Programación en C

Juan Ángel Lorenzo del Castillo
juanangel.lorenzo@usc.es

Grupo de Arquitectura de Computadores
Departamento de Electrónica y Computación
Universidad de Santiago de Compostela

CESGA, 11 - 15 de Julio 2011

CESGA Computational Science Summer School 2011

CESGA

USC UNIVERSIDADE DE SANTIAGO DE COMPOSTELA

Índice

1. Introducción
2. Tipos, declaraciones y expresiones
3. Control de flujo
4. Funciones
5. Entrada/salida
6. Arrays y Punteros
7. Tipos avanzados
8. Librerías
9. El preprocesador de C

Índice

1 Introducción
2 Tipos, declaraciones y expresiones
3 Control de flujo
4 Funciones
5 Entrada/salida
6 Arrays y Punteros
7 Tipos avanzados
8 Librerías
9 El preprocesador de C

Motivación y organización del curso

Bienvenidos al *Curso de Programación en C* de la $C^2S^3$2011

- Éste es un curso de programación **en C**: su objetivo es mostrar y enseñar a utilizar los recursos que ofrece el lenguaje. No se enseña algorítmica.
- En el Aula CESGA (*http://aula.cesga.es*, código de curso `PRC`) encontraréis:
  - Los códigos de ejemplo utilizados en estos apuntes.
  - Los ejercicios propuestos en cada uno de los laboratorios que se realizarán al final de cada tema.
  - Enlaces y referencias a recursos útiles en internet.
  - Un foro para la resolución de dudas, comentarios, sugerencias, etc.
- Las tres reglas del curso:
  1. Si una explicación no es suficientemente clara..
  2. Si no se entiende algo...
  3. Si se quiere saber más sobre un concepto en particular...

¡Preguntad!

Por qué estudiar C

Ventajas:

- Permite iniciarse fácilmente en la programación. Es un lenguaje excelente para entender cómo trabajan los programas en la memoria de un ordenador.
- Muy usado en aplicaciones científicas, industriales, sistemas embebidos, simulaciones de vuelo, etc.
- Muy potente, eficiente y flexible. Existen compiladores para casi todos los sistemas conocidos. Todas las versiones de UNIX, Linux y el kernel de MS Windows están escritos principalmente en C.
- Permite derivar fácilmente a otros lenguajes de alto nivel, como C++ y Java, que se apoyan fuertemente en C.

Inconvenientes:

- No dispone de un recolector de basura.
- La seguridad depende casi exclusivamente de la experiencia del programador.
- No proporciona encapsulamiento.
- No proporciona sobrecarga de funciones.

Evolución histórica y características de C

- C es un lenguaje procedimental desarrollado en 1970 por **Dennis Ritchie** en los laboratorios *AT&T Bell*, con el objeto de implementar UNIX y utilidades con el mayor grado posible de independencia de una plataforma hardware específica.
  - Está basado en los lenguajes BCPL (*Basic Combined Programming Language*, por Martin Richards) y B (por Ken Thompson).
- **Características:**
  - Lenguaje de **medio nivel**, más comprensible que el lenguaje ensamblador, que permite abstraer al programador de las complicaciones de la arquitectura subyacente. Al mismo tiempo, permite trabajar a bajo nivel.
  - **Estructurado** (utiliza sólo las estructuras *secuencial*, *condicional* y *repetitiva*), y **modular** (afronta la solución de un problema descomponiéndolo en subproblemas más simples).
  - Permite la **portabilidad** de código fuente.
  - Muy **eficiente**.
- **Brian Kernighan** y **Dennis Ritchie** publicaron una descripción de C en 1978.
  - El C de K&R fue estandarizado por un comité del ANSI (*American National Standard Institute*) con objeto de garantizar su portabilidad entre distintos computadores, dando lugar al **ANSI C**, que es la variante utilizada actualmente.

Estructura básica de un programa en C

Elementos de un programa en C

- Al ser un lenguaje de programación modular, podemos organizar las funciones que componen nuestro código en uno o varios **ficheros de código fuente** (`*.c`). Cada uno de los ficheros puede ser editado y compilado por separado.
- De entre todos los ficheros fuente que constituyan nuestro programa, uno y sólo uno deberá contener una **función** `main()`, que será la primera en ser invocada al ejecutar el programa.
- Podemos incluir uno o varios **ficheros de cabecera** (`*.h`) que contendrán definiciones de variables, funciones, macros y constantes para ser utilizados por uno o varios de los ficheros fuente que componen el programa.

Estructura básica de un programa en C

```
1  /* circulo.c: Calcula el area de una circunferencia *
2   * Uso: ./circulo                                    */
3
4  #include <stdio.h>
5  #include <stdlib.h>
6  #define PI 3.1415926536
7
8  double areaCircular( double );
9
10 int main(int argc,char **argv)
11 {
12   double radio = 10, area = 0.0;
13
14   printf( "    Area de un circulo\n\n" );
15   printf( "     Radio          Area\n"
16           "-------------------------\n" );
17   area = areaCircular( radio );
18   printf( "%10.1f      %10.2f\n", radio, area );
19   return 0;
20 }
21
22 // Parametro:           Radio del circulo
23 // Valor de retorno:    Area del circulo
24 double areaCircular( double r )
25 {
26   return  PI * r * r;
27 }
```

Comentarios

Directivas del preprocesador

Definición de función

Función *main()*
(función principal)

Otras funciones

Estructura básica de un programa en C

- La compilación consiste en utilizar un programa, llamado **compilador**, que nos traduce nuestro código fuente a un código máquina que puede ser entendido por el ordenador.
- Compilamos nuestro código de ejemplo usando `gcc`:

```
gcc  -o circulo circulo.c
```

- El compilador nos generará un fichero ejecutable llamado `circulo`. Si lo ejecutamos:

```
home$ ./circulo
    Area de un circulo

     Radio          Area
------------------------
      10.0         314.16
```

Compilación en Unix/Linux con gcc

El proceso de compilación sigue la siguiente secuencia:

```
gcc -o fich_ejecutable fich_main.c fich_funciones1.c fich_funciones2.c
```

1. El compilador opera sobre una ***unidad de traducción*** formada por un fichero fuente y todos los ficheros de cabecera referenciados por éste.
2. Si no se encuentran errores, se genera un ***módulo*** o ***fichero objeto*** (`*.o`) que contiene el código máquina correspondiente.
3. El compilador repite el proceso para cada una de las unidades de traducción que componen el programa, generando los correspondientes ficheros objeto. Uno y sólo uno de los ficheros debe contener la función **`main()`**, que es la primera que se ejecutará al invocar al programa. Sería equivalente a coger cada uno de los ficheros fuente y generar los ficheros objeto por separado:

```
gcc -c fich_main.c
gcc -c fich_funciones1.c
gcc -c fich_funciones2.c
```

4. Por último, invoca al ***enlazador*** `ld` (*linker*), el cual combina todos los ficheros objeto y las librerías externas necesarias en un único fichero ejecutable. Sería equivalente a:

```
gcc -o fich_ejecutable fich_main.o fich_funciones1.o fich_funciones2.o
```

Compilación en Unix/Linux con gcc

Algunas opciones de `gcc`:

```
gcc -Wall programa.c
```

- **-Wall:** Indica al compilador que muestre todos los avisos de los posibles problemas que pueda encontrar en el código. Por defecto, el ejecutable generado tiene el nombre `a.out`. Lo podemos ejecutar desde línea de comandos escribiendo `./a.out`

```
gcc -o programa programa.c
```

- **-o:** El ejecutable tendrá el nombre `programa` (en vez de `a.out`). Lo podemos ejecutar desde línea de comandos escribiendo `./programa`

```
gcc -S programa.c
gcc -S -fverbose-asm programa.c
```

- **-S:** Genera el código ensamblador y lo almacena en `programa.s`. La opción `-fverbose-asm` incluye los nombres de las variables como comentarios en el código ensamblador.

Compilación en Unix/Linux con gcc

Algunas opciones de `gcc` (II):

```
-I directorio[: directorio[...]]
```

- **-I:** Indica que se busquen los ficheros de cabecera requeridos por el programa (con *#include*) en los directorios especificados, aparte de en los directorios *include* estándar del sistema. El orden de búsqueda típico para incluir ficheros de cabecera es:
  1. El mismo directorio en el que se encuentra el fichero fuente. Busca los ficheros de cabecera especificados entre comillas dobles.
  2. Los directorios especificados por la opción `-I`, en el orden indicado en la opción.
  3. Los directorios especificados en las variables de sistema C_INCLUDE_PATH y CPATH.
  4. Los directorios `include` por defecto del sistema.

Compilación en Unix/Linux con gcc

Algunas opciones de `gcc` (III):

- Opciones de optimización:
  - **-O0** : Desactiva todas las opciones de optimización
  - **-O,-O1** : Intenta hacer el programa más pequeño y más rápido, pero sin aumentar excesivamente el tiempo de compilación.
  - **-O2** : Aplica casi todas las técnicas de optimización soportadas buscando un compromiso entre el tamaño del código y la velocidad de ejecución. Aumenta el tiempo de compilación.
  - **-O3** : Optimizaciones de O2 más algunas mejoras adicionales.
  - **-Os** (*optimization for size*): Técnicas de O2 excepto aquellas que provocan un aumento en el tamaño del código.

Para consultar todas las opciones disponibles: `man gcc`.

El comando `GNU make`

- Compilar un programa grande y complejo, compuesto por varios ficheros fuente, puede ser complicado y tedioso.
- **`make`** es una utilidad que permite automatizar y gestionar el proceso de compilación de programas de cualquier tamaño y complejidad, analizando las dependencias entre los ficheros involucrados.
- Además, cuando se compila repetidamente, `make` comprueba qué ficheros no han sido modificados, para no compilarlos de nuevo innecesariamente.
- Reglas:
  - Un ejecutable es un *objetivo* (***target***) que debe volver a crearse cada vez que alguno de los ficheros objeto cambia.
  - Los ficheros objeto son los ***prerrequisitos***.
  - Al mismo tiempo, los ficheros objeto son también ***objetivos intermedios***, que deben ser recompilados si su código fuente o alguno de los ficheros de cabecera cambia.
- Las reglas de compilación para cada uno de los objetivos conforman el *script de comandos*, que `make` ejecuta para construir el programa.
- Las reglas de compilación tienen una sintaxis especial y se agrupan en un fichero de texto: el **Makefile**

El comando GNU make

- Para utilizar una serie de reglas contenidas en un fichero, se utiliza la opción `-f`:

```
make -f fichero_reglas
```

- Sin la opción `-f`, `GNU make` buscará primero un fichero llamado `GNUmakefile`. Si no lo encuentra, otro llamado `makefile`. En su defecto, buscará `Makefile`.
- Ejemplo:

```
1  # Makefile basico para "prodmat".
2
3  CC = gcc
4  CFLAGS = -Wall -g -std=c99
5  LDFLAGS = -lm
6
7  all: prodmat
8
9  prodmat : prodmat.o funcMatriz.o producto.o
10         $(CC) $(LDFLAGS) -o $@ $^
11
12 prodmat.o : prodmat.c
13         $(CC) $(CFLAGS) -o $@ -c $<
14
15 funcMatriz.o : funcMatriz.c
16         $(CC) $(CFLAGS) -o $@ -c $<
17
18 producto.o: producto.c
19         $(CC) $(CFLAGS) -o $@ -c $<
20
21 clean:
22         rm *.o
```

