = (struct stack *) malloc(sizeof
                                   (struct stack)) ;

    s->size = 0 ;
    s->head = NULL ;

    return s ;
} ;

int stackPush(struct stack * s, int v) {
    // Implementation of stackPush here
    printf("statPush invoked\n") ;
    return 0 ;
}

...
```

Programación modular

- Ejemplo: módulo para implementar una pila mediante lista encadenada de punteros
- Programa que hace uso del módulo

```
#include <stack.h> stack.test.c

int main(int argc, char *argv[]) {
    struct stack * myStack ;

    myStack = stackCreate() ;

    stackPush(myStack, 4) ;
}
```

Programación modular

- Compilación directa:

```
gcc stackTest.c -o stackTest stack.c -I.
```

- Compilación separada

```
gcc stackTest.c -c -I.  
gcc stack.c -c -I.  
gcc stackTest.o stack.o -o stackTest
```

- Herramienta para automatizar la compilación separada:
 - make

Módulos en C. Herramienta make

- Ejemplo Makefile:

```
CC = gcc
CFLAGS = -I. -g
LDFLAGS = -lm

stackTest: stackTest.o stack.o
    $(CC) stackTest.o -o stackTest stack.o $(LDFLAGS)

stackTest.o: stackTest.c
    $(CC) stackTest.c -c $(CFLAGS)

stack.o: stack.c stack.h
    $(CC) stack.c -c $(CFLAGS)

clean:
    rm stackTest.o stack.o stackTest
```


Índice de contenidos

- El lenguaje C. Introducción
- E/S
- Control de flujo de ejecución
- Tipos de datos:
 - Tipo de datos simples, estructurados y el tipo puntero
- Subprogramas: procedimientos y funciones
- Gestión de memoria dinámica
- Programación modular
- **Persistencia de datos: Ficheros**
- Operaciones de bajo nivel

Persistencia de datos: Ficheros

- Un fichero en C representa una secuencia de bytes en formato texto o binario
 - Funciones y tipos de datos definidos en `<stdio.h>`
 - Tipo de datos: **FILE** (usado con punteros)
- Apertura de ficheros:

```
FILE *fopen(const char *filename, const char *mode);
```

```
Ej: FILE *fent;
```

```
if ((fent = fopen("fichero.txt", "rt")) == NULL) {  
    perror("Error abriendo fichero.txt");  
}
```

Muy importante: **controlar el posible error en la apertura**

Persistencia de datos: Ficheros

- Posibles modos de apertura
 - **"r"** Abre para lectura. Si el archivo no existe o no se encuentra, la llamada falla.
 - **"w"** Abre un archivo vacío para escritura. Si el archivo especificado existe, se destruye su contenido.
 - **"a"** Abre para escritura al final del archivo (anexar). Crea el archivo si no existe.
 - Si añadimos un **"+"**, Ej: **"r+"**, **"w+"**, permite abrir para lectura/escritura
 - En el caso de ficheros binarios, tenemos que utilizar también **"b"** (**"t"** para ficheros de texto)
 - Ej: **"rb"**, **"wb"**

Persistencia de datos: Ficheros

- Cierre de ficheros
 - `fclose(f)` donde `f` es de tipo `FILE *` y resultado de un `fopen` anterior
- Escritura en ficheros
 - Muchas operaciones disponibles
 - Ej: `fputc`, `fputs`, `fprintf`, `fwrite`
 - Ej: `fprintf(fd, "escribir %d\n", num);`

Persistencia de datos: Ficheros

- Lectura de ficheros

- Muchas operaciones disponibles

- Ej: fgetc, fgets, fscanf, fread

- Ej:

- ```
int a, b, leídos;
```

- ```
leídos = fscanf(fd, "%d %d", &a, &b);
```

Retorna el
número
de ítems
leídos

- ```
leídos = fscanf(fd, "%s", cad);
```

- Comprobación de final de fichero: `feof`

- Ej: 

```
while (!feof(fd)) { ... }
```

# Persistencia de datos: Ficheros

- Las operaciones `fscanf/fprintf` manipulan ficheros de texto.
- Es necesario conocer el formato del fichero para poder realizar una lectura correcta de la información.
- Si el fichero es binario la lectura/escritura debe realizarse por bloques de bytes.
  - Se utilizan `fread` y `fwrite`

# Persistencia de datos: Ficheros

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

