Programación en Lenguaje C

- Introducción al lenguaje C.
- Estructura de un programa.
- Tipos de datos y declaraciones de datos.
- Operadores y expresiones.
- Entrada/ salida simple.
- Instrucciones de control.
- Funciones y paso de parámetros.
- Modularidad.
- Tratamiento de errores.

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
- 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

Existen diversas versiones del lenguaje C:

- C común: apareció en 1978 y estaba directamente ligado al sistema operativo UNIX
- 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.)

Versiones del lenguaje C (cont.)

- Posteriormente se refina la versión ISO/ANSI en 1999
  - soporte para caracteres internacionales
  - soporte para números complejos
  - tipo entero `long`
  - comentarios `//`
  - etc.

En este curso usaremos el ISO C 99

2. Estructura de un programa

Estructura de un bloque en Java y en C:

```
{
   <declaraciones>

   <instrucciones>
}
```


Estructura de un programa (cont.)

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>`
	`}`

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");
      // println escribe
   }
}
```


Algunas observaciones:

C	Java
se distingue entre mayúsculas y minúsculas	idem
las instrucciones acaban con “`;`”, pero los bloques no	idem
las funciones son bloques que se pueden anidar	las funciones son bloques que siempre son métodos de las clases
comentarios entre `/* */` o empezando con `//`	además, existen los comentarios de documentación
todas las variables deben declararse	idem
los strings se encierran entre `" "`	idem
el retorno de línea se pone en el string `"\n"`	retorno de línea con operaciones específicas (`println`)

3. Tipos de datos y declaraciones de datos

Tipos predefinidos:

Java	C	Otros tipos C
byte short int long	signed 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	(se usa 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
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
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

Tipos enumerados

En C se pueden definir tipos enumerados, cuyos valores son identificadores:

```
typedef enum {planox,planoy,planoz} dimension;
dimension fuerza, linea;
...
fuerza=planox;
linea=fuerza+1; // toma el valor planoy
```

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

Tipos enumerados en Java

En Java se puede hacer lo equivalente a un tipo enumerado mediante el uso de constantes

```
final int planox=0;
final int planoy=1;
final int planoz=2;
int fuerza, linea;
...
fuerza=planox;
linea=fuerza+1; // toma el valor planoy
```


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 (`planoy`) en el ejemplo

Caracteres

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

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

Los caracteres de control tiene una representación especial:

Carácter	C y Java
salto de línea	'\n'
carácter nulo	'\0'
bell (pitido)	'\a'
apóstrofe	'\''

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

Arrays (cont.)

Las siguientes declaraciones de arrays en Java:

```
final int MAX=8;
float[] v1; //índice va de 0 a MAX-1
v1=new float[MAX]; // tamaño puesto en la creación
short[][] c=new short[MAX][MAX];
```

Tienen el siguiente equivalente en C:

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

En C los arrays de números no se inicializan a cero

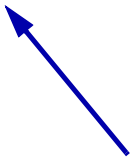
Tipos array

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

```
typedef float vector[MAX];
typedef short int contactos[MAX][MAX];
typedef int numeros[4];
```

declaraciones:

```
vector v={0.0,0.0,0.0,0.0,0.0,0.0,0.0,0.0};
vector w;
contactos c1,c2;
numeros 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;
```

No hay asignación de arrays en C

```
c1=c2; // no se puede
```


Strings en C

Los strings en C son arrays de caracteres, con un número de elementos variable.

El string debe terminar siempre con un carácter nulo (código ASCII cero), que sirve para saber dónde acaba el string.

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

```
char linea[80];
char otra[]="hola esto es un string";
char linea2[]="hola esto es una linea\n";
                // incluye un salto de linea
```

Los strings constantes ya incluyen el carácter nulo al final

Operaciones con strings

El módulo <string.h> define funciones para manipular strings

`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 el módulo <string.h>, para convertir errores a strings

`char * strerror(errnum)`	Convierte el código de error **errnum** a texto

En el módulo <time.h>, para convertir fechas a strings

`char * ctime(t)`	Retorna la fecha y hora representada por **t** (del tipo `time_t`) convertida a texto

El módulo <stdlib.h> define funciones para convertir strings a números

`int atoi(s1)`	Convierte el string **s1** a entero
`double atof(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 agno;
};
// declaración de estructuras
struct fecha f1={12,4,1996}; //sólo al inicializar
struct fecha f2;
...
f2.mes=5;
```


Estructuras (cont.)