El comando GNU make

- Cada *target* (*prodmat*, *prodmat.o*, *funcMatriz.o*, *producto.o*) debe comenzar al principio de línea.
- Cada línea de comandos debe comenzar por una tabulación.
- El *target* en cada regla dice: *si algún target es más antiguo que algún prerrequisito, ejecuta la línea de comandos*. Para cada uno de los prerrequisitos comprueba también si éstos, a su vez, tienen otros prerrequisitos.
- Para cada regla:

```
prodmat : prodmat.o funcMatriz.o producto.o
        $(CC) $(LDFLAGS) -o $@ $^
```

  - Cada variable debe ser invocada con el carácter dólar y entre paréntesis.
  - $@: El fichero *target*
  - $<: Primer prerrequisito
  - $^: La lista de prerrequisitos

La función `printf`

- Muestra una salida formateada por el dispositivo estándar de salida (la pantalla, típicamente)

```
#include <stdio.h>
int printf ( const char * formato , ... );
```

- Convierte diferentes tipos de datos en cadenas de caracteres de modo que puedan ser mostrados por pantalla, según se indique mediante ciertos símbolos en la cadena `formato`.
- Los datos se mostrarán en el mismo orden en el que se indiquen como parámetros en la parte opcional (`...`) de la función.
- Ejemplo:

```
1  /* ejprintf.c */
2  #include <stdio.h>
3
4  int main()
5  {
6    int entero = 3;
7    float pf = 3.4;
8    double db = 3.1415927;
9    char caracter = 'a';
10   char texto[] = "texto";
11
12   printf("entero: %d\t dir.de entero: %p\n",entero,&entero);
13   printf("float: %2.2f\t double: %f\n",pf,db*2);
14   printf("caracter: %c\t cadena: %s\n",caracter,texto);
15 }
```

La función `printf`

- Sintaxis de conversión de los argumentos:

  `%[flags][field width][.precision][length modifier]specifier`

  - Significado:

flag	Significado
`+`	Añade un signo “+” delante de los números positivos
`' '`	Añade un espacio delante de los números positivos
`-`	Alínea la salida a la izquierda
`0`	Rellena la salida con ceros a la iquierda. Incompatible con ’-’

  - ***field width*** es un número entero que especifica el número mínimo de caracteres que los datos ocuparán en la salida. Si es mayor que el tamaño de los datos a representar, se rellena con espacios (o con ceros, si se indica el flag ’0’. Ej:

```
printf("float: %05.2f\n",3.4); //Devuelve 03.40
```

  - Para los especificadores de conversión de punto flotante (`f,F,e,E,a` y `A`), ***precision*** indica el número de decimales a mostrar. Por defecto, es 6. Para enteros (`u,d,i,x` y `o`), especifica el mínimo número de caracteres a mostrar. Para cadenas de texto (`s`), especifica la longitud máxima de texto a mostrar.

La función `printf`

- Sintaxis de conversión de los argumentos:
  `%[flags][field width][.precision][length modifier]specifier`
  - ***length modifier*** modifica los especificadores de conversión de la siguiente forma:

flag	Con los especificadores	Significado
`hh`	d, i, o, u, x, o X	`signed char` o `unsigned char`
`h`	d, i, o, u, x, o X	`short int` o `unsigned short int`
`l`	d, i, o, u, x, o X	`long int` o `unsigned long int`
`ll`	d, i, o, u, x, o X	`long long` o `unsigned long long`
`L`	a, A, e, E, f, F, g, o G	`long double`

  - ***specifier*** indica el tipo de argumento y cómo es convertido. El argumento correspondiente de la función deberá tener un tipo compatible:

Conversor	tipo de argumento	Salida
`d,i`	`int`	Decimal
`u`	`unsigned int`	Decimal
`o`	`unsigned int`	Octal
`x,X`	`unsigned int`	Hexadecimal
`f,F`	`float` o `double`	Punto flotante
`e,E`	`float` o `double`	Notación exponencial
`g,G`	`float` o `double`	Punto flotante o notación exponencial, la que sea más corta
`a,A`	`float` o `double`	Notación exponencial hexadecimal
`c`	`char` o `int`	Un carácter
`s`	`char *`	El string apuntado por el puntero `s`
`p`	Cualquier tipo puntero	Un símbolo %

Laboratorio 1

(Tomada de http://tux.crystalxp.net/)

Índice

1 Introducción
2 Tipos, declaraciones y expresiones
3 Control de flujo
4 Funciones
5 Entrada/salida
6 Arrays y Punteros
7 Tipos avanzados
8 Librerías
9 El preprocesador de C

Tipos de datos y variables

- Un programa almacena y procesa diferentes tipos de datos (enteros, punto flotante, etc.). Para ello, el compilador necesita saber qué tipo de dato representa un valor dado.
- Una **variable** es una zona de memoria cuyo contenido representa un valor. Cada variable se **declara** (identifica) mediante un **nombre de variable**.
- El **tipo** de una variable determina cuánto espacio ocupa y qué clase de datos puede almacenar esa variable

Reglas

- Deben empezar con una letra o "_" seguido de letras, dígitos o "_"
- Distingue mayúsculas y minúsculas (`promedio` ≠ `Promedio`)
- No se pueden utilizar *palabras reservadas* (`int`, `char`)
- Declaración de variables:
  `tipo nombre_variable`
  Si no está inicializada, el valor es *indefinido*

```
//Nombres de variable válidos
  promedio
  pi
  numero_de_estudiantes
  _temperatura

//Incorrectos
  3valores
  valor#
  prueba dos
  int
```

```
//Declaración variables
int promedio;
//inicialización
float pi = 3.1415926;
char *estudiante = "Juan García";
```

Tipos de datos y variables

Tipos en C

- Tipos básicos:
  - Enteros
  - Punto flotante reales y complejos
- Tipos avanzados:
  - Enumeraciones
  - El tipo **void**
  - Tipos derivados:
    - Punteros
    - Arrays
    - Estructuras
    - Uniones
    - Funciones

Tipos básicos de datos y variables

Tipos Enteros con signo (*signed integer*):

Tipo	Tamaño (bytes)	Rango*	Sinónimos
`signed char`	1	[-128, 127]	-
`char`	Tamaño y rango como *signed char* o *unsigned char*		
`int`	2	[-32.768, 32.767]	`signed`,`signed int`
	4	[-2.147.483.648, 2.147.483.647]	`signed`,`signed int`
`short`	2	[-32.768, 32.767]	`short int`, `signed short`, `signed short int`
`long`	4	[-2.147.483.648, 2.147.483.647]	`long int`, `signed long`, `signed long int`
`long long`	8	[-9.223.372.036.854.775.808, 9.223.372.036.854.775.807]	`long long int`, `signed long long`, `signed long long int`

Tipos Enteros sin signo (*unsigned integer*):

Tipo	Tamaño (bytes)	Rango*	Sinónimos
`_Bool`	1	[-128, 127]	`bool` (definido en `stdbool.h`)
`unsigned char`	1	[0, 255]	-
`unsigned int`	2 o 4	[0, 65.535 o 2.147.483.647]	`unsigned`
`unsigned short`	2	[0, 65.535]	`unsigned short int`
`unsigned long`	4	[0, 4.294.967.295]	`unsigned long int`
`unsigned long long`	8	[0, 18.446.744.073.709.551.615]	`unsigned long long int`

*Ejemplo de tamaño típico para UNIX 32 bits. Para una arquitectura y compilador específicos deberá consultarse utilizando el operador `sizeof()`

Tipos básicos de datos y variables

Tipos reales (punto flotante):

Tipo	Tamaño (bytes)	Rango*	Valor positivo más pequeño	Precisión
`float`	4	±3.4E+38	1.2E-38	6 dígitos
`double`	8	±1.7E+308	2.3E-308	15 dígitos
`long double`	10	±1.1E+4932	3.4E-4932	19 dígitos

- Los tipos en PF representan valores no enteros con un punto decimal en cualquier posición.
- Un valor en PF tiene una *precisión* limitada, dada por el formato binario usado para representarlo y la cantidad de memoria utilizada para almacenarlo.
- Ejemplo: *Precisión de 6 dígitos* significa que es posible almacenar un número real de 6 dígitos decimales, independientemente de la posición del punto decimal. Los valores 123.456.000 y 0.00123456 pueden ser almacenados en un tipo con 6 dígitos de precisión.

*Se pueden utilizar las macros definidas en `float.h` para obtener el rango de valores y la precisión del tipo *float*.

Tipos básicos de datos y variables

Tipos reales (punto flotante):

- En C, las operaciones aritméticas con números en FP se realizan internamente con precisión *double* o superior:

```
float altura = 1.2345, anchura = 2.3456;
      // Float tiene precisión simple.
double area = altura * anchura;
      // El cálculo se realiza con precisión
      // double o superior
```

- Si se asigna el resultado a una variable `float`, el valor es redondeado.

Tipos reales complejos (C99):

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

- Coordenadas cartesianas ($z = x + y \cdot i$)
- Cada uno de estos tipos tiene el mismo tamaño que un array de dos elementos `float`, `double` o `long double`

```
#include <complex.h>
// ...
double complex z = 1.0 + 2.0 * I;
z *= I;     // Rotamos z 90° en sentido
            // antihorario sobre el origen.
```

Cualificadores de tipo

- Es posible añadir cualificadores para modificar el tipo de una variable en una declaración.
- Una declaración puede contener cualquier número de cualificadores de tipo en cualquier orden.
- Los cualificadores disponibles son:

const: Un objeto cuyo tipo es cualificado con `const` es constante. El programa no puede modificarlo tras su definición.

volatile: Indica que el tipo cualificado así podría ser modificado por otros procesos o eventos. Obliga al compilador a que el programa vuelva a leer el valor del objeto cualificado cada vez que se use, aunque éste no haya cambiado desde el último acceso.

restrict: Sólo se aplica a punteros. Se introdujo en C99 e indica al compilador que, si el objeto apuntado es modificado, no podrá ser accedido de otra forma más que mediante ese puntero. Permite al compilador aplicar ciertas técnicas de optimización.

Conversiones de tipo (casting)

Conversiones de tipo implícitas

- Cuando en una expresión se mezclan variables de distintos tipos, C realiza una serie de conversiones implícitas, ya que es necesario que los operandos tengan el mismo tipo.
- Si aparecen dos tipos diferentes de constantes y/o variables en una expresión relacionadas por un operador, el compilador convierte los operandos al tipo del de mayor rango, según la siguiente jerarquía (mayor a menor rango):

```
long double > double > float > unsigned long > long > unsigned int > int > char
```

- Otro tipo de casting implícito ocurre cuando se asigna a una variable el resultado de una expresión. En ese caso, la expresión se convierte automáticamente al tipo de la variable. Si el tipo de la variable es de menor rango que el de la expresión, podemos perder información.

Conversiones de tipo (casting)

Conversiones de tipo explícitas

- En ocasiones es necesario convertir explícitamente el tipo de una variable a otro tipo. El formato es:

```
(tipo) expresión
```

que calcula el valor de `expresión` y lo convierte al tipo especificado.

- Es especialmente útil cuando se trabaja con variables de tipo entero y punto flotante. Por ejemplo:

```
int sum = 22, count = 5;
double mean = sum / count;
```

La división de los enteros `sum` y `count` devuelve un entero. Los decimales se truncan ($4.4 \rightarrow 4$) y el valor asignado a `mean` es el entero 4 convertido implícitamente a `double` (4.0). En cambio, si hacemos:

```
int sum = 22, count = 5;
double mean = (double)sum / count;
```

El casting explícito de `sum` a un tipo de rango mayor a `int` obliga a que el otro operando se convierta implícitamente a ese tipo. El resultado es un `double` y se asigna correctamente a `mean`.