```
size_t fread(void *ptr, size_t size, size_t n,
 FILE *stream);
```

`ptr` es un array donde se van a almacenar los datos que se van a leer. `size` indica el tamaño de cada bloque de lectura. `n` especifica el número de elementos que se van a leer. `stream` es un puntero a un fichero que ha tenido que haber sido abierto previamente. `size_t` es un tipo que está definido también en la cabecera `stdio.h`.

La función devuelve el número de elementos que realmente se han leído (que puede ser menor que `n`); si todo ha ido bien o aún no se ha llegado al final del fichero, ese número devuelto debería ser mayor que 0 e igual o menor (en caso de que no sepamos la longitud de lo que vamos a leer) que el tercer argumento, `n`

# Persistencia de datos: Ficheros

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

```
size_t fwrite(const void *ptr, size_t
 size, size_t n, FILE *stream)
```

ptr hace referencia a un array con los datos que se van a escribir al fichero abierto apuntado por stream. size indica el tamaño de cada bloque de escritura. n especifica el número de elementos que se van a escribir. stream es un puntero a un fichero que ha tenido que haber sido abierto previamente.

La función devuelve el número de elementos que se han escrito; luego si no ha habido ningún error, ese número devuelto debería ser igual que el tercer argumento, n. El valor devuelto puede ser menos que este n si ha habido algún error.



# Persistencia de datos: Ficheros

- Copia de fichero

```
#define SIZE 1024
int ok=1,leidos,escritos;
char buffer[SIZE];
size_t bytes;

FILE *fent,*fsal;

fent=fopen("entrada.dat","rb"); /* Control de errores omitido */
fsal=fopen("salida.dat","wb");

while (ok && (leidos = fread(buffer, 1, sizeof(buffer), fent))>0) {
 escritos = fwrite(buffer, 1, leidos, fsal);
 ok = escritos == leidos;
}

fclose(fent);
fclose(fs al);
```

# Índice de contenidos

- El lenguaje C. Introducción
- E/S
- Control de flujo de ejecución
- Tipos de datos:
  - Tipo de datos simples, estructurados y el tipo puntero
- Subprogramas: procedimientos y funciones
- Gestión de memoria dinámica
- Programación modular
- Persistencia de datos: Ficheros
- Operaciones de bajo nivel

# Operaciones de bajo nivel

- Una de las ventajas de C es que se pueden realizar operaciones de bajo nivel:
  - Más eficiencia
  - Registros y variables volátiles
  - Manipulación de bits
- Aspectos de bajo nivel a considerar
  - Little vs big endian
  - Alineamiento en memoria de las variables
- No existe un tipo básico bit en C
  - Usar enteros o bytes y usar operadores de manipulación de bits
  - Tipos válidos: `char`, `short`, `int`, `long`
  - Campos de bits en estructuras

# Manipulaciones de bits

- Operadores

| Operador | Descripción              |
|----------|--------------------------|
| &        | Bitwise AND              |
|          | Bitwise inclusive OR     |
| ^        | Bitwise exclusive OR     |
| <<       | Left shift               |
| >>       | Right shift              |
| ~        | One's complement (unary) |

a: 

|   |   |   |   |   |   |   |   |
|---|---|---|---|---|---|---|---|
| 1 | 0 | 1 | 0 | 1 | 0 | 1 | 0 |
|---|---|---|---|---|---|---|---|

b: 

|   |   |   |   |   |   |   |   |
|---|---|---|---|---|---|---|---|
| 0 | 0 | 0 | 0 | 1 | 1 | 1 | 1 |
|---|---|---|---|---|---|---|---|

a&b: 

|   |   |   |   |   |   |   |   |
|---|---|---|---|---|---|---|---|
| 0 | 0 | 0 | 0 | 1 | 0 | 1 | 0 |
|---|---|---|---|---|---|---|---|

a|b: 

|   |   |   |   |   |   |   |   |
|---|---|---|---|---|---|---|---|
| 1 | 0 | 1 | 0 | 1 | 1 | 1 | 1 |
|---|---|---|---|---|---|---|---|

a^b: 

|   |   |   |   |   |   |   |   |
|---|---|---|---|---|---|---|---|
| 1 | 0 | 1 | 0 | 0 | 1 | 0 | 1 |
|---|---|---|---|---|---|---|---|

a<<1: 

|   |   |   |   |   |   |   |   |
|---|---|---|---|---|---|---|---|
| 0 | 1 | 0 | 1 | 0 | 1 | 0 | 0 |
|---|---|---|---|---|---|---|---|

b>>2: 

|   |   |   |   |   |   |   |   |
|---|---|---|---|---|---|---|---|
| 0 | 0 | 0 | 0 | 0 | 0 | 1 | 1 |
|---|---|---|---|---|---|---|---|

~b: 

|   |   |   |   |   |   |   |   |
|---|---|---|---|---|---|---|---|
| 1 | 1 | 1 | 1 | 0 | 0 | 0 | 0 |
|---|---|---|---|---|---|---|---|

# Manipulaciones de bits

- Ejemplos

