Programación II

Bloque temático 1. Lenguajes de programación

Tema 1. Introducción a la programación en lenguaje C

Tema 14. Introducción a la programación funcional: Lenguaje Haskell

Bloque temático 2. Metodología de programación

Bloque temático 3. Esquemas algorítmicos

Tema 1. Introducción a la programación en lenguaje C

1.1 Introducción al lenguaje C

El lenguaje C es uno de los lenguajes de programación más populares que existen hoy en día

Características más importantes:

- **lenguaje sencillo (aunque más difícil de aplicar)**
- **estructurado**
- **no orientado a objetos**
- **tipos definidos por el usuario**
- **no exige tipificación estricta**
- **permite compilación separada**
- **es de un nivel relativamente bajo**
- **compiladores baratos y eficientes**

Versiones del lenguaje C

- **Desarrollo inicial por Dennis M. Ritchie en los laboratorios Bell de AT&T (entre 1969 y 1973)**
  - **descendiente de un lenguaje anterior llamado "B"**
- **K&R C: en 1978 Brian Kernighan y Ritchie publicaron el libro "El lenguaje de programación C"**
  - **este libro fue durante años la especificación del lenguaje**
- **C ANSI o C ISO: estandarización en 1988, y luego en 1990; versión no ambigua y más portable, que añade**
  - **asignación de estructuras (registros)**
  - **tipos enumerados**
  - **prototipos de funciones**
  - **librerías estándar (funciones matemáticas, entrada/salida, etc.)**

- **C99: Posteriormente se refina la versión ISO/ANSI en 1999**
  - **las variables se pueden declarar en cualquier sitio**
  - **soporte para caracteres internacionales**
  - **soporte para números complejos**
  - **tipo entero `long`**
  - **comentarios `//`**
  - **etc.**

En este curso usaremos el ISO C 99

1.2 Estructura de un programa

Java	C

`import modulo1.*;`
`public class Clase {`
`  public static void main`
`  (String[] args)`
`  {`
`  <declaraciones>`
`  <instrucciones>`
`  }`
`}`
	`#include <modulo1.h>`
	`int main()`
	`{`
	`  <declaraciones>`
	`  <instrucciones>`
	`}`

Un programa C está compuesto al menos por una función

- **la función `main()`**
- **que, a su vez, puede llamar a otras funciones y así sucesivamente**

Ejemplo

```c
#include <stdio.h>

int main()
{
   printf("hola\n"); // printf escribe en pantalla
   return 0;
}
```

```java
public class Hola{

   public static void main(String[] args)
   {
      System.out.println("hola");
   }

}
```

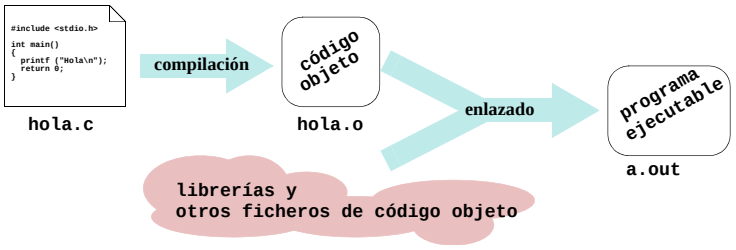

Ejemplo con varias funciones

```c
#include <stdio.h>
void func2() {
  printf("En función 2\n");
}
void func1() {
  printf("En función 1, antes de llamar a func2\n");
  func2();
  printf("En función 1, después de llamar a func2\n");
}

int main() {
  printf("En main, antes de llamar a func1\n");
  func1();
  printf("En main, después de llamar a func1\n");

  return 0;
}
```

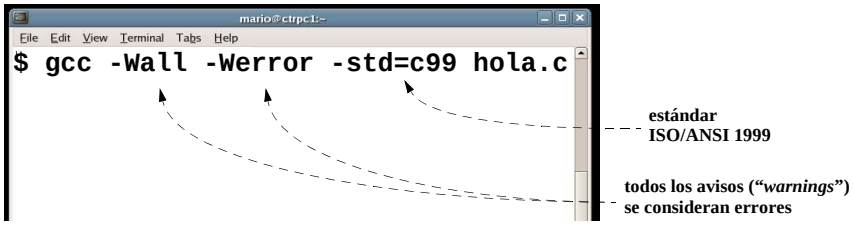

Algunas observaciones

C	Java
se distingue entre mayúsculas y minúsculas	idem
las instrucciones acaban con “;”, pero los bloques no	idem
comentarios entre `/* */` o empezando con `//`	además, existen los comentarios de documentación
todas las variables deben declararse	idem
los strings se encierran entre `""`	idem
`printf`: el retorno de línea se pone en el string `"\n"`	Igual que el printf de Java En Java también existen `println` y `print` (no disponibles en C)

