Sistemas Operativos II

Universidad Nacional ARTURO JAURETCHE

Introducción a "C"

Docentes:
Ing. Jorge Osio
Ing. Eduardo Kunysz

Definición de Programa

Conjunto de **instrucciones** que se ejecutan de **forma secuencial**.

El programa en C se basa en **identificadores**, tanto **para datos** como para **componentes elementales** del programa

Estos **componentes** en otros lenguajes se denominan rutinas o procedimientos, pero en C se los llama **funciones**

Identificadores

- Nombre simbólico que se refiere a un **dato, variable o programa**
- El nombre debe estar relacionado con el dato al que representa.
- Los identificadores **no pueden** ser **palabras reservadas**
- Deben ser **declarados por el usuario**, indicando el **nombre y el tipo de dato** que van a representar

Concepto de función

- Los programas de gran tamaño se realizan mediante el concepto **modularización**
- **Cada módulo es** denominado comúnmente **Función**, en C se escribe Function
- La división de un programa en Funciones permite:

1. Modularización
2. Ahorro de memoria y tiempo de desarrollo
3. Independencia de datos

Concepto de función

Características de una Función

Una función **realiza una determinada tarea**

Está **asociada con un identificador**

Se debe **definir, declarar** y para utilizarla hacer una **llamada**

Se utilizan **argumentos** y se obtiene un **valor de retorno**

Concepto de función

Declaración de la Función

- En la declaración **no es necesario indicar los nombres** de los **argumentos** , pero si los tipos.
- Debe cerrarse con el **";"**
- Permite que el **compilador chequee la cantidad de argumentos y el tipo**, lo mismo del valor de retorno

Ejemplo: double power(double, double);

Concepto de función

Ejemplo de Definición de Función

```
double power(double base, double exponente)
{
    double resultado;
    ...
    resultado = ... ;
    return resultado;
}
```

Concepto de función

LA FUNCIÓN MAIN( )

- ***programa principal*** que es con el que **se comienza la ejecución del programa.** Este programa principal es también una función. Esta función se llama **main() y tiene la forma siguiente (la palabra *void es opcional en este caso):***

```
void main(void)
{
   sentencia_1
   sentencia_2
   ...
}
```

- Las ***llaves {...}*** *constituyen el modo utilizado por el lenguaje C para* ***agrupar varias*** **sentencias** de modo que se comporten como una sentencia única (*sentencia compuesta o bloque).*

Componentes Sintácticos

- *palabras clave*
- *identificadores*
- *Constantes*
- *cadenas de caracteres*
- *operadores y separadores*

Componentes Sintácticos

Palabras Clave

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

Componentes Sintácticos

Identificadores

1. Un ***identificador se forma con letras minúsculas,*** mayúsculas y ***dígitos.***
2. El carácter ***subrayado (_) se considera como una letra más.***
3. Un identificador **no** puede contener **espacios en blanco**,
4. El **primer carácter** de un identificador debe ser siempre **una letra** o un (_)
5. Se hace **distinción** entre letras **mayúsculas y minúsculas**.
6. **ANSI C** permite definir **identificadores** de hasta **31 caracteres**.

Ejemplos:
tiempo, distancia1, caso_A, PI, velocidad_de_la_luz

Componentes Sintácticos

Constantes

1. ***Constantes numéricas. Son valores numéricos, enteros o de punto flotante.*** *Se permiten* también constantes *octales y hexadecimales.*
2. ***Constantes carácter. Cualquier carácter individual encerrado entre apóstrofos (tal como*** 'a', 'Y', ')', '+', etc.) es considerado por C como una constante carácter***.***
3. ***Cadenas de caracteres. Un conjunto de caracteres alfanuméricos encerrados entre*** **comillas** es también un tipo de constante del lenguaje C
4. ***Constantes simbólicas. Las constantes simbólicas tienen un nombre (identificador)***. **No pueden cambiar de valor** a lo largo de la ejecución del programa. Se definen por medio de la palabra clave ***const.***