Expresiones y Operadores

Operadores:

- Un **operador** es un carácter o grupo de caracteres que actúa sobre una, dos o más variables para realizar una operación y devolver un resultado. Pueden ser **unarios**, **binarios** o **ternarios**, según actúen sobre uno, dos o tres operandos, respectivamente.

Operadores aritméticos:

Operador	Significado	Ejemplo	Resultado
`*`	Multiplicación	`x * y`	Producto de *x* e *y*
`/`	División	`x / y`	Cociente de *x* entre *y*
`%`	Módulo	`x % y`	Resto de la división entera de *x* entre *y*
`+`	Adición	`x + y`	Suma de *x* e *y*
`-`	Sustracción	`x - y`	Diferencia de *x* e *y*
`+(unario)`	Signo positivo	`+x`	Valor de *x*
`-(unario)`	Signo negativo	`-x`	Negación aritmética de *x*

Operadores de asignación (atribuyen a una variable el resultado de una expresión o el valor de otra variable):

Operador	Significado	Ejemplo	Resultado
`=`	Asignación simple	`x = y`	Asigna a *x* el valor de *y*
`+= -= *= /= %= &= ^= |= ≪= ≫=`	Asignación compuesta	`x *= y`	Para un operador *op* aritmético o binario, *x op=y* equivale a *x=x op (y)*

Expresiones y Operadores

Operadores incrementales:

Operador	Significado	Resultado
`x++`	Incremento	Incrementa x en 1 (como $x = x + 1$)
`++x`		
`x--`	Decremento	Decrementa x en 1 (como $x = x - 1$)
`--x`		

Valor de la expresión
El valor de x++ es el valor que *x* tenía antes de ser incrementado (primero se obtiene *x* y después se incrementa). El valor de $++x$ es el valor que tiene *x* después de ser incrementado (primero se incrementa y después se obtiene *x*)
El valor de `x--` es el valor que *x* tenía antes de ser decrementado (primero se obtiene *x* y después se decrementa). El valor de `--x` es el valor que tiene *x* después de ser decrementado (primero se decrementa y después se obtiene *x*)

```
i = 2; j = 2;
m = i++;   // ahora m=2 e i=3
n = ++j;   // ahora n=3 y j=3
```

Operadores relacionales o comparativos: siempre binarios, devuelven un valor de tipo `int` con 1 (*true*) o 0 (false)

Operador	Significado	Ejemplo	Resultado
`<`	menor que	`x < y`	1 si *x* menor que *y*. 0 en caso contrario.
`>`	mayor que	`x > y`	1 si *x* mayor que *y*. 0 en caso contrario.
`<=`	menor o igual que	`x <= y`	1 si *x* menor o igual que *y*. 0 en caso contrario.
`>=`	menor que	`x >= y`	1 si *x* mayor o igual que *y*. 0 en caso contrario.
`==`	igual a	`x == y`	1 si *x* igual a *y*. 0 en caso contrario.
`!=`	distinto de	`x != y`	1 si *x* distinto de *y*. 0 en caso contrario.

Expresiones y Operadores

Operadores lógicos: Permiten combinar los resultados de los operadores relacionales. Devuelven un valor de tipo `int` con 1 (*true*) o 0 (false)

Operador	Significado	Ejemplo	Resultado
`&&`	AND lógico	`x && y`	1 si *x* e *y* distintos de cero. 0 en caso contrario.
`||`	OR lógico	`x || y`	0 si alguno de *x* e *y* iguales de cero. 1 en caso contrario.
`!`	NOT lógico	`!x`	1 si *x* es igual a cero. 0 en caso contrario.

```
(promedio < -0.5) || (promedio > 0.8)
!((promedio >= -0.5) && (promedio <= 0.8))
```

Operadores booleanos a nivel de bit:

Operador	Significado	Ejemplo	Resultado (para cada bit)
`&`	AND binario	`x & y`	1 si *x* e *y* igual a 1. 0 si alguno o ambos igual a cero.
`|`	OR binario	`x | y`	1 si *x*, *y* o ambos igual a uno. 0 si ambos igual a cero.
`^`	XOR	`x ^ y`	1 si *x* o *y* (pero no ambos) igual a uno. 0 en otro caso.
`~`	NOT binario (C1)	`~x`	1 si cero en *x*. 0 si uno en *x*.

```
int a = 6;    // 0 ... 0 0 1 1 0
int b = 11;   // 0 ... 0 1 0 1 1

a & b         // 0 ... 0 0 0 1 0
a | b         // 0 ... 0 1 1 1 1
a ^ b         // 0 ... 0 1 1 0 1
~a            // 1 ... 1 1 0 0 1
```

```
a &= ~0x20;    // Pone a 0 el bit 5 de a

int mask = 0xC;
a |= mask;  // Pone a 1 los bits 2 y 3 de a
a ^= mask;  // Invierte los bits 2 y 3 de a
```

Expresiones y Operadores

Operadores de desplazamiento a nivel de bit (ambos operandos enteros):

Operador	Significado	Ejemplo	Resultado
«	Desplazamiento a la izquierda	`x « y`	Cada bit en `x` es desplazado `y` posiciones a la izquierda
»	Desplazamiento a la derecha	`x » y`	Cada bit en `x` es desplazado `y` posiciones a la derecha

```
unsigned long n = 0xB,   // 0 ... 0 0 0 1 0 1 1
              result = 0;
result = n << 2;         // 0 ... 0 1 0 1 1 0 0
result = n >> 2;         // 0 ... 0 0 0 0 0 1 0
```

- **Desplazamientos a la izquierda:** Los bits que ”entran“ por la derecha son ceros. Los bits que ”salen“ por la izquierda se pierden. Si el operando tiene un tipo `unsigned`, `x « y` equivale a la multiplicación aritmética $x * 2y$
- **Desplazamientos a la derecha:** Bits que ”entran“ por la izquierda:
  - 0 si el operando tiene tipo `unsigned` o `signed` con valor no negativo. `x » y` equivale a la división aritmética $x/2y$
  - Si el operando es negativo, depende del compilador: puede ser 0 o el valor del bit de signo

Expresiones y Operadores

Otros operadores:

Operador	Significado	Ejemplo	Resultado
`&`	Dirección de memoria	`&x`	Puntero a `x`.
`*`	Indirección	`*p`	Contenido de la posición de memoria a la que apunta `p`.
`.`	Designador de miembro de estructura o unión	`x.y`	El miembro `y` de la estructura o unión `x`
`->`	Designador por referencia de miembro de estructura o unión	`p->y`	El miembro `y` de la estructura o unión a la que apunta `p`
`sizeof`	Tamaño en bytes de una variable o tipo	`sizeof(x)`, `sizeof(double)`	Tamaño en bytes de la variable `x` o del tipo **`double`**
`()`	Llamada a una función	`log(x)`	Pasa el control a la función especificada, con los parámetros indicados
`(tipo)`	Conversión explícita (*cast*)	`(short)x`	El valor de `x` convertido al tipo especificado
`?:`	Evaluación condicional	`x ? y : z`	El valor de `y` si `x` es verdadero. En caso contrario, valor de `z`
`,`	Evaluación secuencial	`x, y`	Evalúa primero `x`, después `y`. El resultado global de la expresión es el valor de `y`

sizeof

`sizeof` no es una función, sino un operador (por eso no se encuentra en `man 3 sizeof`). Formas de uso:

```
sizeof expresión unitaria // paréntesis no necesarios
sizeof(tipo) //con paréntesis
```

Expresiones y Operadores

Precedencia de operadores:

Mayor

() [] -> .
++ -- ! sizeof (tipo) +(unario) -(unario) *(indir.) &(dirección)
* / %
+ -
< <= > >=
== !=
&&
||
?:
= += -= *= /=
,

⇕

Menor

Expresiones y Operadores

Expresiones:

- Una **expresión** consiste en una secuencia de constantes, variables y operadores que el programa evalúa para realizar sobre ellos las operaciones indicadas.
- El **tipo de una expresión** es el tipo del valor que resulta cuando la expresión es evaluada.
- Una expresión compleja puede estar formada por otras más sencillas (expresiones primarias) y puede contener paréntesis de varios niveles agrupando distintos términos.

Expresiones aritméticas: Formadas por variables y/o constantes, y distintos operadores aritméticos e incrementales ($+$, $-$, $/$, $\%$, $++$, $--$):

```
x=(-b+sqrt((b*b)-(4*a*c)))/(2*a);
```

Expresiones lógicas: Formadas por valores lógicos (*true = 1* y *false = 0*) y operadores lógicos (`||`, `&&`, `!`) y/o relacionales ($>$, $<$, $\leq$, $\geq$, $==$, $!=$). El tipo será lógico

```
a = ((b>c)&&(c>d))||((c==e)||(e==b));
```

Expresiones generales: Basándose en que el resultado de una operación lógica es siempre un valor numérico (0 o 1), combina expresiones y operadores de distintos tipos. Una expresión aritmética puede aparecer como sub-expresión de una expresión lógica y viceversa:

```
(a - b*2.0) && (c != d)
```

Sentencias

Sentencias:

- Son unidades completas, ejecutables en sí mismas, compuestas por declaraciones y expresiones.

Sentencias simples: Las que terminan con "`;`". Por ejemplo, las declaraciones o expresiones aritméticas:

```
int tmp;
area = pi * radio * radio;
```

Sentencia vacía: Ocupa un lugar en el programa pero no realiza ninguna acción. Es el carácter "`;`".

Sentencias compuestas o bloques: Conjunto de declaraciones y sentencias (simples y compuestas) agrupadas dentro de llaves:

```
{  double resultado = 0.0, x = 0.0;    // Declaraciones
   static long status = 0;
   extern int limite;

   ++x;                                 // Sentencia
   if ( status == 0 )
   {                                    // Nuevo bloque
      int i = 0;
      while ( status == 0 && i < limite )
      {  /* ... */  }                  // Otro bloque
   }
   else
   {  /* ... */  }                     // Otro bloque más
}
```

Constantes

Constantes:

- Una **constante** o **literal** denota un valor fijo (entero, de punto flotante, carácter o string) cuyo valor es definido en tiempo de compilación y no puede ser cambiado.

Constantes enteras: Pueden expresarse en notación decimal, octal o hexadecimal. El tipo se determina automáticamente según su magnitud, o de modo explícito mediante determinados caracteres.

Explícitamente:

Constante entera	Tipo	Notación
`0x200`	`int`	Hexadecimal
`512U`	`unsigned int`	Decimal
`011L`	`long`	Octal
`0Xf0fUL`	`unsigned long`	Hexadecimal
`0777LL`	`long long`	Octal
`0xAAAllu`	`unsigned long long`	Hexadecimal

Automáticamente: Si el valor excede el tamaño de un determinado tipo, el compilador le asigna el tipo siguiente en tamaño según la siguiente jerarquía:

- Constantes decimales: `int` → `long` → `long long`
- Constantes octales y hexadecimales:
  `int` → `unsigned int` → `long` → `unsigned long` → `long long` → `unsigned long long`

Constantes

Constantes de punto flotante: Pueden expresarse en notación decimal o hexadecimal. Si no se especifica un tipo, por defecto se toman como `double`

Notación decimal

Constante PF	Valor	Tipo
`10.0`	10	`double`
`123.456F`	123.456	`float`
`1.54e-5`	1.54×10^{-5}	`double`
`.0075`	0.0075	`double`
`987E7L`	987×10^{7}	`long double`

Notación hexadecimal: Tienen la ventaja de poder almacenarse en el formato binario en punto flotante del ordenador de forma exacta, sin redondeo.