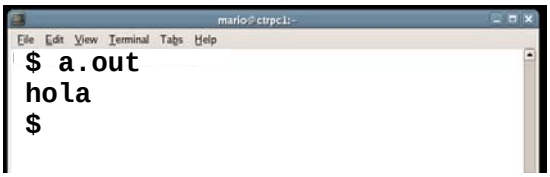
1.3 Compilación de un programa C



Para compilar y enlazar utilizaremos el compilador gcc:



Para ejecutar el programa:



Otras opciones de `gcc`:

- **Compilar varios ficheros de código**
  `$ gcc -Wall -Werror -std=c99 hola.c un.c otro.c`
  - **sólo uno puede contener una función `main()`**
- **Cambiar el nombre del ejecutable**
  `$ gcc -Wall -Werror -std=c99 hola.c -o hola.exe`

1.4 Tipos de datos y declaraciones de datos

Tipos predefinidos:

Java	C	Otros tipos C

`byte`
`short`
`int`
`long`
	`unsigned char`
	`short, short int`
	`int`
	`long, long int`
	`long long`
		`char`
		`unsigned short int`
		`unsigned int`
		`unsigned long int`
		`unsigned long long`

`boolean`	(se usa `int`)	
`char`	`char`	

`float`
`double`
	`float`
	`double`
	`long double`
		`float _Complex`
		`double _Complex`
		`long double _Complex`

`String`	`char[], char*`	

Declaraciones

C

```
int lower;
char c,resp;  // dos variables de tipo char
int i=0;      // variable inicializada
int j=0xA2;   // valor en hexadecimal
int k=072;    // valor en octal
const float eps=1.0e-5;  // constante
#define MAX 8            // otra constante
```

Java

```
int lower;
char c,resp;  // dos variables de tipo char
int i=0;      // variable inicializada
int j=0xA2;   // valor en hexadecimal
int k=072;    // valor en octal
final float eps=1.0e-5;  // constante
final int max=8;         // otra constante
```


Observaciones sobre las declaraciones

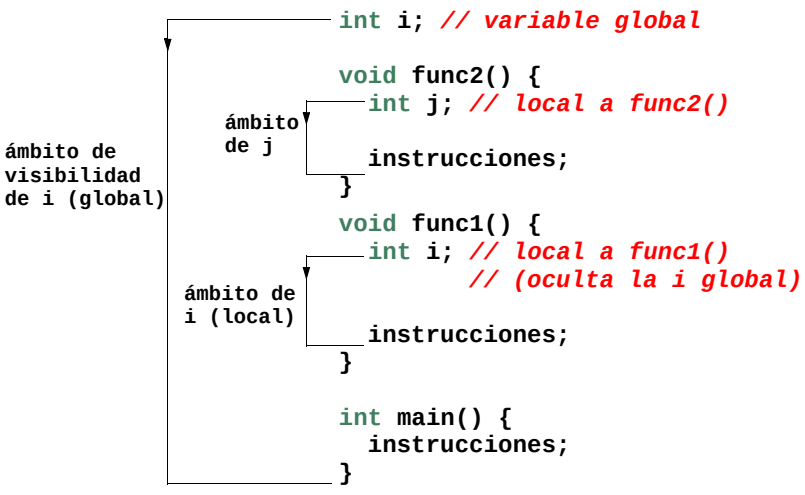
Puede observarse que, tanto en C como en Java:

- **las variables se pueden inicializar**
- **se pueden declarar como constantes**
- **se pueden declarar varias variables juntas**

Formato de los identificadores

- **todo en minúsculas y "_" para separar las palabras**
- **ejemplos: `cuenta_usuarios`, `máxima_velocidad_alcanzada`**

Ámbito de visibilidad



Tipos enumerados

En C se pueden definir tipos enumerados:

```
typedef enum {plano_x, plano_y, plano_z} dimension_t;
dimension_t fuerza, linea;
...
fuerza = plano_x;
linea = fuerza + 1; // toma el valor plano_y
```

La instrucción `typedef` permite crear tipos de datos nuevos, con los cuales se pueden crear variables más adelante

Observaciones:

- **En C y Java el operador de asignación es "="**
- **En C los valores enumerados se tratan como enteros, con valores 0, 1, 2, ...**
  - **`linea` toma el valor 1 (`plano_y`) en el ejemplo**