Componentes Sintácticos

Operadores

- Los *operadores son* ***signos especiales, a veces, conjuntos de dos caracteres que indican*** determinadas **operaciones** a realizar con las variables y/o constantes.
- **operadores:** ***aritméticos*** (+, -, *, /, %), ***de asignación*** *(=, +=, -=, *=, /=),* ***relacionales*** *(==, <, >, <=, >=,* !=), *lógicos (&&, ||, !).*

Ejemplo de uso:

```
espacio = espacio_inicial + 0.5 * aceleracion * tiempo *
    tiempo;
```

Componentes Sintácticos

Separadores

- Están *constituidos por uno o varios* ***espacios en*** **blanco**, **tabuladores**, y **caracteres de nueva línea**. Su papel es ayudar al compilador a descomponer el programa fuente***.***
- ***Es conveniente introducir*** **espacios en blanco** incluso cuando no son estrictamente necesarios, con objeto de **mejorar la legibilidad** de los programas.

Componentes Sintácticos

Separadores

- Permiten introducir ***comentarios en los ficheros fuente que*** contienen el código de su programa.
- Los caracteres **(/*) se emplean para iniciar un comentario introducido entre el código del** programa; el comentario termina con los caracteres **(*/). Por ejemplo:**

```
variable_1 = variable_2; /* En esta línea se asigna a
variable_1 el valor
contenido en variable_2 */
```

- El lenguaje ANSI C también toma como comentario Todo lo que va en cualquier línea del código detrás de la doble barra **(//)** y hasta el final de la línea. **Por ejemplo:**

```
variable_1 = variable_2; // En esta línea se asigna a
// variable_1 el valor
// contenido en variable_2
```

Tipos de Datos

Datos enteros	char	signed char	unsigned char
	signed short int	signed int	signed long int
	unsigned short int	unsigned int	unsigned long int
Datos reales	float	double	long double

CONTROL DEL FLUJO DE EJECUCIÓN
bifurcaciones

- SENTENCIA IF
- SENTENCIA IF ... ELSE
- SENTENCIA IF ... ELSE MÚLTIPLE
- SENTENCIA SWITCH
- SENTENCIAS IF ANIDADAS

CONTROL DEL FLUJO DE EJECUCIÓN
bifurcaciones

SENTENCIA IF

permite ejecutar o no una **sentencia simple o compuesta** según se cumpla o no una determinada condición

```
if (expresión)
sentencia;
```

CONTROL DEL FLUJO DE EJECUCIÓN
bifurcaciones

SENTENCIA IF ... ELSE

permite realizar una *bifurcación,* ***ejecutando una parte u otra del programa*** según se cumpla o no una cierta condición

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

CONTROL DEL FLUJO DE EJECUCIÓN
bifurcaciones

SENTENCIA IF ... ELSE MÚLTIPLE

- permite realizar una ramificación múltiple, **ejecutando *una entre varias partes* del programa** según se cumpla *una entre n condiciones*

```
if (expresion_1)
   sentencia_1;
else if (expresion_2)
   sentencia_2;
else if (expresion_3)
   sentencia_3;
else if (...)
   ...
[else
   sentencia_n;]
```

CONTROL DEL FLUJO DE EJECUCIÓN
bifurcaciones

SENTENCIA SWITCH

- Se evalúa **expresion y se considera el resultado de dicha evaluación. Si dicho** resultado coincide con el valor constante **expresion_cte_1, se ejecuta sentencia_1 seguida de las demás**
- Si se desea ejecutar únicamente una **sentencia_i (y no todo un conjunto de ellas),** basta poner una sentencia **break** a continuación

```
switch (expresion) {
   case expresion_cte_1:
     sentencia_1;
   case expresion_cte_2:
     sentencia_2;
     ...
   case expresion_cte_n:
     sentencia_n;
   [default:
     sentencia;]
}
```

CONTROL DEL FLUJO DE EJECUCIÓN
bifurcaciones

SENTENCIAS IF ANIDADAS

Una sentencia ***if puede incluir otros if dentro de la parte correspondiente a su sentencia, A*** estas sentencias se les llama ***sentencias anidadas***

```
if (a >= b)
if (b != 0.0)
c = a/b;
```

```
if (a >= b)
if (b != 0.0)
c = a/b;
else
c = 0.0;
```

```
if (a >= b) {
if (b != 0.0)
c = a/b;
}
else
c = 0.0;
```