Constante PF	Valor
`0xa.fP-10`	$(10+15/16) \times 2^{-10}$
`0xA.Fp-10`	$(10+15/16) \times 2^{-10}$
`0x5.78p-9`	$(5+7/16+8/16^2) \times 2^{-9}$
`0xAFp-14`	$(10*16+15) \times 2^{-14}$
`0x.02BCp0`	$(2/16^2+11/16^3+16/16^4) \times 2^{0}$

Constantes

Constantes carácter: Uno o más caracteres encerrados entre comillas simples:

```
'a' 'JK' '3' '*'
```

En C no existen constantes de tipo **char** sino que, en realidad, son constantes enteras (**int**). El valor de una constante carácter es su valor según el código ASCII (Ej. la constante `'8'` tiene un valor ASCII de 56).

Para representar caracteres no imprimibles o con un significado especial, como las comillas simples (`'`), se utilizan las **secuencias de escape** mediante una barra invertida (\):

Secuencia de escape	Valor
`\a`	Sonido de alerta
`\b`	Retroceso
`\f`	Salto de página
`\n`	Salto de línea
`\r`	Retroceso
`\t`	Tabulación horizontal
`\v`	Tabulación vertical
`\\`	Barra invertida
`\'`	Comilla simple
`\"`	Comillas dobles
`\0`	Carácter nulo (NULL)
`\?`	Carácter de interrogación
`\x9AF`	El carácter con el código hexadecimal 9AF
`\o33`	El carácter con el código octal 33

Constantes

Cadenas de caracteres: Cualquier secuencia de caracteres y/o caracteres de escape encerrados entre comillas dobles:

```
"Hola Mundo\n"
"Esto es una \"Cadena de caracteres\".\n "
```

El compilador añade siempre un carácter nulo (\0) al final de cada cadena para indicar el final de ésta. Por tanto, el tamaño de la cadena será de un byte por cada carácter más uno por el carácter nulo.
Una cadena de caracteres es un array estático de elementos tipo `char`. Por tanto, se pueden utilizar para inicializar un array:

```
char ruta_doc[128] = ".\\share\\doc";
printf("\aConsultar la documentación en el directorio \"%s\"\n", ruta_doc);
```

para inicializar un puntero a `char`:

```
char *ptr = "Hola Mundo"; // ptr apunta a "H"
```

o para inicializar un array de punteros a `char`:

```
char *msg_error[] = {"Código de error desconocido\n",
                     "Memoria insuficiente\n",
                     "Acceso ilegal a memoria\n"};
```

Constantes

Cualificador `const`: Se utiliza en la declaración e inicialización de una variable o array para indicar que esa variable o los elementos del array no pueden cambiar de valor:

```
const int x = 30;
const int y[] = {1, 2, 3, 4};
```

Se suele utilizar por motivos de eficiencia, cuando se pasan argumentos por referencia a funciones y no se desea que dichos argumentos sean modificados por éstas.

Directiva `define`: Aunque tiene más usos, permite definir una constante, y el preprocesador sustituirá todas sus ocurrencias en el código durante la compilación:

```
#define TAM_MAX_NOMBRE  256
#define SIZE_BUFFER  (4*256)
```

Laboratorio 2

(Tomada de http://tux.crystalxp.net/)

Índice

1 Introducción
2 Tipos, declaraciones y expresiones
3 Control de flujo
4 Funciones
5 Entrada/salida
6 Arrays y Punteros
7 Tipos avanzados
8 Librerías
9 El preprocesador de C

Bifurcaciones

Las **bifurcaciones** o *sentencias de selección* permiten dirigir el flujo de un programa dependiendo de una condición dada.

Sentencia `if`:

Formato: `if ( expresión ) sentencia1 [ else sentencia2 ]`

Se evalúa `expresión`. Si es `1` (`true`), se ejecuta `sentencia1`. Si es `0` (`false`) se salta `sentencia1`, se ejecuta `sentencia2` en caso de existir un `else` y se continúa en la siguiente línea.

Ejemplos (I):

```
1  if ( exp == 0 ) return 1.0;
2  else return base * power( base, exp-1 );
```

```
1  if ( n > 0 )
2     if ( n % 2 == 0 )
3        puts( "n es positivo y par" );
4     else
5        puts( "n es positivo e impar" );
```

Bifurcaciones

Sentencia `if`:

Ejemplos (II):

```
1  if ( n > 0 )
2  {
3     if ( n % 2 == 0 )
4        puts( "n es positivo y par" );
5  }
6  else
7        puts( "n es negativo o cero" );
```

```
1  k = (z>9) ? 3 : (8+z);
```

```
1  printf ( "Tienes %d item%s.\n", n, n==1 ? "" : "s");
```

```
1  double spec = 10.0, medido = 10.3, diff;
2  diff = medido - spec;
3
4  if ( diff >= 0.0 && diff < 0.5 )
5    printf( "Desviación hacia arriba: %.2f\n", diff );
6  else if ( diff < 0.0 && diff > -0.5 )
7    printf( "Desviación hacia abajo: %.2f\n", diff );
8  else
9    printf( "Deviación fuera de límite!\n" );
```

Bifurcaciones

Sentencia `switch`:

Formato:

```
switch (expresion)
{
  case expresion_constante_1:
       sentencia_1;
  case expresion_constante_2:
       sentencia_2;
  ...
  case expresion_constante_n:
       sentencia_n;
  [default:
       sentencia_default;]
}
```

Se evalúa `expresión`. Si es igual a `expresion_constante_1` se ejecutan `sentencia_1`, `sentencia_2`, etc. Si es igual a `expresion_constante_2`, se ejecutan `sentencia_2`, `sentencia_3`, etc. y así sucesivamente. Si no coincide ninguna, se ejecuta `sentencia_default`.

Bifurcaciones

Sentencia `switch`:

Ejemplos:

```
1   switch ( menu( ) )
2   {
3     case 'a':
4     case 'A':  action1( );
5                break;
6     case 'b':
7     case 'B':  action2( );
8                break;
9     default:   putchar( '\a' );
10  }
```

```
1   switch ( var )
2   {
3     case A1: {
4                int a = 5;
5                /* ... */
6              }
7              break;
8     case A2:
9              /* ... */
10  }
```

`break` permite terminar la ejecución de la sentencia `switch`

Bucles

Sentencia `while`:

Formato: `while ( expresión ) sentencia`

Se evalúa `expresión`. Si es `1` (`true`), se ejecuta `sentencia` y se vuelve a evaluar `expresión`. Si es `false` se salta `sentencia`.

Ejemplo:

```
1  /* Cuenta de 1 a n */
2  int i = 1;
3  int n = 10;
4  while(i <= n)
5  {
6    printf("%d\n",i);
7    i++;
8  }
```

Bucles

Sentencia `do-while`:

Formato: `do sentencia; while( expresión )`

Se ejecuta `sentencia`. A continuación, se evalúa `expresión`. Si es `1` (`true`), se vuelve a ejecutar `sentencia`.

Ejemplo:

```
1  /* Cuenta de 1 a n */
2  int i = 1;
3  int n = 10;
4
5  do{
6    printf("%d\n",i);
7    i++;
8  }while(i <= n);
```

Siempre se ejecuta la primera iteración como mínimo

Bucles

Sentencia `do-while`:

Ejemplo: *Algoritmo de ordenación por burbuja*

```
1   /* bubbleSort.c */
2   void bubbleSort( float arr[ ], int len )
3   {
4     int isSorted = 0;
5     do
6     {
7        float temp;         // Para intercambiar valores.
8        isSorted = 1;
9        --len;
10       for ( int i = 0;  i < len;  ++i )
11         if ( arr[i] > arr[i+1] )
12         {
13            isSorted = 0;       // Aún no finalizado.
14            temp = arr[i];      // Intercambia valores adyacentes.
15            arr[i] = arr[i+1];
16            arr[i+1] = temp;
17         }
18    } while ( !isSorted );
19  }
```

Bucles

Sentencia `for`:

Formato:

```
for (inicializacion; expresion_de_control; actualizacion)
    sentencia;
```

Inicialización: Se evalúa una sóla vez, antes de ejecutarse la expresión de control, para hacer las inicializaciones necesarias.

Expresión de control: Se evalúa en cada iteración. La ejecución del bucle termina cuando esta expresión devuelve `false`.

Actualización: Ajuste, tal como el incremento de un contador. Se ejecuta después de cada iteración y antes de que `expresion_de_control` se evalúe de nuevo.

Bucles

Sentencia `for`:

Ejemplo: *Contador*

```
1  /* contador.c */
2  #include<stdio.h>
3
4  int main()
5  {
6       /* Cuenta de 0 a n-1 */
7      int i;
8      int n = 10;
9      for(i=0; i < n; i++)
10     {
11       printf(" %d\n",i);
12     }
13     return 0;
14 }
```

```
./contador
0
1
2
3
4
5
6
7
8
9
```

Bucles

Sentencia `for`:

Variaciones:

```
1  //bucle infinito
2  for( ; ; )
```

```
1  //Como while(expresion)
2  for ( ; expresion; )
```

```
1  //compilar con gcc -std=c99
2  for(int i=0; i < n; i++)
```

```
1  void strReverse( char* str)
2  {
3    char ch;
4    for ( int i = 0, j = strlen(str)-1;  i < j;  ++i, --j )
5      ch = str[i],  str[i] = str[j],  str[j] = ch;
6  }
```

Bucles

Sentencia `for`:

Variaciones:

```
1   /* for_var.c */
2   #include<stdio.h>
3
4   int main()
5   {
6       char *letras[] = {"a","b","c",NULL};
7       char **p;
8
9       p = letras;
10      for(; *p; p++)
11      {
12        printf(" %s\n",*p);
13      }
14  }
```

Bucles

Sentencia `continue`:

- Sólo puede utilizarse dentro del cuerpo de un bucle.
- Detiene la ejecución de la iteración en curso y salta a la siguiente.

Ejemplo:

```
1  /* continue.c */
2  #include<stdio.h>
3
4  void main()
5  {
6       /* Cuenta de 0 a n-1 */
7      int sum, i;
8      int n = 10;
9
10     sum = 0;
11     for(i=0; i < n; i++)
12     {
13         if(i==7)
14         {
15           printf("Saltando el 7\n");
16           continue;
17         }
18         sum += i;
19     }
20     printf("Suma total (excepto el 7): %d\n",sum);
21 }
```

Bucles

Sentencia `continue`:

Ejemplo (II):

```
1  /* continue2.c */
2  #include <stdio.h>
3  char  line[100];   // linea por teclado
4  int   total; // Suma total de elementos
5  int   item;        // Siguiente elemento
6  int   minus_items; // Valores negativos
7
8  int main()
9  {
10   total = 0;
11   minus_items = 0;
12
13   while (1) {
14
15     printf("Introduzca numero para
           sumar valor ");
16     printf("o 0 para terminar: ");
17
18     fgets(line, sizeof(line), stdin);
19     sscanf(line, " %d", &item);
20
21     if (item == 0) break;
23     if (item < 0)
24     {
25          ++minus_items;
26          continue;
27     }
28
29          total += item;
30          printf("Total:  %d\n", total);
31   }
32
33     printf("Total final  %d\n", total);
34     printf("con  %d valores negativos
            omitidos\n", minus_items);
35
36     return (0);
37 }
```

Laboratorio 3

(Tomada de http://tux.crystalxp.net/)

Índice

1 Introducción
2 Tipos, declaraciones y expresiones
3 Control de flujo
4 Funciones
5 Entrada/salida
6 Arrays y Punteros
7 Tipos avanzados
8 Librerías
9 El preprocesador de C

Funciones

- Todas las instrucciones de un programa en C están contenidas en **funciones**.
- Cada función realiza una **tarea determinada**.
- La función `main()` es la que primero se ejecuta en un programa.
- Es posible enviar datos denominados **parámetros** a una función al invocarla.
- Permiten estructurar un programa de forma **modular**, esto es, en partes más pequeñas de finalidad muy concreta.
  - Facilitan el desarrollo y mantenimiento de un programa.
  - Evitan errores.
  - Facilitan la reutilización de código.
- Cada función es definida una sóla vez y puede ser invocada tantas veces como sea necesario.