Caracteres

Los caracteres en C y Java se representan encerrados entre apóstrofes:

```
char c1,c2;
c1 = 'a';
```

Los caracteres de control tienen una representación especial:

Carácter	C y Java
salto de línea	'\n'
carácter nulo	'\0'
tabulador	'\t'
bell (pitido)	'\a'
apóstrofe	'\''

Las variables de tipo `char` pueden utilizarse en expresiones aritméticas (igual que ocurría en Java)

Arrays

Los arrays de C son muy diferentes de los de Java:

- **no tienen un tamaño conocido durante la ejecución**
  - **pero sí un tamaño fijo en memoria**
- **el usuario es responsable de no exceder su tamaño real**
- **se pueden manipular mediante punteros**

Los arrays de Java son mucho más seguros

Las siguientes declaraciones de arrays en Java:

```
final int MAX=8;
float[] v1; // creación de la referencia
v1=new float[MAX]; // creación: tamaño de 0 a MAX-1
short[][] c=new short[MAX][MAX];
```

Tienen el siguiente equivalente en C:

```
#define MAX 8 // constante
float v1[MAX]; //índice va de 0 a MAX-1
               //array creado al declararlo
short int c[MAX][MAX];
```

En C los arrays de números no se inicializan a cero

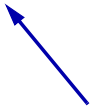
Tipos array

En C se pueden crear tipos array para usarlos más tarde en declaraciones de arrays

```
typedef float vector_t[MAX];
typedef short int contactos_t[MAX][MAX];
typedef int numeros_t[4];
```

declaraciones:

```
vector_t v = {0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0};
vector_t w;
contactos_t c1, c2;
numeros_t a;
```



Literal de array
(solo en inicialización)

Uso de elementos de arrays

Las siguientes instrucciones se escriben igual en Java y C:

```
w[3]=10.0;
c1[0][3]=1;
a[0]=9; a[1]=3; a[2]=0; a[3]=4;
```

Ejemplo de uso de arrays

```
#include <stdio.h>
#define NUM_ELE_VECTOR 10
typedef int vector_t[NUM_ELE_VECTOR];
int suma_elementos_vector(vector_t v) {
  int suma = 0;
  for(int i=0; i < NUM_ELE_VECTOR; i++) {
    suma += v[i];
  }
  return suma;
}

int main() {
  vector_t v = {0, 1, 2, 3, 4, 5, 6, 7, 8, 9};
  printf("La suma de los elementos del vector es:%d\n",
         suma_elementos_vector(v));
  return 0;
}
```

En C *NO* existe v.length

Strings en C

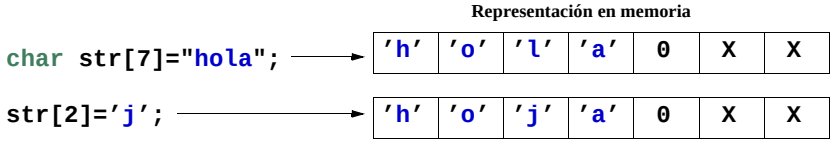
Los strings en C son simples arrays de caracteres

Los strings constantes se representan encerrados entre comillas, como en Java

```
char linea[80];  // string sin inicializar
char otra[] = "esto es un string"; // inicializado
```

Los strings deben incluir un carácter nulo (código ASCII cero)

- **sirve para indicar dónde acaba la parte "utilizada" del string**
- **los strings constantes ya le incluyen**



Operaciones con strings

Funciones para manipular strings (definidas en `<string.h>`)

`strcpy(s1, s2)`	Copia el string `s2` en `s1`
`strncpy(s1, s2, n)`	Copia hasta n caracteres de `s2` en `s1`
`strcat(s1,s2)`	Añade el string `s2` al final del `s1`
`size_t strlen(s)`	Retorna un entero con la longitud de `s`
`int strcmp(s1,s2))`	Compara el string `s1` con `s2` y retorna: entero menor que cero si `s1<s2` entero mayor que cero si `s1>s2` cero si `s1=s2`

Las tres primeras son operaciones peligrosas, pues no se comprueba si el resultado cabe en el espacio reservado a `s1`

Operaciones de conversión con strings

En `<stdlib.h>` se definen funciones para convertir strings a números

`int atoi(const char *s1)`	Convierte el string `s1` a entero
`double atof(const char *s1)`	Convierte el string `s1` a real