CONTROL DEL FLUJO DE EJECUCIÓN
bucles

- SENTENCIA WHILE
- SENTENCIA FOR
- SENTENCIA DO ... WHILE

CONTROL DEL FLUJO DE EJECUCIÓN

bucles

SENTENCIA WHILE

permite ejecutar repetidamente, *mientras se cumpla una determinada condición*, *una sentencia o bloque de sentencias*

```
while (expresion_de_control)
  sentencia;
```

CONTROL DEL FLUJO DE EJECUCIÓN
bucles

SENTENCIA FOR

for (inicializacion; expresion_de_control; actualizacion)
Sentencia;

se ejecuta ***inicializacion, (una o más*** sentencias que asignan valores iniciales a ciertas variables) o contadores. A continuación se evalúa **expresion_de_control y si es *false se prosigue en la sentencia siguiente a la*** construcción ***for; si es true se ejecutan sentencia y actualización, y se vuelve a evaluar*** **expresion_de_control.**

```
for (pe =0.0, i=1; i<=n; i++){
  pe += a[i]*b[i];
}
```

CONTROL DEL FLUJO DE EJECUCIÓN

bucles

SENTENCIA DO ... WHILE

- funciona de modo análogo a ***while, con la diferencia de que la evaluación de*** **expresion_de_control se realiza al final del bucle, después de haber ejecutado al menos una** vez las sentencias entre llaves

```
do
  sentencia;
while(expresion_de_control);
```

CONTROL DEL FLUJO DE EJECUCION
Sentencias *break, continue, goto*

- **Break** interrumpe la ejecución del bucle
- **Continue** hace que el programa comience el siguiente ciclo del bucle
- **Goto** hace saltar al programa a la etiqueta indicada.

```
sentencias ...
...
if (condicion)
   goto otro_lugar; // salto al lugar indicado por la etiqueta
   sentencia_1;
   sentencia_2;
...
otro_lugar: // esta es la sentencia a la que se salta
   sentencia_3;
...
```

TIPOS DE DATOS DERIVADOS

Punteros

Vectores, matrices y cadenas de caracteres

Estructuras

TIPOS DE DATOS DERIVADOS
Punteros

CONCEPTO DE PUNTERO O APUNTADOR

- El **valor de cada variable** está almacenado en un lugar determinado de la memoria, caracterizado por una ***dirección.*** El ordenador mantiene una ***tabla de direcciones que relaciona el nombre de cada*** variable con su dirección en la memoria
- En C se dispone del ***operador de dirección (&) que permite determinar la dirección de una variable,*** *y de* un tipo especial de **variables destinadas a contener direcciones de variables**. Estas variables se llaman ***punteros***
- ***Declaración de puntero:***

int *direc; // direc contendrá la dirección de cualquier var tipo int

TIPOS DE DATOS DERIVADOS
Punteros

OPERADORES DIRECCIÓN (&) E INDIRECCIÓN (*)

- Para acceder al valor depositado en la zona de memoria a la que apunta un *puntero se debe utilizar el* ***operador indirección (*).***

```
int i, j, *p; // p es un puntero a int
p = &i; // p contiene la dirección de i
*p = 10; // i toma el valor 10
p = &j; // p contiene ahora la dirección de j
*p = -2; // j toma el valor -2
```

TIPOS DE DATOS DERIVADOS
Punteros

OPERADORES DIRECCIÓN (&) E INDIRECCIÓN (*)

Las siguientes sentencias son ilegales:

```
p = &34; // las constantes no tienen dirección
p = &(i+1); // las expresiones no tienen dirección
&i = p; // las direcciones no se pueden cambiar
p = 17654; // habría que escribir p = (int *)17654;
```

existe un **tipo indefinido de punteros** (***void *, o punteros a void),*** *que puede asignarse y* ***al que puede asignarse cualquier tipo de puntero.***

Por ejemplo:

```
int *p;
double *q;
void *r;
p = q; // ilegal
p = r = q; // legal
```

TIPOS DE DATOS DERIVADOS
Punteros

ARITMÉTICA DE PUNTEROS

***No** están permitidas las **operaciones que no tienen sentido con direcciones de variables**, como **multiplicar o dividir**, pero sí otras como* **sumar o restar.** Además estas operaciones se realizan de un modo que no es el ordinario. Así, la sentencia: p = p+1; hace que p apunte a la dirección siguiente.