Definición de una función

```
1  tipo_valor_retorno nombre_funcion(lista argumentos con tipos)
2  {
3    declaración de variables
4    otras sentencias
5    return(expresión);
6  }
```

- *Línea 1*: Cabecera o ***header***.

  `tipo_valor_retorno`: Tipo del valor que devuelve la función al término de su ejecución.

  - Puede ser `void` o cualquier tipo, excepto arrays. No pueden devolver arrays u otra función. Sin embargo, pueden devolver punteros a una función o un array.
  - Puede incluir el especificador de función `inline` y/o uno de los especificadores de almacenamiento `extern` o `static`.
  - Pueden aparecer varios `return` en una misma función. Devuelve el control al programa que llama y devuelve el valor de retorno.

  La **`lista de argumentos`** se especifica separándolos con comas. Si no son necesarios, la lista se puede dejar vacía "( )" o utilizar la palabra `void`.

- *Líneas 3 a 5*: ***Cuerpo*** de la función. Cada función puede disponer de sus propias variables.

Declaración de una función

Toda función debe ser declarada antes de usarse. Modos:

1. Invocando a la función en el programa, si no ha sido previamente *definida* o *declarada*. Se considera *int* como valor de retorno implícito. Este modo está desaconsejado.
2. Definiendo la función antes de que sea invocada en el programa. Desventaja: si la función se cambia de lugar, pasamos a tener el caso anterior.
3. Haciendo declaración explícita, previa a la invocación, mediante el *prototipo*:

```
1   #include <stdio.h>
2
3   int gestionNomina(int, char *, double );  // prototipo
4
5   int main(int argc,char **argv)
6   {
7      /* .... */
8   }
9
10  int gestionNomina(int idEmpleado, char *dpto, double sueldoBase )
11  {
12    /* .... */
13  }
```

Invocación a una función

Ejemplo: Cálculo del área de un triángulo

```
1   /* triangulo.c */
2   #include <stdio.h>
3
4   float triangulo(float , float );
5
6   int main()
7   {
8       printf("Triangulo #1  %f\n", triangulo(1.3, 8.3));
9       printf("Triangulo #2  %f\n", triangulo(4.8, 9.8));
10      printf("Triangulo #3  %f\n", triangulo(1.2, 2.0));
11      return (0);
12  }
13
14
15  float triangulo(float base, float altura)
16  {
17      float area;
18
19      area = base * altura / 2.0;
20      return (area);
21  }
```

```
./triangulo
Triangulo #1 5.395000
Triangulo #2 23.520000
Triangulo #3 1.200000
```

Invocación a una función

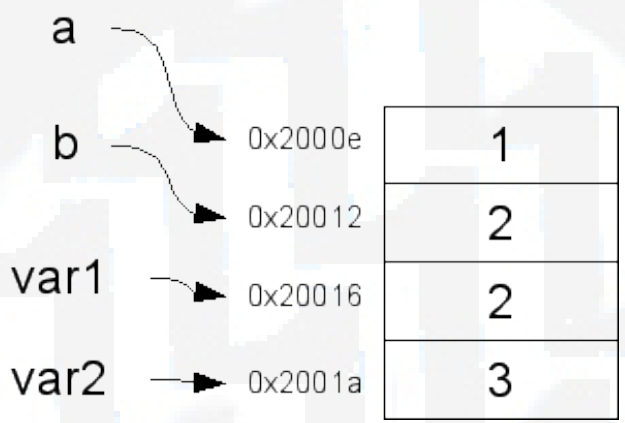
- La llamada se hace incluyendo el nombre de la función en una expresión o sentencia del programa principal u otra función.
- Los argumentos van entre paréntesis, separados por comas.
- El número de argumentos debe coincidir con el número de argumentos de la definición de la función.
- Si el tipo de los argumentos incluidos en la llamada (*actuales*) no coincide con el de los argumentos de la definición (*formales*), se tratará de convertirlos.
- La función se ejecutará hasta llegar a un `return` o al final del cuerpo de la función. Se devuelve el control al programa que realizó la llamada, junto con el valor de retorno, si existe.

Argumentos y valor de retorno

Paso por valor: Por defecto, cuando se pasan argumentos a una función, se realiza *una copia* de éstos dentro de la función. Los cambios que se hagan en la función no se transmiten a las variables del programa que la ha llamado.

```
1  /* pasoxvalor.c */
2  #include <stdio.h>
3
4  void incrementa(int , int );
5
6  int main()
7  {
8    int a = 1, b = 2;
9
10   printf("Valor de a: %d\n",a);
11   printf("Valor de b: %d\n",b);
12
13   incrementa(a,b);
14
15   printf("Valor de a: %d\n",a);
16   printf("Valor de b: %d\n",b);
17
18   return (0);
19 }
20
21 void incrementa(int var1, int var2)
22 {
23   printf("\nIncrementando...\n");
24   printf("var1: %d\n",++var1);
25   printf("var2: %d\n\n",++var2);
26 }
```

```
./pasoxvalor
Valor de a: 1
Valor de b: 2

Incrementando...
var1: 2
var2: 3

Valor de a: 1
Valor de b: 2
```

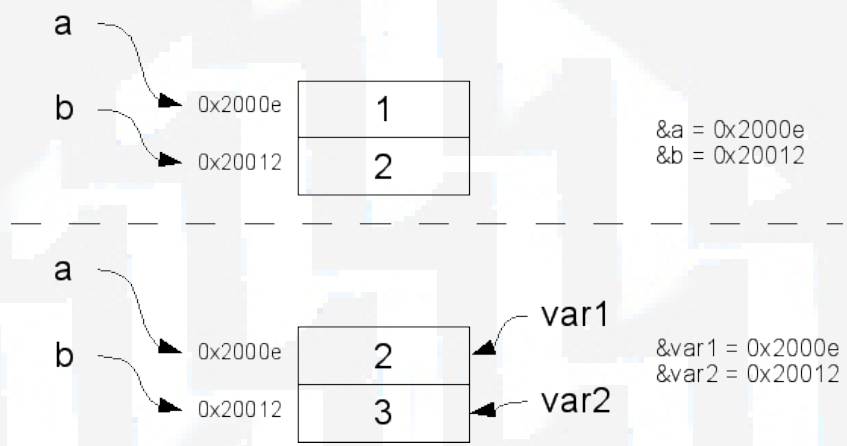


Argumentos y valor de retorno

Paso por referencia: Si es necesario transmitir los cambios hechos a las variables del programa que llamó a la función, el paso de parámetros debe realizarse indicando la dirección en memoria de la variable (con &).

```
1  /* pasoxreferencia.c */
2  #include <stdio.h>
3
4  void incrementa(int *, int *);
5
6  int main()
7  {
8    int a = 1, b = 2;
9
10   printf("Valor de a: %d\n",a);
11   printf("Valor de b: %d\n",b);
12
13   incrementa(&a,&b);
14
15   printf("Valor de a: %d\n",a);
16   printf("Valor de b: %d\n",b);
17
18   return (0);
19 }
20
21 void incrementa(int *var1, int *var2)
22 {
23   printf("\nIncrementando...\n");
24   printf("var1: %d\n",++(*var1));
25   printf("var2: %d\n\n",++(*var2));
26 }
```

```
./pasoxreferencia
Valor de a: 1
Valor de b: 2

Incrementando...
var1: 2
var2: 3

Valor de a: 2
Valor de b: 3
```



Ámbito y visibilidad de las variables

Ámbito de una variable:

- Una **variable global** existe durante toda la ejecución del programa y es visible por todas las funciones de éste.
- Una **variable local** es válida únicamente dentro de un bloque ({..}) y sus bloques anidados. La variable es creada al comenzar a ejecutarse el bloque y deja de existir al finalizar éste. No son inicializadas por defecto.

```c
/* ambito.c */
#include <stdio.h>

int global;

int main()
{
   int local = 2; // local a main
   global = 1;

   if(1)  //Nuevo bloque
   {
     //local al bloque if
     int muy_local;
     muy_local = local + global;
   }
   //muy_local ya no puede
   //ser usada aqui
   return 0;
}
```

```c

int funcion()
{
   //local a la funcion
   int local_funcion;
   /* ... */
}
```

Ámbito y visibilidad de las variables

Ámbito de una variable:

- Una variable puede ser ***ocultada*** por la declaración de una nueva variable con el mismo nombre en un bloque anidado.

```
1  /* ocultacion.c */
2  #include <stdio.h>
3
4  int main()
5  {
6    int total = 0;
7    int cont = 0;
8
9    {
10     int cont; //Contador local.
11               //Oculta el cont anterior
12
13     cont = 0;
14     while(1)
15     {
16       if(cont > 10)
17         break;
18       total += cont;
19       ++cont;
20     }
21     printf("cont en el bloque:  %d\n",cont);
22
23   }
24    printf("cont fuera del bloque:  %d\n",cont);
25 }
```

```
./ocultacion
cont en el bloque: 11
cont fuera del bloque: 0
```

Ámbito y visibilidad de las variables

Modos de almacenamiento en variables y funciones

extern:

- Indica que una variable o función está definida fuera del fichero actual.

extern.c

```
1  #include <stdio.h>
2
3  extern int counter;
4  extern void inc_counter(void);
5
6  main()
7  {
8      int index;
9
10     for (index = 0; index < 10; index++)
11         inc_counter();
12
13     printf("Contador es  %d\n", counter);
14
15     return 0;
16 }
```

count.c

```
1  int counter = 0;
2
3  void inc_counter(void)
4  {
5      ++counter;
6  }
```

- Por defecto, todas las funciones tienen modo `extern` (la línea 4 no es necesaria).

Ámbito y visibilidad de las variables

Modos de almacenamiento en variables y funciones

static:

- Una **variable** declarada como `static` **dentro de un bloque** conserva su valor entre ejecuciones de dicho bloque (se comporta como una global). La inicialización se realiza únicamente la primera vez. Por defecto, se inicializan a 0.

```
1  /* static.c */
2  int main()
3  {
4    int j;
5
6    for(j=0;j<5;j++)
7      funcionNoStatic();
8
9    for(j=0;j<5;j++)
10     funcionStatic();
11
12   return 0;
13 }
14
15 void funcionStatic()
16 {
17   static int contador = 1;
18
19   contador ++;
20   printf("cont. static: %d\n",
           contador);
21 }
```

```
23 void funcionNoStatic()
24 {
25   int contador = 1;
26
27   contador ++;
28   printf("cont. no static: %d\n",
           contador);
29 }
```

```
./static
cont. no static: 2
cont. no static: 2
cont. no static: 2
cont. no static: 2
cont. no static: 2
cont. static: 2
cont. static: 3
cont. static: 4
cont. static: 5
cont. static: 6
```

Ámbito y visibilidad de las variables

Modos de almacenamiento en variables y funciones

static:

- Una **variable global** definida como `static` sólo será visible para ese archivo.
- Una **función** definida como `static` sólo será visible para las funciones definidas después de dicha función y en el mismo fichero.