Estructuras

El equivalente a la clase Java sin operaciones (sólo con datos) es la estructura en C:

```
// definición
struct fecha {
       int dia;
       int mes;
       int anho;
};

// declaración de estructuras
struct fecha f1 = {11, 7, 2010}; //sólo al inicializar
struct fecha f2;

// acceso a los campos de la estructura
f2.mes = 5;
int dia = f1.dia;
```


También se podía haber escrito:

```
typedef struct {
       int dia;
       int mes;
       int anho;
} fecha_t;

fecha_t f1,f2;
```

Observar que la diferencia es que ahora el tipo se llama `fecha_t` en lugar de `struct fecha`

La asignación entre estructuras realiza la copia campo a campo

```
f2 = f1;
```

equivale a

```
f2.dia = f1.dia;
f2.mes = f1.mes;
f2.anho = f1.anho;
```

1.5 Operadores y expresiones

Tipo de Operador	C	Función
Aritméticos	**+,-** ***, /,** **%**	**Suma, Resta** **Multiplicación, División** **Módulo**
Relacionales	**==, !=** **>, >=** **<, <=**	**Igual a, Distinto de** **Mayor, Mayor o Igual** **Menor, Menor o Igual** **(*producen un entero*)**
Lógicos	**&&,| |, !**	**And, Or, Not** **(*trabajan con enteros*)**

Tipo de Operador	C	Función
Incremento y decremento	**++x, --x** **x++, x--**	**x=x+(-)1 y valor=x+(-)1** **valor=x y x=x+(-)1**
Manejo de bits	**&, |, ^** **<<, >>**	**And, Or, Or exclusivo** **Desplazamiento Izq., Der.**
Conversión de tipo	**(tipo) expr.**	**Convierte la expresión al tipo indicado**

Tipo de Operador	C	Función
Asignación **Nota: no hay asignación para arrays ni strings en C**	**=** **+=** **-=** ***=** **/=** **%=** **<<=** **>>=** **&=** **|=** **~=**	**Asigna el valor y lo retorna** **izqdo=izqdo+drcho** **izqdo=izqdo-drcho** **izqdo=izqdo*drcho** **izqdo=izqdo/drcho** **izqdo=izqdo%drcho** **izqdo=izqdo<<drcho** **izqdo=izqdo>>drcho** **izqdo=izqdo&drcho** **izqdo=izqdo|drcho** **izqdo=izqdo~drcho**

Operaciones matemáticas

Son funciones definidas en `<math.h>`:

`sin(x), cos(x), tan(x)`	**trigonométricas, en radianes**
`asin(x), acos(x), atan(x)` `atan2(y,x)`	**trigonométricas inversas**
`exp(x), log(x)`	**exponencial y logaritmo neperiano**
`abs(x)  (<stdlib.h>)`	**valor absoluto para enteros**
`fabs(x)`	**valor absoluto para reales**
`pow(x,y)`	**x elevado a y**
`sqrt(x)`	**raíz cuadrada**

1.6 Entrada/salida simple

Fichero de cabeceras `<stdio.h>`

```
int printf(string_formato[, expr, ...]);
    // Escribe en pantalla
    // Muy parecido al System.out.printf de Java
    // Retorna el número de caracteres escritos

int scanf(string_formato,&var[,&var...]);
    // Lee de teclado
    // Retorna el número de datos leídos correctamente
    // (veremos lo que significa el '&' cuando hablemos de
    // los punteros)

int getchar();
    // lee y retorna un carácter
```


Strings de formato más habituales:

`%d, %i` ⇨ enteros
`%c` ⇨ char
`%s` ⇨ string (una sola palabra al leer)
`%[^\n]` ⇨ lee un string con varias palabras
`%f` ⇨ leer *float*; escribir *float* y *double*, coma fija
`%e` ⇨ leer *float*; escribir *float* y *double*, exponencial
`%lf` ⇨ leer *double* en coma fija
`%le` ⇨ leer *double* en notación exponencial

Puede añadirse después del "%" la especificación de anchura y precisión (ancho.precisión); p.e.:

`%12.4f` ⇨ número con 12 caracteres (de ellos 4 son decimales)
`%4s` ⇨ string con 4 caracteres mínimo
`%5d` ⇨ número entero que ocupa mínimo 5 caracteres

Ejemplo de uso de `printf()`

```
#include <stdio.h>

int main() {
  char *saludo = "hola";
  char nombre[] = "Pepe";
  int edad = 14;  double num_real = 98.123456789;

  printf("Un saludo: %s\n", saludo);
  printf("%s tiene %d años\n", nombre, edad);
  printf("Número con pocos decimales:%1.1f\n",num_real);
  printf("Número con muchos decimales:%1.7f\n",num_real);

  return 0;
}
```

```
$ a.out
Un saludo: hola
Pepe tiene 14 años
Número con pocos decimales:98.1
Número con muchos decimales:98.1234568
$
```