TIPOS DE DATOS DERIVADOS
Punteros

- **El siguiente ejemplo ilustra la aritmética de punteros:**

```
void main(void) {
int a, b, c;
int *p1, *p2;
void *p;
p1 = &a; // Paso 1. La dirección de a es asignada a p1
*p1 = 1; // Paso 2. p1 (a) es igual a 1. Equivale a a = 1;
p2 = &b; // Paso 3. La dirección de b es asignada a p2
*p2 = 2; // Paso 4. p2 (b) es igual a 2. Equivale a b = 2;
p1 = p2; // Paso 5. El valor del p1 = p2
*p1 = 0; // Paso 6. b = 0
p2 = &c; // Paso 7. La dirección de c es asignada a p2
*p2 = 3; // Paso 8. c = 3
printf("%d %d %d\n", a, b, c); // Paso 9. ¿Qué se imprime?
p = &p1; // Paso 10. p contiene la dirección de p1
*p = p2; // Paso 11. p1= p2;
*p1 = 1; // Paso 12. c = 1
printf("%d %d %d\n", a, b, c); // Paso 13. ¿Qué se imprime?
}
```

TIPOS DE DATOS DERIVADOS
Vectores, matrices y cadenas de caracteres

- Un ***array (también conocido como arreglo, vector o matriz) es un modo de manejar una gran*** *cantidad de datos del mismo tipo bajo un mismo nombre o identificador.*
- La forma general de la **declaración** de un vector es: ***tipo nombre[numero_elementos];***
- *Una* ***cadena de Caracteres*** *es un Vertor tipo char*
- Las ***matrices*** *se declaran con* ***corchetes independientes para cada*** **subíndice**. La forma general de la declaración es:

tipo nombre[numero_filas][numero_columnas];

TIPOS DE DATOS DERIVADOS
Vectores, matrices y cadenas de caracteres

Ejemplos de uso de vectores:

```
double a[10];
a[5] = 0.8;
a[9] = 30. * a[5];
a[0] = 3. * a[9] - a[5]/a[9];
a[3] = (a[0] + a[9])/a[3];
char ciudad[20] = "San Sebastián";
```

TIPOS DE DATOS DERIVADOS
Vectores, matrices y cadenas de caracteres

Inicialización

- – Declarando el array e inicializándolo mediante lectura o asignación por medio de un bucle ***for:***

```
double vect[N];
...
for(i = 0; i < N; i++)
scanf(" %lf", &vect[i]);
...
```

- – Inicializándolo en la misma declaración, en la forma:

```
double v[6] = {1., 2., 3., 3., 2., 1.};
float d[] = {1.2, 3.4, 5.1}; // d[3] está implícito
int f[100] = {0}; // todo se inicializa a 0
int h[10] = {1, 2, 3}; // restantes elementos a 0
int mat[3][2] = {{1, 2}, {3, 4}, {5, 6}};
```

TIPOS DE DATOS DERIVADOS
Estructuras

- Una ***estructura es una forma de agrupar un conjunto de datos de distinto tipo bajo un mismo*** *nombre o identificador*
- El *modelo de una estructura es:*

```
struct alumno {
char nombre[31];
char direccion[21];
unsigned long no_matricula;
unsigned long telefono;
float notas[10];
};
```

TIPOS DE DATOS DERIVADOS
Estructuras

- Para declarar dos variables de tipo **alumno en C se debe utilizar la sentencia** incluyendo las palabras ***struct y alumno***

struct alumno alumno1, alumno2;

struct alumno nuevo_alumno, clase[300];

alumno1 como alumno2 son una estructura, que podrá almacenar: *un nombre de hasta 30 caracteres, una dirección de hasta 20 caracteres, el número de matrícula, el número de teléfono y las notas de las 10 asignaturas*