Con estos modos se puede controlar la visibilidad de una función, es decir, desde qué otras funciones puede ser llamada.

register:

- *Recomendación* al compilador de que almacene la variable en un registro del procesador (más rápido, más eficiente).

La función main()

Con excepción de los sistemas embebidos, todo programa en C debe contener una función de nombre `main`, que será la primera invocada cuando se inicie el programa. Dos posibles formatos:

```
int main( void ) { /* ... */ }
```

```
int main( int argc, char *argv[ ] ) { /* ... */ }
```

- Devuelve al SO un `int` con el estado final del programa: 0 (EXIT_SUCCESS) o cualquier otro valor (p.ej. EXIT_FAILURE) cuando se produjo un error (constantes definidas en *stdlib.h*). No es necesario especificar una sentencia `return`, por defecto se devuelve 0.
- Terminar la función `main` equivale a llamar a la función **`exit()`**, cuyo argumento sería el valor de retorno de `main`.
- **`argc`** (*argument count*): Número de argumentos introducidos por línea de comandos al invocar el programa. El propio nombre del programa se cuenta.
- **`argv`** (*argument vector*): array de punteros a `char` que apunta a cada uno de los strings de los argumentos.
  - El número de elementos en el array es `argc` (se indexan de 0 a *argc-1*). `argv[0]` es el nombre del programa.
  - `argv[1]` a `argv[argc-1]` contienen los argumentos introducidos por línea de comandos.

La función main()

Ejemplo:

```
1  /* main.c */
2  #include <stdio.h>
3
4  int main( int argc, char **argv )
5  {
6    int i;
7
8    printf( "Programa en ejecucion: %s\n\n", argv[0] );
9    if ( argc == 1 )
10       puts( "No hay argumentos." );
11   else
12    {
13       puts( "Argumentos introducidos:" );
14
15       for (i = 1; i < argc; ++i )
16         puts( argv[i] );
17    }
18 }
```

```
./main uno dos
Programa en ejecucion: ./main

Argumentos introducidos:
uno
dos
```

La función main()

Ejemplo: Gestión eficiente de opciones y argumentos con `getopt()`

```
1  #include <unistd.h> /* ...*/ /* getopt.c
       */
2  int main (int argc, char **argv)
3  {
4   int aflag = 0, bflag = 0;
5   char *cvalue = NULL;
6   int index, c;
7   opterr = 0;
8
9   while ((c=getopt(argc,argv,"abc:"))!=-1)
10   switch (c){
11    case 'a':
12     aflag = 1;
13     break;
14    case 'b':
15     bflag = 1;
16     break;
17    case 'c':
18     cvalue = optarg;
19     break;
20    case '?':
21     if (optopt == 'c')
22      fprintf(stderr,"Option -%c requires
            an argument.\n",optopt);
23     else if (isprint (optopt))
24      fprintf(stderr,"Unknown option `-%c
            '.\n",optopt);
```

```
26      else
27       fprintf (stderr,"Unknown option
             character `\\x%x'.\n",optopt);
28      return 1;
29     default:
30      abort ();
31   }
32
33   printf ("aflag=%d,bflag=%d,cvalue=%s\n"
       ,aflag,bflag,cvalue);
34
35   for (index=optind;index<argc;index++)
36     printf ("Non-option argument %s\n",
         argv[index]);
37   return 0;
38   }
```

La función main()

Ejemplo: Gestión eficiente de opciones y argumentos con `getopt()`

```
getopt(int argc, char **argv, const char *options)
```

- `getopt` obtiene cada uno de los argumentos de la lista especificada por *argv*
- El argumento `options` es un string que especifica los caracteres de opción válidos para el programa. Si al carácter le siguen dos puntos (':') quiere decir que requiere un argumento.
- Cada invocación a `getopt` devuelve el siguiente carácter de opción. Cuando se han devuelto todos, la función devuelve -1. La llamada al programa puede contener más argumentos que no son opciones. Se comprueba utilizando la variable externa `optind`.
- **`char *optarg`:** cuando una opción requiere un argumento, `getopt` lo almacena en la dirección apuntada por `*optarg`.
- **`int optind`:** apunta al índice del siguiente elemento a procesar en *argv*. Una vez que `getopt` ha devuelto todos los argumentos de opción, se puede usar esta variable para determinar dónde comienzan los argumentos no opcionales. El valor inicial de `optind` es 1.
- **`int optopt`:** Cuando `getopt` encuentra un carácter de opción desconocido, o un carácter de opción sin un argumento requerido, almacena ese carácter en esta variable.
- **`int opterr`:** Cuando esta variable es distinta de cero, `getopt` imprime un mensaje de error si encuentra una opción no válida o sin un argumento requerido. Si es cero, no se imprime ningún mensaje.

Si `getopt` encuentra un carácter de opción en *argv* que no estaba incluido en el argumento `options`, o si falta un argumento de opción, devuelve '?' y almacena dicho carácter en `optopt`. Si el primer carácter en `options` son dos puntos (':'), `getopt` devuelve ':' en vez de '?' para indicar que falta un argumento.

Funciones inline

Llamar a una función puede ser muy costoso:

1. Se salva la dirección de la instrucción actual.
2. Se salta a la dirección de la función y se ejecuta.
3. Se recupera la dirección guardada y se vuelve a saltar.

Funciones pequeñas que son invocadas frecuentemente degradan el rendimiento del programa.

Solución: Funciones *inline*: Se solicita al compilador que incluya el código máquina de la función en cada zona del programa en la que se llame a la función:

```
1       // Termina el programa con un mensaje de error:
2       inline void error_exit( int status, const char *error_msg )
3       {
4         fputs( error_msg, stderr );
5         exit( status );
6       }
```

```
1       inline int iMax(int a, int b) { return a >= b ? a : b; }
```

Funciones con número variable de argumentos

- Requieren un número fijo de parámetros obligatorios (al menos, uno) seguidos por un número variable de parámetros optativos.
- Los tipos de los argumentos optativos pueden variar.
- El número de argumentos optativos se determina a través de uno de los argumentos obligatorios o a través de un valor especial que termina la lista de argumentos optativos.
- Para acceder a los argumentos optativos se utilizan tres macros, definidas en `stdarg.h`:
  - **`void va_start( va_list argptr, lastparam )`**: Inicializa el puntero `argptr` con la posición del primer argumento optativo. `lastparam` debe contener el nombre del último argumento obligatorio de la lista.
  - **`type va_arg( va_list argptr, type )`**: Devuelve el argumento optativo apuntado por `argptr` y avanza `argptr` para apuntar al siguiente elemento de la lista. El argumento devuelto tendrá el tipo `type`.
  - **`void va_end( va_list argptr )`**: Debe llamarse una vez que se ha terminado de utilizar `argptr`.

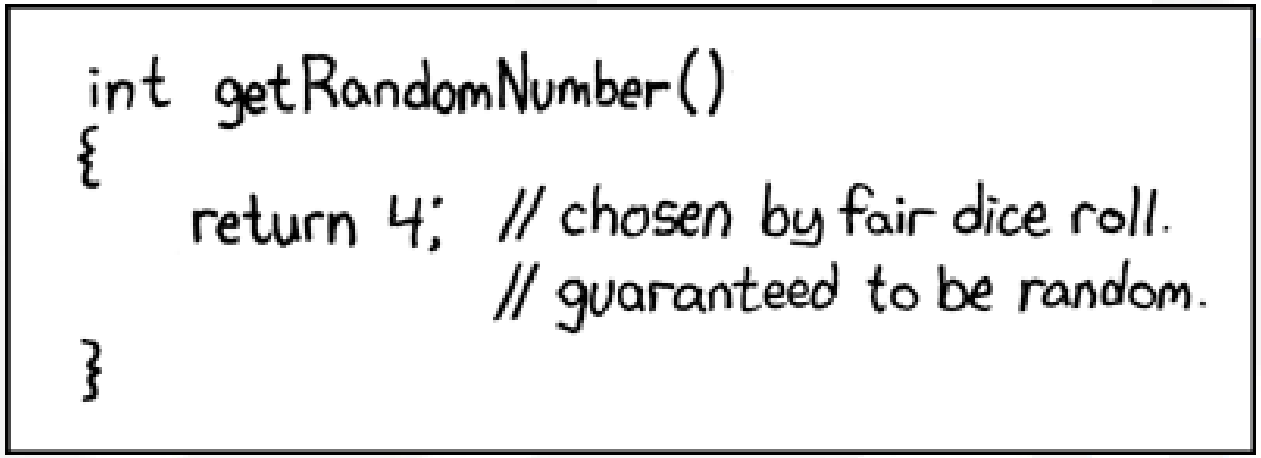
Funciones con número variable de argumentos

Ejemplo:

```c
/* varpar.c */
#include <stdio.h>
#include <stdarg.h>
#include <stdlib.h>
// Calcula la suma de
// los argumentos optativos
// Argumentos: El obligatorio
//   indica el numero de argumentos
//   opcionales. Los opcionales son
//   de tipo doble.
double add( int n, ... )
{
  int i = 0;
  double sum = 0.0;

  va_list argptr;
  va_start( argptr, n );

  for ( i = 0; i < n; ++i )
  {
    sum += va_arg( argptr, double );
  }

  va_end( argptr );
  return sum;
}
```

```c
// Muestra un mensaje de error y sale
 static void
 fatal_error(char *fmt, ...)
 {
     va_list ap;

     va_start(ap, fmt);
     vfprintf(stderr, fmt, ap);
     va_end(ap);

     exit(1);
 }


int main()
{
  double res = add(2,5.5,10.5);
  printf("res:  %f\n",res);

  fatal_error("Parametros  %d y \"%s\"
         no validos\n",3,"cadena");

}
```

Laboratorio 4



(Tomada de http://xkcd.com/)

Índice

1 Introducción
2 Tipos, declaraciones y expresiones
3 Control de flujo
4 Funciones
5 Entrada/salida
6 Arrays y Punteros
7 Tipos avanzados
8 Librerías
9 El preprocesador de C

Lectura y escritura de ficheros

- Los ficheros son estructuras de datos almacenadas en memoria secundaria (discos duros, etc)
- Mientras que en memoria principal basta con hacer una asignación a una variable para almacenar ese dato, en memoria secundaria es necesario realizar una serie de operaciones previas.
- Para manejar un fichero se debe definir primero un puntero a archivo (tipo `FILE`, incluido en `stdio.h`) :

```
FILE *fd;  //fd: file descriptor
```

- El procedimiento de utilización de ficheros tiene la siguiente secuencia: *apertura del fichero*, *lectura y/o escritura del fichero* y *cierre del fichero*. El puntero irá avanzando a medida que escribamos o leamos.
- Durante la **apertura** del archivo se decide el tipo de datos que va a contener: texto (almacena caracteres) o binario (almacena cualquier tipo de dato. Por ejemplo, un `struct`):

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

  - `path` es una cadena de caracteres indicando el nombre del fichero (y, si es necesario, la ruta completa).
  - `mode` es una cadena de caracteres indicando el modo de apertura.
  - Si tiene éxito, devuelve un puntero al fichero abierto. En caso contrario, devuelve NULL.