```
n = n & 0177
```

- Pone a cero todos los bits de `n` excepto los 7 de menor orden

```
n = n | SET_ON
```

- Pone a uno todos los bits que estén a uno en `SET_ON`

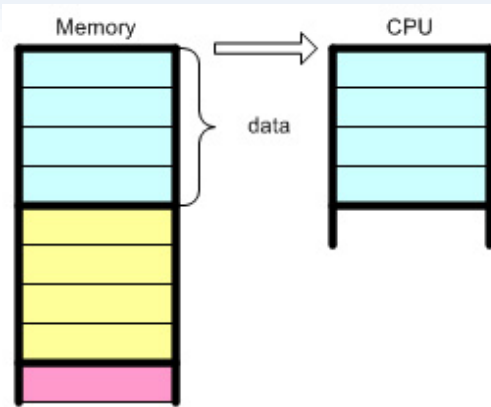
# Máscaras de bits

- Una máscara de bits (bitmask) es una palabra en la que cada bit tiene un significado concreto
- Para poner a uno o cero un determinado bit de la máscara hay que usar operaciones de bit
  - Típicamente, existen constantes con todos los bits a cero menos el que se quiere activar
  - Se realiza una operación OR entre la máscara y la constante

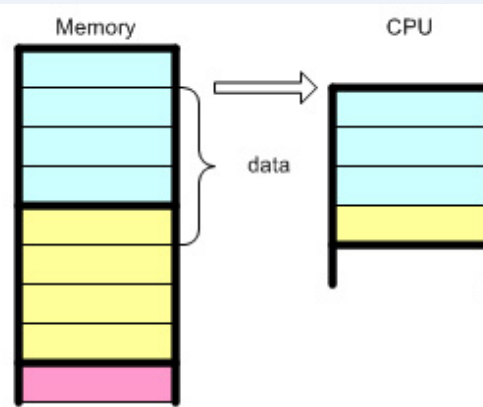
# Alineamiento de las variables

- La CPU no lee la memoria byte a byte
  - Lo hace en bloques de 2, 4, 8, etc.
  - Motivo: rendimiento
- El alineamiento de datos implica
  - Que las direcciones de los datos han de ser divisibles por 1, 2, 4, ...

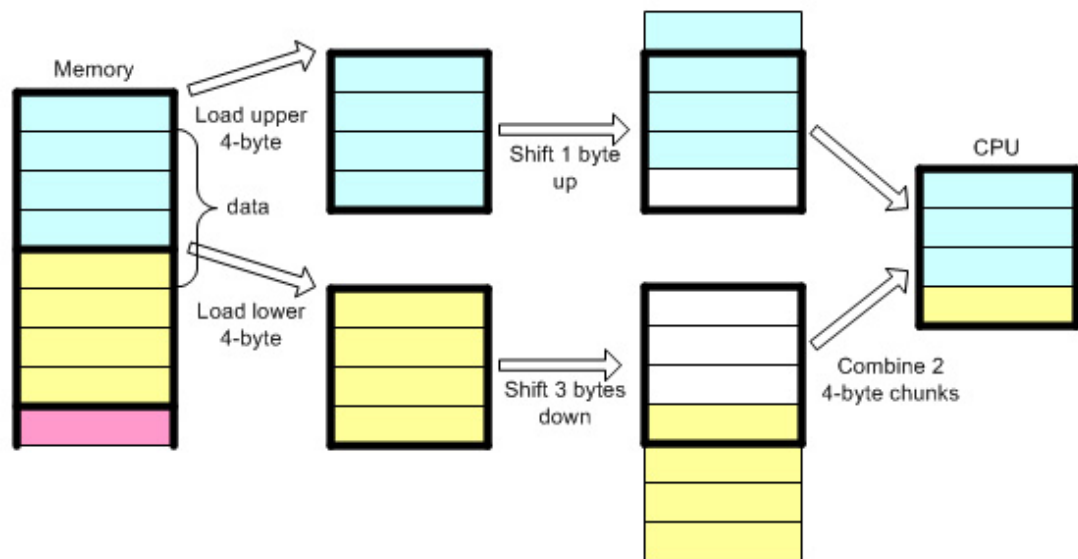
# Alineamiento de las variables



4-byte memory access for aligned data



4-byte memory access for misaligned data





# Alineamiento de las variables

- En muchas arquitecturas
  - Los compiladores alinean las variables según su tamaño
- El problema se presenta en estructuras, uniones y clases (en C++)
  - Para evitar problemas de rendimiento, es posible que el compilador use bytes de relleno

# Alineamiento de las variables

- Ejemplo:
  - Estructura con 1 char (1 byte) y 1 int ( 4 bytes)
  - El compilador puede insertar tres bytes entre el char y el int
    - Para un total de memoria de 8 bytes en lugar de 5
  - Así las direcciones son múltiplo de 4
  - Se pierde espacio de almacenamiento
  - Se gana en rendimiento