Ejemplo de uso de `scanf()`

```
#include <stdio.h>

int main() {
  char nombre[50]; // suficientemente grande
  int edad;

  printf("¿Como te llamas?...");
  scanf("%[^\n]", nombre);
  printf("¿Cuántos años tienes?...");
  scanf("%d", &edad);

  printf("Nombre:%s, edad:%5d\n", nombre, edad);

  return 0;
}
```

```
$ a.out
¿Cómo te llamas?...Pepe García
¿Cuántos años tienes?...18
Nombre:Pepe García, edad:   18
$
```


Ejemplo de uso de `getchar()`

```
#include <stdio.h>

int main() {
  char c;

  while (1) { // bucle infinito
    printf("Introduce una letra...");
    c = getchar();
    while (getchar() != '\n'); // consume chars "sobrantes"

    if (c>='a' && c<='z') {
      printf("\'%c\' es una letra minúscula\n", c);
    } else {
      printf("\'%c\' NO es una letra minúscula\n", c);
    }
  }

  return 0;
}
```

```
$ a.out
Introduce una letra...e
'e' es una letra minúscula
Introduce una letra...A
'A' NO es una letra minúscula
```

1.7 Instrucciones de control

Instrucción condicional

Java	C

`if (exp_booleana) {`
`  instrucciones;`
`}`
	`if (exp_entera) {`
	`  instrucciones;`
	`}`

`if (exp_booleana) {`
`  instrucciones;`
`} else {`
`  instrucciones;`
`}`
	`if (exp_entera) {`
	`  instrucciones;`
	`} else {`
	`  instrucciones;`
	`}`

- Un resultado `0` en la `exp_entera` equivale a `False`
- Un resultado distinto de `0` equivale a `True`

Un fallo frecuente

Estas instrucciones ponen "`valor de i=4`" en pantalla

```
int i=2;

if (i=4) {
   printf("valor de i=%d\n",i);
} else {
   printf("no es 4\n");
}
```

En Java, el compilador hubiera detectado el fallo

- en C, como mucho, es un aviso (*warning*) ⇦ (si se utiliza la opción de compilación `-Wall`)

Entrada/salida con errores

```
#include <stdio.h>
int main ()  {
     int nota1, nota2, nota3, media;
     printf("Nota primer trimestre: ");
     if (scanf ("%d",&nota1)==0 || nota1<0 || nota1>10) {
           printf("Error"); return -1;
     }
     printf("Nota segundo trimestre: ");
     if (scanf ("%d",&nota2)==0 || nota2<0 || nota2>10) {
           printf("Error"); return -1;
     }
     printf("Nota tercer trimestre: ");
     if (scanf ("%d",&nota3)==0 || nota3<0 || nota3>10) {
           printf("Error"); return -1;
     }
     media=(nota1+nota2+nota3)/3;
     printf ("La nota media es : %d\n",media);
     return 0;
}
```

Instrucción condicional múltiple

Java	C

`switch (exp_discreta) {`
`  case valor1 :`
`  instrucciones;`
`  break;`
`  case valor2 :`
`  case valor3 :`
`  instrucciones;`
`  break;`
`  default :`
`  instrucciones;`
`}`
	`switch (exp_entera) {`
	`  case valor1 :`
	`  instrucciones;`
	`  break;`
	`  case valor2 :`
	`  case valor3 :`
	`  instrucciones;`
	`  break;`
	`  default :`
	`  instrucciones;`
	`}`

Cuidado: No olvidarse el "`break`"

Lazos

Java	C

`while (exp_booleana) {`
`  instrucciones;`
`}`
	`while (exp_entera) {`
	`  instrucciones;`
	`}`

`while (true) {`
`  instrucciones;`
`}`
	`while (1) {`
	`  instrucciones;`
	`}`

`do {`
`  instrucciones;`
`} while (exp_booleana);`
	`do {`
	`  instrucciones;`
	`} while (exp_entera);`

Lazo for

Es igual en ambos lenguajes:

```
for (int i=v_inic; i<=v_fin; i++) {
  instrucciones;
}
```

Observar que:

- **la declaración de la variable de control dentro del propio `for` es una característica introducida en el C99**
- **tanto en Java como en C pueden usarse:**
  - **`break` para salirse de un lazo (`for`, `while`, `do-while`)**
  - **`continue` para saltar hasta el final del lazo, pero permaneciendo en él**