Lectura y escritura de ficheros

- Los modos de apertura disponibles son:
  - **r**: Sólo lectura. Si no existe devuelve un error.
  - **w**: Sólo escritura. Si no existe el fichero, se crea. Si ya existe, se destruye y se crea uno nuevo.
  - **a**: Abre para añadir datos al final (*append*).
  - **r+**: Permite abrir para lectura y escritura. Si no existe, devuelve un error.
  - **w+**: Permite abrir para escritura y lectura. Si no existe, se crea.
  - **rb**: Abre para lectura en modo binario (por defecto, abre en modo texto).
  - **wb**: Abre para escritura en modo binario (por defecto, abre en modo texto).
  - **ab**: Abre para añadir datos al final en modo binario.
- Ejemplo:

```
1  #include <stdio.h>  /* ficheros.c */
2  #include <stdlib.h>
3
4  int main()
5  {
6    FILE *fd;
7    char *fichName = "contabilidad.txt";
8
9    if((fd=fopen(fichName,"r")) == NULL)
10   {
11      printf("Error: no se pudo abrir el fichero %s\n",fichName);
12      exit(1);
13   }
14   /* ... */
15 }
```

Lectura y escritura de ficheros

Lectura y escritura formateada de texto (fscanf-fprintf):

- `fprintf` se comporta igual que `printf`, indicándole el descriptor del fichero en el que escribir:

```
fprintf(fd,"%d\t %.2f\t %s\n",item,precio,articulo);
```

- `fscanf` lee texto formateado del fichero indicado por el descriptor. El formato es similar al de `fprintf`, pero los parámetros se deben pasar por referencia para que la función les asigne los valore leidos:

```
fscanf(fd,"%d\t %f\t %s\n",&itemLeido,&precioLeido,articuloLeido);
```

Lectura y escritura de ficheros

Lectura y escritura de caracteres (fgetc-fputc) y cadenas (fgets-fputs):

- `fputc` escribe el carácter `c` en el fichero apuntado por `fd`:

```
1  char c = 'a';
2  fputc(c,fd);
```

- `fgetc` lee un carácter del fichero apuntado por `fd` y lo almacena en `c`. `fgetc` devuelve el carácter leído o EOF (*End Of File*) si ha llegado al final de fichero:

```
1  char c;
2  c = fgetc(fd);
```

- `fputs` escribe una cadena de texto en el fichero apuntado por `fd`:

```
1  char *texto = "En un lugar de la Mancha...";
2  fputs(texto,fd);
```

- `fgets` lee una cadena de texto de tamaño `tam-1` caracteres desde el fichero apuntado por `fd` y lo almacena en `textoLeido`, colocando el carácter de fin de cadena ('\0') al final de ésta. Se detiene tras llegar a fin de línea o a EOF:

```
1  char textoLeido[256];
2  int tam = 10;
3  fgets(textoLeido,tam,fd);
```

Lectura y escritura de ficheros

Lectura y escritura binaria (fread-fwrite):

- `fwrite` obtiene un bloque de datos de la dirección apuntada por `dir_dato`, y escribe `tamaño_dato * numero_datos` bytes en el fichero apuntado por `fd`. La función devuelve el número de datos escritos que, si es correcto, debe coincidir con `numero_datos`:

```
size_t fwrite (dir_dato, tamaño_dato, numero_datos, fd);
```

- `fread` escribe un bloque de datos en la dirección apuntada por `dir_dato`, tras leer `tamaño_dato * numero_datos` bytes en el fichero apuntado por `fd`. La función devuelve el número de datos leídos que, si es correcto, debe coincidir con `numero_datos`:

```
size_t fread(dir_dato, tamaño_dato, numero_datos, fd);
```

Lectura y escritura de ficheros

Lectura y escritura binaria (fread-fwrite):

- Ejemplo de `fwrite`:

```
1  #include <stdio.h>  /* fwrite.c */
2  #include <stdlib.h>
3
4  int main()
5  {
6    FILE *fd;
7    char *fichName = "vectores.dat";
8    float v[] = {1.5, 3.4, 5.5, 4.3, 9.4};
9    int escritos, entero = 5;
10
11   if((fd=fopen(fichName,"wb")) == NULL)
12   {
13      printf("Error: no se pudo abrir el fichero %s\n",fichName);
14      exit(1);
15   }
16   //Escribimos todo el vector v
17   escritos = fwrite(v,sizeof(float),5,fd);
18
19   //Escribimos 3 elementos de v desde v[2]
20   escritos += fwrite(&v[2],sizeof(float),3,fd);
21
22   //Escribimos un entero
23   escritos += fwrite(&entero,sizeof(int),1,fd);
24
25   printf("Total elementos escritos: %d\n",escritos);
26   fclose(fd);
27 }
```

Lectura y escritura de ficheros

Lectura y escritura binaria (fread-fwrite):

- Ejemplo de `fread`:

```
1  #include <stdio.h>  /* fread.c */
2  #include <stdlib.h>
3
4  int main()
5  {
6    FILE *fd;
7    char *fichName = "vectores.dat";
8    float v[5], v2[3];
9    int leidos, entero, i;
10
11   if((fd=fopen(fichName,"rb")) == NULL) {
12      printf("Error: no se pudo abrir el fichero %s\n",fichName);
13      exit(1);
14   }
15   //Leemos todo el vector v
16   leidos = fread(v,sizeof(float),5,fd);
17   for(i=0;i<5;i++) printf("%.1f  ",v[i]); putchar('\n');
18   //Leemos 3 elementos y los metemos en v2
19   leidos += fread(v2,sizeof(float),3,fd);
20   for(i=0;i<3;i++) printf("%.1f  ",v2[i]); putchar('\n');
21   //Leemos un entero
22   leidos += fread(&entero,sizeof(int),1,fd);
23   printf("Entero: %d\n",entero);
24
25   printf("Total elementos leidos: %d\n",leidos);
26   fclose(fd);
27 }
```

Lectura y escritura de ficheros

Cierre de un fichero:

- Si `fd` es un puntero a un fichero abierto, podemos cerrarlo con `fclose` :

```
fclose(fd);
```

Recorrido secuencial de un fichero (`feof`):

- Cada vez que se realiza una lectura o escritura, el puntero al archivo se incrementa, apuntando a la siguiente posición donde leer o escribir.
- Para leer todos los datos de un fichero basta con ir recorriéndolo hasta llegar al final. Aunque sería posible leer el dato leído y comprobar si es un EOF, resulta más sencillo utilizar la función `feof`:

```
feof(fd);
```

- `feof` devuelve 0 mientras no se alcance el final de fichero.

Lectura y escritura de ficheros

Recorrido secuencial de un fichero (`feof`):

- Ejemplo de `feof`:

```
1  /** copiaFich.c: Copia un fichero de texto **/
2  #include <stdio.h>
3  #include <stdlib.h>
4
5  int main()
6  {
7    FILE *fdr, *fdw;
8    char *original = "texto.txt";
9    char *copia = "texto_copia.txt";
10   char c;
11   int i = 0;
12
13   if(((fdr=fopen(original,"r")) == NULL) ||((fdw=fopen(copia,"w")) == NULL))
14   {
15      printf("Error: no se pudo abrir uno de los ficheros\n");
16      exit(1);
17   }
18
19   c = getc(fdr);
20   do{
21     putc(c,fdw);
22     c = getc(fdr);
23   }while(!feof(fdr));
24
25   fclose(fdr);
26   fclose(fdw);
27 }
```

Lectura y escritura de ficheros

Acceso directo a los datos `fseek`:

- Normalmente accedemos a los datos de un fichero *secuencialmente*, a medida que vamos leyéndolos. `fseek` nos permite acceder a una posición determinada en el fichero:

```
fseek(fd, desplazamiento, origen)
```

- coloca el puntero `fd` a tantos bytes del `origen` como indica `desplazamiento`.

  `origen` puede ser:
  - SEEK_SET (0): principio del fichero
  - SEEK_CUR (1): posición actual del puntero
  - SEEK_END (2): final del fichero
- `fseek` devuelve 0 si no ha habido ningún error.
- Ejemplo:

```
fseek(fd,3*sizeof(int),SEEK_CUR)
```

  colocará el puntero tres posiciones (suponiendo que estamos leyendo enteros) más allá de la posición actual.
- Para saber la posición actual en bytes del puntero, contando desde el inicio del fichero, podemos usar la función:

```
ftell(fd)
```

Lectura y escritura por entrada/salida estándar

- Las funciones para lectura y escritura por entrada/salida estándar (típicamente teclado y pantalla) son iguales o equivalentes a las utilizadas en ficheros para mostrar escritura formateada de texto, caracteres y cadenas:

`fprintf`	$\Longrightarrow$	`printf`
`fscanf`	$\Longrightarrow$	`scanf`
`fgetc`	$\Longrightarrow$	macro `getchar` (equivale a `getc(stdin)`)
`fputc`	$\Longrightarrow$	macro `putchar` (equivale a `putc(c,stdout)`)
`fgets`	$\Longrightarrow$	macro `gets`
`fputs`	$\Longrightarrow$	macro `puts`

- Existen otras funciones para realizar conversiones de entrada y salida formateada (`sprintf`, `sscanf`). Para más información, consultar la página del manual.

Laboratorio 5

(Tomada de http://tux.crystalxp.net/)

Índice

1 Introducción
2 Tipos, declaraciones y expresiones
3 Control de flujo
4 Funciones
5 Entrada/salida
6 Arrays y Punteros
7 Tipos avanzados
8 Librerías
9 El preprocesador de C

Arrays

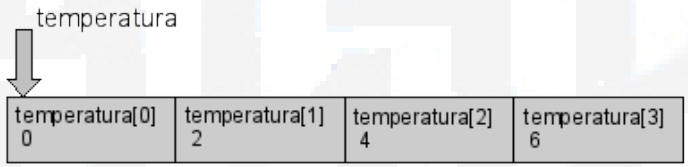
- Un **array** contiene objetos de un mismo tipo, almacenados consecutivamente en un bloque de memoria contiguo
- Cada uno de los objetos almacenados se denomina `elemento`.
- El número de elementos en un array se denomina *dimensión* del array.

Definición de un array:

```
double buffer[4*512]; // array de doubles de dimensión 2048.
                      // Dimensión fija.
char nombre[2*n]; // array de char de dimensión variable 2*n
```

- Se reserva memoria para 2048 variables de tipo *double* y 2*n *char* (siendo n conocido).
- Los elementos se numeran de 0 a *dimensión - 1* (2043).
- **Acceso**, para inicialización y lectura, mediante un subíndice:

```
1  /* arrays.c */
2  #include <stdio.h>
3  #define SIZE 4
4
5  int main()
6  {
7   int i, temperatura[SIZE];
8
9   for(i=0; i< SIZE; i++)
10    temperatura[i] = 2*i;
11 }
```



Inicialización y acceso

- En un array sin inicializar, los valores de sus elementos están indefinidos.
- Definición e inicialización simultáneas:

```
int a[4] = { 1, 2, 4, 8 };
int a[] = { 1, 2, 4, 8 };
```

- Inicialización elemento a elemento:

```
a[0] = 1,  a[1] = 2,  a[2] = 4,  a[3] = 8;
```

- Los elementos van siempre del índice 0 a *dimension - 1*. Lo siguiente es ilegal:

```
1 int a[4];
2 a[4] = 1  //Ilegal!!
```

(El compilador no se queja, pero estamos accediendo a posiciones de memoria fuera del array y se puede provocar un *fallo de segmentación*).