También se podía haber escrito:

```
typedef struct {
     int dia;
     int mes;
     int agno;
} fecha;

fecha f1,f2;
```

Observar que la diferencia es que ahora el tipo se llama `fecha` y no `struct fecha`

Punteros y estructuras de datos dinámicas

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
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 una referencia la variable i
```


Punteros (cont.)

La declaración en C de una lista enlazada con creación dinámica de variables sería:

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

primero=(struct nudo *) malloc (sizeof (struct nudo));
primero->valor=3;
primero->proximo=0;
```

Punteros (cont.)

Observar:

- el símbolo "`->`" se usa para indicar el miembro de una estructura a la que apunta el puntero
- el puntero no se inicializa por defecto a 0 (que es el equivalente a `null`)
- `malloc`() equivale al `new` del Java, pero es preciso pasarle el número de bytes que se reservan para la nueva variable
- el número de bytes que ocupa un tipo de datos se obtiene con `sizeof`()

4. 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*)

Operadores y expresiones (cont.)

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

Operadores y expresiones (cont.)

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
`fabs(x)`	valor absoluto
`pow(x,y)`	x elevado a y
`sqrt(x)`	raíz cuadrada

5. Entrada/salida simple

Módulo <stdio.h>

```
int printf(string_formato[, expr, ...]);
    // escribe y retorna el número de caracteres
    // escritos
int scanf (string_formato,&var[,&var...]);
    // lee y retorna el número de datos leídos
    // correctamente
char *fgets(char *s, int size, FILE* stream);
    // lee una línea y la mete en s, hasta el
    // final de la línea (inclusive) o hasta un máximo
    // de size-1 caracteres; añade un nulo al final
    // stream es un fichero: para el teclado usar stdin
    // retorna NULL si hay error
```

Entrada/salida simple (cont.)

Strings de formato más habituales:

`%d, %i`	enteros
`%c`	char
`%s`	string (una sola palabra al leer)
`%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
```


Ejemplo de entrada/salida simple (C)

```
#include <stdio.h>

int main ()
{
      int nota1, nota2, nota3, media;
      // falta la detección de errores
      printf("Nota primer trimestre: ");
      scanf ("%d", &nota1);
      printf("Nota segundo trimestre: ");
      scanf ("%d",&nota2);
      printf("Nota tercer trimestre: ");
      scanf ("%d",&nota3);
      media=(nota1+nota2+nota3)/3;
      printf ("La nota media es : %d\n",media);
      return 0;
}
```

6. 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",i);
} else {
   printf("no es 4");
}
```

En Java, el compilador hubiera detectado el fallo

Ejemplo de entrada/salida simple (C, versión 2)

```
#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;
     }
     printf("Nota segundo trimestre: ");
     if (scanf ("%d",&nota2)==0 || nota2<0 || nota2>10) {
          printf("Error"); return;
     }
     printf("Nota tercer trimestre: ");
     if (scanf ("%d",&nota3)==0 || nota3<0 || nota3>10) {
          printf("Error"); return;
     }
     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);`

Lazos (cont.)

Java	C

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

Observar que:

- En C, la variable de control del lazo debe aparecer en las declaraciones; en Java se puede declarar en el propio lazo
- Puede usarse `break` para salirse de un lazo
- Puede usarse `continue` para saltar hasta el final del lazo, pero permaneciendo en él

7. Funciones y paso de parámetros

En C todos los métodos se llaman funciones, y no están asociados a las clases

Su estructura tiene la misma forma que en Java:

```
tipo_retornado nombre_funcion (parámetros)
{
        declaraciones locales;
        instrucciones;
}
```

La función retorna un valor al ejecutar la instrucción `return`

Si no retorna nada, se escribe la cabecera como:

```
void nombre_funcion (parámetros)
```


Parámetros

Los parámetros en C se separan con comas y se especifican así:

```
tipo nombre_parámetro
```

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
- también usa un puntero cuando el parámetro es grande

Parámetros (cont.)

Por ejemplo la siguiente función es incorrecta:

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

La llamada a esta función habría sido:

```
int a,b;
intercambia (a,b);
```

Pero no habría cambiado nada

Parámetros (cont.)

La función correcta sería:

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

Y su llamada es:

```
int a,b;
intercambia (&a,&b);
```

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

Parámetros (cont.)

En C, al pasar un parámetro de entrada mediante un puntero (para evitar la pérdida de eficiencia), puede indicarse que el parámetro no debe ser cambiado por la función:

```
struct t_alumno {
        char nombre[30];
        int nota1, nota2, nota3;
};
int nota_media (const struct t_alumno *alu)
{
     return (alu->nota1+alu->nota2+alu->nota3)/3;
}
main ()
{
        struct t_alumno alu1; int i;
        i=nota_media(&alu1); ...
}
```


Punteros a funciones

En C es posible crear punteros a funciones

Ejemplo:

```
void (*func) (int *,int *);   // declaración

func=&intercambia;            // asignación de valor
(*func)(&a,&b);               // llamada
```

Ejemplo de punteros a funciones

```
float integral (float (*f)(float x), float a, float b,
                int num_intervalos)
{
     float delta_x, x;
     float resultado = 0.0;
     int i;

     delta_x = (b-a)/(float)num_intervalos;
     x = a+delta_x/2.0;
     for (i=1;i<=num_intervalos;i++)
     {
          resultado+=f(x)*delta_x;
          x+=delta_x;
     }
     return resultado;
}
```


Ejemplo de punteros a funciones (C, cont.)

Ejemplo de cálculo de la integral de x^2 entre 0.5 y 1.5:

```
#include <stdio.h>

float cuadrado (float x)
{
    return x*x;
}


int main ()
{
     printf("La Integral de x**2 entre 0.5 y 1.15: %f\n",
            integral(&cuadrado,0.5,1.5,1000));
     return 0;
}
```

8. Modularidad y ocultamiento de información

Las funciones C pueden compilarse separadamente

En C pueden ponerse una o varias funciones (y declaraciones de datos) en un mismo fichero. Se le suele dar la terminación "`.c`"

En Java se agrupan varias funciones o métodos en módulos denominados clases

Las clases tienen datos y métodos que pueden ser:

- públicas
- privadas
- de paquete
- de la clase y sus derivados

Modularidad en C

Puede utilizarse un método para crear módulos de programa con una especificación separada del cuerpo:

- la especificación, compuesta por declaraciones y cabeceras de función se pone en un fichero de "cabeceras" (acabado en "`.h`")
- el cuerpo se pone en otro fichero acabado en "`.c`"
- para usarlo se pone `#include "nombre_fichero.h"`

Todas las funciones son públicas

Pueden ocultarse en parte no poniéndolas en el fichero de cabeceras

Ficheros de cabeceras

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

En el ejemplo que se muestra a continuación hay 2 módulos de programa:

- principal: `main.c`
- stacks: `stacks.h` y `stacks.c`

Ejemplo de módulos en C

stacks.h

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

main.c

```
#include "stacks.h"
#include <math.h>
int main() {
   stack s;

   push(1,&s);
   ...
}
```

stacks.c

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

9. Tratamiento de errores en C

```
#include <stdio.h>
int main ()
{
      int suma = 0;
      int i;
      char c;
      do {
              printf("Introduce valor (-999999 para acabar) : ");
              if (scanf("%d",&i) ==0) {
                  printf("Error\n"); scanf("%*s");
              } else {
                  if (i!=-999999) {
                        suma+=i;
                  }
              }
      } while (i!=-999999);
      printf ("La suma de la serie es : %d\n",suma);
      return 0;
}
```


Complejidad del lenguaje C

El lenguaje C es potente y sencillo, a costa de complicar la labor de programación.

La siguiente línea compila sin errores en C:

```
z+=!x||(w%=++h==x&&q^~q)*3;for(x=k(i<
p->y);x!=(g)w+(--q);z|=!(x&&w%3)/**/){k;}
```

¡Recordar que un programa se escribe una vez y se lee muchas veces!

El código equivalente en Java ocuparía varias líneas, pero:

- Sería mucho más fácil de entender.
- Tendría más posibilidades de ser correcto, ya que el compilador detecta muchos errores.

Conclusión

En C hay muchos errores que el compilador no detecta, lo que hace que los programas sean poco fiables

La programación de sistemas grandes es muy difícil, dada la ausencia de módulos en el programa

El lenguaje C++ corrige parte de estos problemas

- detecta más errores (tipificación más estricta), introduce módulos (clases) y es más expresivo (excepciones, plantillas)

Sin embargo, no soporta concurrencia, ni tiempo real, y los programas escritos en C++ son menos fiables que en Java.