En C *no* existe un lazo equivalente al `for-each` del Java

1.8 Punteros

En Java todas las estructuras de datos son dinámicas, y los objetos se manipulan mediante referencias

En C deben usarse referencias explícitas, denominadas punteros

- **se usa el carácter `'*'` para indicar un tipo puntero**

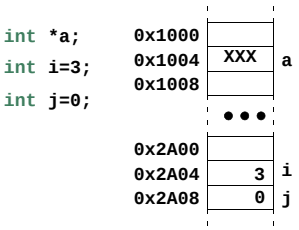
```
int *a; // a es un puntero a un entero (o, dicho de otra
        // manera, el "contenido" de a es un entero)
int i;  // i es un entero
```

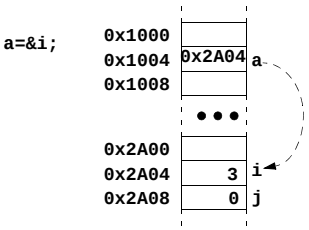
- **se usa el `'*'` para referirse al dato al que apunta el puntero**

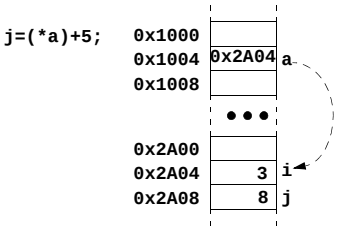
```
i=(*a)+8; // i pasa a tener: (valor al que apunta a) + 8
```

- **se usa el carácter `'&'` para obtener un puntero que apunta a una variable**

```
a=&i; // a pasa a tener un puntero la variable i
```





1.9 Funciones y paso de parámetros

Las funciones constituyen el principal elemento de estructuración del código en C

Las funciones se definen de forma similar a los métodos Java

```
tipo nombre_función (tipo1 param1, tipo2 param2, ...) {
     declaraciones locales;
     instrucciones;
     return ...;
}
```

Ejemplos de funciones C:

```
// retorna el resultado de sumar a + b
int suma(int a, int b) { ... }

// dibuja una línea en pantalla (no retorna nada)
void dibuja_línea(float x0, float y0, float x1, float y1)
{ ... }
```

Paso de parámetros

En C, los parámetros son sólo de entrada y por valor (se pasa una copia del parámetro a la función)

- **cuando se desea que el parámetro sea de salida o de entrada/salida es necesario pasar un puntero explícito**

Por ejemplo la siguiente función es incorrecta (pero compila):

```
void intercambia (int x, int y) { // incorrecta
  int temp;
  temp=x;
  x = y;
  y = temp;
}
int main () {
  int a=4,b=1;
  intercambia (a,b);
  // a sigue valiendo 4 y b sigue valiendo 1
  ...
```


La función correcta sería:

```
void intercambia(int *x, int *y) { // correcta
  int temp;
  temp=*x;
  *x = *y;
  *y = temp;
}

int main() {
  int a=4,b=1;
  intercambia(&a,&b);
  // ahora a vale 1 y b vale 4
  ...
```

En Java hubiera sido necesario usar objetos, que se pasan por referencia

También se usan punteros para parámetros de entrada "grandes"

- **para evitar una copia que implicaría pérdida de eficiencia**
- **en ese caso se usa la palabra reservada `const` para indicar que el parámetro no debe ser cambiado por la función**

```
struct alumno {
  char nombre[30];
  int nota1, nota2, nota3;
};
float nota_media(const struct alumno *alu) {
  return (alu->nota1 + alu->nota2 + alu->nota3)/3.0;
}
int main() {
  struct alumno alu1; float nota;
  ...
  nota=nota_media(&alu1);
  ...
}
```

Accede al campo de una estructura a través de un puntero a esa estructura.

`alu->nota3`

equivale a `(*alu).nota3`

Parámetros de tipo array

Los arrays siempre se pasan por referencia

- **estas dos funciones son equivalentes:**

```
float suma_elementos(float a[]) { ... }
float suma_elementos(float *a)  { ... }
```

Si el parámetro es de entrada se debe utilizar `const`

```
int cuenta_mayusculas(const char str[]) { ... }

int main() {
  char frase[] = "Esto eS una Frase";
  int num_mayusculas = cuenta_mayusculas(frase);
  ...
}
```

- **Observación: ponemos `frase` y no `&frase` puesto que:**
  - **el nombre de una variable de tipo array es también un puntero al primer elemento del array**