# Little vs big endian

- Los ordenadores hablan diferentes lenguas
  - Unos almacenan los datos de izquierda a derecha y otros de derecha a izquierda
- El problema surge cuando se quiere intercambiar datos entre máquinas que hablan lenguas distintas

# Little vs big endian

- En un byte, los bits se numeran de derecha a izquierda
  - El bit 0 es el que está más a la derecha y es el menor
  - El bit 7 es el que está más a la izquierda y es el mayor

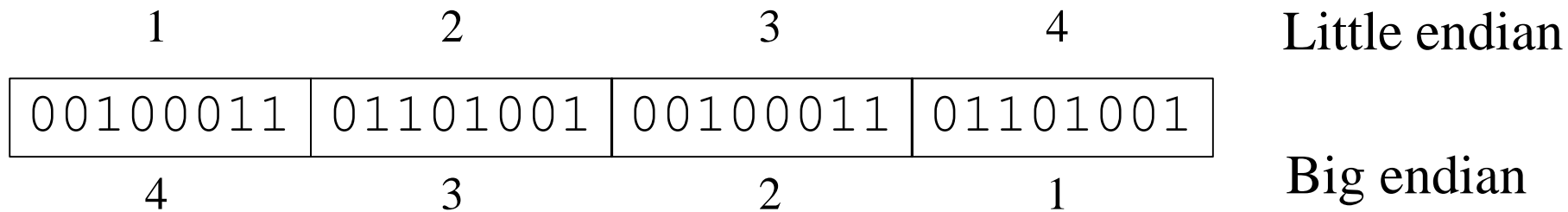
|          |
|----------|
| 00100011 |
|----------|

$$2^5 + 2^1 + 2^0 = 67$$

- El problema surge cuando se almacenan palabras de varios bytes
  - Se pueden ordenar de izquierda a derecha o a la inversa

# Little vs big endian

- Supongamos el caso de un entero en C
  - Se representa con 4 bytes
  - Big endian: el primer byte es el mayor
  - Little endian: el primer byte es el menor



# Little vs big endian. Ejemplo

Direcciones:

0

1

2

3

|          |          |          |          |
|----------|----------|----------|----------|
| 00010010 | 00110100 | 01010110 | 01111000 |
|----------|----------|----------|----------|

Caso 1: char (1 byte)

Caso 2: short (2 bytes)

```
char *c ;

c = 0 ;
*c = 0x12 ;
c = 1 ;
*c = 0x34 ;
c = 2 ;
*c = 0x56 ;
c = 3 ;
*c = 0x78 ;
```

```
short *s ; // dos bytes

s = 0 ;
printf ("%h", s) ;
```

Big endian:  $256 * \text{byte } 0 + \text{byte } 1 = 0x1234$

Little endian:  $256 * \text{byte } 1 + \text{byte } 0 = 0x3412$

# Referencias

- El lenguaje de programación C. Kerninghan y D. Ritchie
- [http://en.wikipedia.org/wiki/C \(programming language\)](http://en.wikipedia.org/wiki/C_(programming_language))