TIPOS DE DATOS DERIVADOS
Estructuras

Para acceder a los miembros de una estructura se utiliza el ***operador punto (.)***, precedido por el nombre de la *estructura y seguido del nombre del miembro.*

Ejemplo:

alumno1.telefono = 4897944;

alumno1.direccion = "C/ Penny Lane 1,2-A";

clase[264].no_matricula= 48351

TIPOS DE DATOS DERIVADOS
Estructuras

Se pueden definir también ***punteros a estructuras:***

*struct alumno *pun;*

pun = &nuevo_alumno;

el puntero **pun apunta a la estructura nuevo_alumno y esto permite una nueva forma** de acceder a sus miembros utilizando el ***operador flecha (->).***

pun->telefono;

*(*pun).telefono;*

FUNCIONES

- Una ***función es una parte de código independiente del*** programa principal y de otras funciones, que puede ser llamada enviándole datos (o sin enviarle nada)
- Parte esencial del correcto diseño de un programa de ordenador es su ***modularidad****, esto es su* división en partes más pequeñas de finalidad muy concreta. **En C estas partes** de código **reciben el nombre de *funciones.***

FUNCIONES

Definición de una función

tipo_valor_de_retorno nombre_funcion(lista de argumentos con tipos)

{

declaración de variables y/o de otras funciones

codigo ejecutable

return (expresión); // optativo

}

FUNCIONES

- Se pueden definir **Variables locales.** La función puede ver **variables globales** definidas con **extern**
- Los ***argumentos formales*** *son la forma en que* la función recibe valores desde el programa que la llama.
- La función puede devolver al programa que la ha llamado un valor (el ***valor de retorno***). *Si el tipo de valor de retorno es* ***void*** *no devuelve ningun valor.*
- La sentencia ***return*** *permite devolver el control al programa que llama.*

```
double valor_abs(double x)
            {if (x < 0.0)
            return -x;
            else
            return x;}
```

FUNCIONES

Declaración

a) Declaración mediante una ***llamada a la función***

b) ***Declaración*** mediante una ***definición previa de la función***

c) Declaración mediante una ***declaración explícita, previa a la llamada***

```
tipo_valor_de_retorno nombre_func (lista de tipos de arg);
```

FUNCIONES

Llamadas

- La ***llamada*** *a una función se hace incluyendo su* ***nombre en una expresión o sentencia*** del programa principal o de otra función.
- Este nombre debe ir seguido de una lista de *argumentos separados por comas y encerrados entre paréntesis*
- En ejemplo de llamada a la **función *seno:***

```
a = d * sin(alfa) / 2.0;
```

- Por ejemplo, la siguiente sentencia llama a una función que multiplica dos matrices (nxn) **A y B, y** almacena el resultado en otra matriz **C.**

```
prod_mat(n, A, B, C);
```

FUNCIONES

// fichero prueba.c

```
#include <stdio.h>
double valor_abs(double); // declaración
void main (void)
    {
    double z, y;
    y = -30.8;
    z = valor_abs(y) + y*y; // llamada en una expresion
    }
```

FUNCIONES
Paso por valor

```
void permutar(double x, double y) // funcion incorrecta
{
   double temp;
   temp = x;
   x = y;
   y = temp;
}

//La función anterior podría ser llamada y comprobada de la
   siguiente forma:

#include <stdio.h>
void main(void)
{
   double a=1.0, b=2.0;
   void permutar(double, double);
   printf("a = %lf, b = %lf\n", a, b);
   permutar(a, b);
   printf("a = %lf, b = %lf\n", a, b);
}
```

FUNCIONES
Paso por referencia

```
void permutar(double *x, double *y)
{
   double temp;
   temp = *x;
   *x = *y;
   *y = temp;
}

//que puede ser llamada y comprobada de la siguiente forma:

#include <stdio.h>
void main(void)
{
   double a=1.0, b=2.0;
   void permutar(double *, double *);
   printf("a = %lf, b = %lf\n", a, b);
   permutar(&a, &b);
   printf("a = %lf, b = %lf\n", a, b);
}
```

FIN Introducción a C