1.10 Creación dinámica de variables

Es posible crear variables dinámicamente en tiempo de ejecución

- **parecido a lo que se hace con el `new` de Java**

Se utilizan las funciones `malloc()` y `calloc()` (`<stdlib.h>`)

```
void *malloc(size_t size);
// reserva un área de memoria de 'size' bytes
// el área reservada no se pone a 0

void *calloc(size_t n, size_t size);
// reserva el espacio en memoria para un array de 'n'
// elementos, cada uno de 'size' bytes de tamaño
// el área reservada se inicializa a 0
```

Cuando se deja de usar un área de memoria, hay que liberarla expresamente (en C no hay recolector de basura)

```
void free(void *ptr);
```


Ejemplo de uso de `malloc()` y `calloc()`

```
struct punto {
  float x;
  float y;
};
struct punto *punto_ptr, *punto_array;

// crea un punto
punto_ptr =
     (struct punto *)malloc(sizeof(struct punto));
punto_ptr->x=3.0;
punto_ptr->y=-4.2;

// crea un array de 5 puntos
punto_array=
    (struct punto *)calloc(5,sizeof(struct punto));
punto_array[1].x=2.1;
punto_array[1].y=1.4;
...
```

`sizeof` sirve para conocer el número de bytes que ocupa una variable o un tipo

Creación del valor retornado por una función

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

// Retorna el nuevo string resultante de concatenar
// los dos strings pasados como parámetros
char *concatena(const char *str1, const char *str2);

int main() {
  char *str1="hola";
  char *str2=" y adiós";
  char *suma;
  suma=concatena(str1, str2);
  printf("\"%s\"+\"%s\"=\"%s\"\n",str1,str2,suma);
  return 0;
}
```



```
// Retorna el nuevo string resultante de concatenar
// los dos strings pasados como parámetros
char *concatena(const char *str1, const char *str2){
  // crea el espacio para el nuevo string
  char *str_resultado =
          malloc(strlen(str1) + strlen(str2) + 1);
  // copia el primer string
  strcpy(str_resultado, str1);
  // añade el segundo string al final
  strcat(str_resultado, str2);
  // retorna el nuevo string
  return str_resultado;
}
```

es un error común poner:

```
char str_resultado[strlen(str1) + strlen(str2) + 1];
```

La variable se crearía en el stack, con lo que la memoria que ocupa dejaría de estar reservada al finalizar la ejecución de la función

Ejemplo: lista enlazada

```
struct nudo {
  int valor;
  struct nudo *proximo;
};
struct nudo *primero;

primero=(struct nudo *)malloc(sizeof(struct nudo));
primero->valor = 3;
primero->proximo = NULL;
```

`NULL` es una constante que vale 0 (es equivalente al `null` de Java)

el símbolo "`->`" se usa para indicar el miembro de una estructura a la que apunta el puntero

1.11 Modularidad y ocultamiento de información

Un módulo en C es:

- **un conjunto de funciones, variables, constantes y definiciones de tipos**
- **que cumplen una determinada funcionalidad**

La modularidad en C se basa en la compilación separada de ficheros:

- **el cuerpo del "módulo" se pone en un fichero ("`módulo.c`")**
- **la especificación, compuesta por declaraciones y cabeceras de función, se pone en un fichero de "cabeceras" ("`módulo.h`")**
- **para usar el módulo desde otro fichero hay que poner:**

```
#include "módulo.h"
```

El método no es seguro, ya que el compilador no comprueba la correspondencia entre la especificación y el cuerpo

Ocultamiento de información

Por defecto todas las funciones y variables definidas en el cuerpo de un módulo ("`módulo.c`") son accesibles por otros módulos

- **pueden hacerse locales al módulo ("privadas") utilizando la palabra reservada `static`**
- **observar que el significado de `static` es totalmente distinto al que tiene en el lenguaje Java**

Ejemplo de fichero "`módulo.c`":

```
static int función_local_al_módulo(int a) {
  ...
}
void función_utilizable_por_otros_módulos(float f) {
  ...
}
...
```


Ejemplo de módulos en C

`stacks.h`

```
typedef struct {
  int elem[30];
  int top;
} stack_t;
void push(int i,
          stack_t *s);
int pop (stack_t *s);
...
```

`stacks.c`

```
#include "stacks.h"
void push(int i,
          stack_t *s) {
  ...
}
int pop (stack_t *s) {
  ...
}
...
```

`main.c`

```
#include "stacks.h"
#include <math.h>

int main() {
  stack_t s;
  ...
  push(1,&s);
  ...
}
```

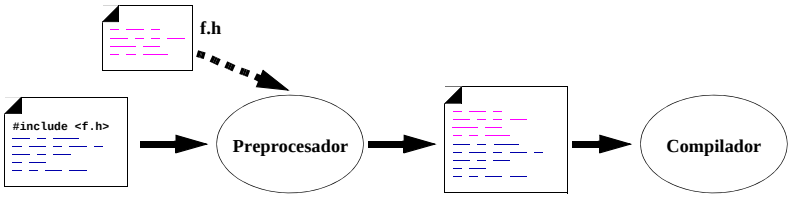
Para compilar todo junto:
`gcc -Wall -std=c99 main.c stacks.c`

1.12 Preprocesador

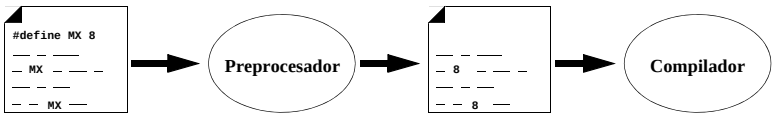
El código C, antes de ser compilado es preprocesado

Ya hemos visto algunas directivas al preprocesador:

- **`#include`: incluye un fichero**

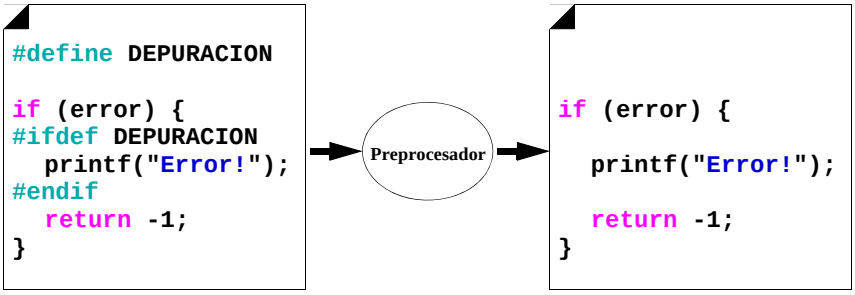


- **`#define`: sustitución**

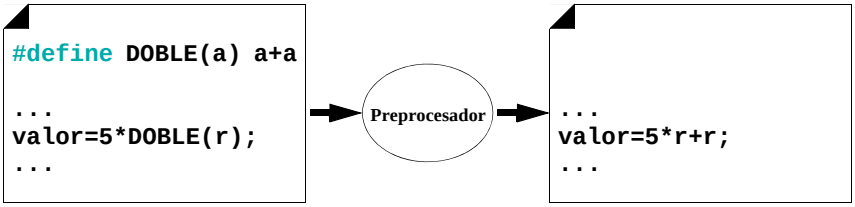


Otras directivas al preprocesador

- **`#ifdef`: inclusión condicional de código**



- **`#define`: macro**



  - **Para evitar errores hay que utilizar paréntesis:**
    `#define DOBLE(a) ((a)+(a))`

No se debe abusar de las *directivas al procesador*

- **puesto que pueden llegar a hacer el *código ilegible***

1.13 Librería estándar

El estándar C define una librería de funciones repartidas en un conjunto de ficheros de cabeceras:

- **`<stdio.h>`: `printf()`, `scanf()`, `getchar()`, ...**
- **`<stdlib.h>`: `malloc()`, `calloc()`, `free()`, `atoi()`, ...**
- **`<string.h>`: `strcpy()`, `strcat()`, `strcmp()`, `strlen()`, `strtok()`, ...**
- **`<math.h>`: `sin()`, `cos()`, `pow()`, `sqrt()`, `floor()`, ...**
- **`<assert.h>`, `<complex.h>`, `<ctype.h>`, `<errno.h>`, `<fenv.h>`, `<float.h>`, `<inttypes.h>`, `<iso646.h>`, `<limits.h>`, `<locale.h>`, `<setjmp.h>`, `<signal.h>`, `<stdarg.h>`, `<stdbool.h>`, `<stddef.h>`, `<stdint.h>`, `<tgmath.h>`, `<time.h>`, `<wchar.h>`, `<wctype.h>`**

Para obtener las funciones incluidas en un fichero de cabeceras:

```
$ man stdio.h
```


1.14 Bibliografía

[1] Kernighan, R. "El lenguaje de programación C". PrenticeHall-Pearson.

[2] Menéndez, R. "Programación básica en C". Escuela de Caminos.

[3] Harbison. "C, A reference manual". Prentice Hall. 5ª.