- Si el array es de dimensión variable, no puede ser `static`:

```
1 int i = 4;
2 static int a[i]; // Error de compilación
```

- No se puede incluir una inicialización en la definición de un array de tamaño variable:

```
1 int j=5;
2 int b[j] = {1,2,3,4,5}; //Error de compilación
```

Inicialización y acceso

- Si la definición de un array indica la dimensión del array (entre [ ]) y contiene una lista de inicialización, entonces la longitud del array es la especificada entre corchetes.
- Aquellos elementos para los que no haya inicialización en la lista, se inicializan a 0 (NULL para punteros).
- Si la lista inicializa más elementos de los indicados por la dimensión del array, se ignoran.
- Una coma tras el último elemento inicializado se ignora.
- Las siguientes definiciones son equivalentes:

```
int a[4] = {1,2,0,0};
int a[4] = {1,2};
int a[ ] = {1,2,0,0};
int a[ ] = {1,2,0,0,};
int a[4] = {1,2,0,0,5}; //5 se ignora. Genera warning
```

- Otro ejemplo:

```
struct Persona{
   unsigned long id;
   char name[64];
};

struct Persona equipo[6] = { { 1000, "Maria"}, { 2000, "Francisco"} };
```

Los otros cuatro elementos del array `equipo` son inicializados a 0 o, en este caso, a { 0, ” “ }.

Strings

- Un **string** o **cadena de caracteres** es un array de tipo `char` cuyo último elemento es el carácter nulo ('\0').
- Su longitud es el número de caracteres que contiene excluyendo el carácter nulo.

```
char nombre[20] = "Juan";  //Longitud del array: 20
                           //Longitud del string: 4

char nombre[] = "Juan";    //Longitud del array: 5
                           //Longitud del string: 4
```

- Son equivalentes:

```
char nombre[20] = "Juan";
char nombre[] = "Juan";
char nombre[] = {'J','u','a','n','\0'};
```

Operaciones con strings

- No se puede declarar un string directamente si no es durante su inicialización:

```
char apellido[20];
apellido = "García"; // Error de compilación
```

- Para trabajar con strings se utilizan las funciones incluidas en `<string.h>`:

Función	Significado
`strcpy(string_dest, string_orig)`	Copia `string_orig` en `string_dest`. `string_dest` debe ser suficientemente grande como para alojar la copia
`strcat(string_dest, string_orig)`	Concatena `string_orig` al final de `string_orig`. `string_dest` debe ser suficientemente grande como para alojar el resultado
`length = strlen(string)`	Devuelve la longitud del string
`strcmp(string1, string2)`	Devuelve un entero menor, igual o mayor que cero si `string1` es, respectivamente, menor que, igual a, o mayor que `string2`

Operaciones con strings

- **Ejemplo:**

```
1  #include <stdio.h>  /* strings_ej.c */
2  #include <string.h>
3  #define MAX_LEN 20
4
5  int main()
6  {
7   char nombre[MAX_LEN] = "Juan";
8   char apellido[MAX_LEN] = "Garcia";
9   char apellido2[MAX_LEN] = "Lopez";
10  char nombreCompleto[MAX_LEN];
11
12  strcpy(nombreCompleto,nombre);
13  printf("nombre completo: %s\nLongitud: %d\nTamano:
        %d\n\n",nombreCompleto,strlen(
      nombreCompleto),sizeof(nombreCompleto));
14
15  strcat(nombreCompleto," ");
16  strcat(nombreCompleto,apellido);
17  printf("nombre completo: %s\nLongitud: %d\nTamano:
        %d\n",nombreCompleto,strlen(nombreCompleto)
      ,sizeof(nombreCompleto));
18
19  if(strcmp(apellido,apellido2) > 0)
20        printf("Listado: 1. %s, 2. %s\n",apellido2,
            apellido);
21  else
22        printf("Listado: 1. %s, 2. %s\n",apellido,
            apellido2);
23 }
```

```
./strings_ej
nombre completo: Juan
Longitud: 4
Tamano: 20

nombre completo: Juan Garcia
Longitud: 11
Tamano: 20
Listado: 1. Garcia, 2. Lopez
```

Arrays multidimensionales

- Un *array multidimensional* no es más que un array cuyos elementos son también arrays.
- Se declaran de forma análoga a los arrays unidimensionales:

```
double posicion[10][20][30];  //Array de 3 dimensiones
```

- Se accede a cada elemento a través de sus índices:

```
double posicion[9][19][29] = 3.54; //Acceso al último elemento
```

- `posicion` es un array de dimensión 10, cuyos elementos son arrays de dimensión 20, los elementos de cada uno de los cuales son arrays de dimensión 30.

Arrays multidimensionales

- Una matriz es un array de 2 dimensiones. Ejemplo:

```
1  /* matriz.c */
2  #include<stdio.h>
3
4  int main()
5  {
6    int i,j;
7    int rows = 3, cols = 4;
8    float m[rows][cols];
9
10   for(i=0;i<rows;i++)
11   {
12     for(j=0;j<cols;j++)
13     {
14         m[i][j] = (float)(i+j)/3;
15         printf("Valor de m[%d][%d]:  %.2f\n",i,j,m[i][j]);
16     }
17   }
18 }
```

- En C, las filas se almacenan consecutivamente en memoria:

Columna	0	1	2	3
m[0]	0.00	0.33	0.67	1.00
m[1]	0.33	0.67	1.00	1.33
m[2]	0.67	1.00	1.33	1.67

Arrays como argumentos de una función

- Un array se pasa siempre a una función como parámetro por referencia.
- El nombre del array es convertido automáticamente a un puntero al primer elemento del array.
- El parámetro en la función se definirá como `tipo nombreArray[]` o `tipo *nombreArray`.

```
1   /* addArrays.c */
2   #include<stdio.h>
3   #define SIZE 4
4
5   int main()
6   {
7     int i;
8     int totalIncomes[SIZE] = {4,3,4,6};
9     int lastMonth[SIZE] = {1,2,3,1};
10
11    addArrays(totalIncomes,lastMonth,SIZE);
12
13    for(i=0;i<SIZE;i++)
14      printf("totalIncomes[ %d]:  %d\n",i,totalIncomes[i]);
15  }
16
17  int addArrays(int a[], int *b, int len)
18  {
19    int i;
20
21    for(i=0;i<len;i++)  a[i] += b[i];
22  }
```

Arrays como argumentos de una función

- Al pasar un array multidimensional como argumento a una función, ésta recibe un puntero al primer elemento del array (que es otro array).
- Es necesario indicar, en la definición de la función, el tamaño de los elementos (arrays) del array que se pasa por parámetro.

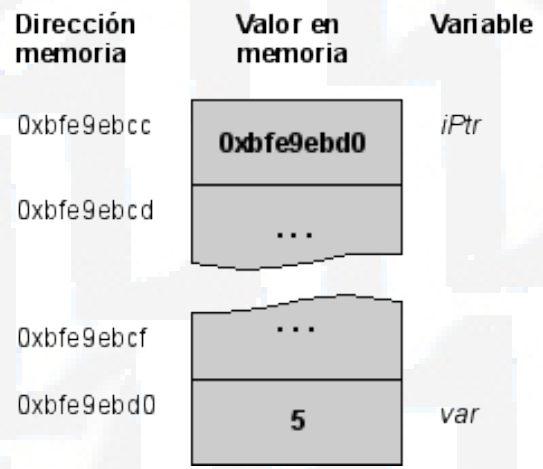
```
1  #include<stdio.h>  /* addMatrices.c */
2  #define ROWS 2
3  #define COLS 3
4
5  int main()
6  {
7    int i,j;
8    int totalIncomes[ROWS][COLS] = {{4,3,4},{2,1,3}};
9    int lastMonth[ROWS][COLS] = {{1,2,3},{4,2,1}};
10
11   addMatrices(totalIncomes,lastMonth,ROWS,COLS);
12
13   for(i=0;i<ROWS;i++)
14    for(j=0;j<COLS;j++)
15      printf("totalIncomes[%d][%d]: %d\n",i,j,totalIncomes[i][j]);
16 }
17
18 int addMatrices(int a[][COLS], int (*b)[COLS], int rows,int cols)
19 {
20   int i,j;
21
22   for(i=0;i<rows;i++)
23    for(j=0;j<cols;j++)
24         a[i][j] += b[i][j];
25 }
```

Punteros

- El valor de cada variable está almacenado en un lugar determinado de la memoria, caracterizado por una dirección.
- En ocasiones es más útil trabajar con las direcciones que con los propios nombres de las variables (reordenación de grandes volúmenes de datos, paso de argumentos a funciones, etc.).
- El **operador dirección** (&) nos permite obtener la dirección de una variable.
- Un **puntero** es un tipo de variable que puede contener la dirección de otra variable. Se declaran utilizando el símbolo *

```
1  /* punteros.c */
2  #include<stdio.h>
3
4  int main()
5  {
6    int var;
7    int *iPtr; //Puntero a entero
8
9    //iPtr apunta a la direccion de var
10   iPtr = &var;
11   var = 5;
12
13   printf( "Valor de iPtr:\t  %p\n"
14           "Dir de var:\t  %p\n"
15           "Dir de iPtr:\t  %p\n"
16           "Valor de *iPtr:\t  %d\n",
17           iPtr, &var ,&iPtr, *iPtr );
18 }
```

```
./punteros
Valor de iPtr:   0xbfe9ebd0
Dir de var:      0xbfe9ebd0
Dir de iPtr:     0xbfe9ebcc
Valor de *iPtr:  5
```



Punteros

- **Explicación:**
  - `iPtr = &var`: Se asigna a `iPtr` la dirección de `var`
  - `*iPtr` accede al contenido de la dirección a la que apunta `iPtr` (el valor de `var`) para leerlo o modificarlo (p.ej, haciendo `*iPtr = 6`)
- Las constantes y expresiones no tienen dirección, por lo que no se les puede aplicar el operador &. Tampoco se puede cambiar la dirección de una variable ni asignar una dirección a un puntero directamente. Ejemplos de sentencias *ilegales*:

```
iPtr = &46
iPtr = &(i+1)
&var = iPtr
iPtr = 2345325
          //Sería correcto: iPtr = (int *)2345325
```

- Un **puntero nulo** es aquel inicializado a `NULL`. Es decir, no apunta a nada.
- Un **puntero a `void`** puede apuntar a cualquier objeto, independientemente de su tipo.

```
//Definición:  void *calloc(size_t nmemb, size_t size);
int *iPtr = calloc( 1000, sizeof(int) ); //conversión en la asignación
```

Operaciones con punteros

- Lectura y modificación de objetos:

```
1  double x, y, *ptr, *ptr2; // Dos variables double
2                            // y dos punteros a double.
3  ptr = &x;               // ptr apunta a x.
4  *ptr = 7.8;             // Asigna el valor 7.8 a x.
5  *ptr *= 2.5;            // Multiplica x por 2.5.
6  y = *ptr + 0.5;         // Asigna a y el resultado de x + 0.5.
7  ptr2 = ptr;             // ptr2 apunta también a x
8  *ptr2 -= 2.0;           // Resta 2 al valor de x
9  (*ptr2)++;              // Incrementa en 1 el valor de x
```

- **Aritmética de punteros:** `p = p+1` apunta a la dirección siguiente a la que apuntaba, teniendo en cuenta el tipo de dato.
  - Ej: si `int *p`, `p` se incrementa `sizeof(int)` bytes, para apuntar al siguiente dato del mismo tipo.
  - Otros ejemplos:

```
1  p = p + 1;
2  p = p + i;
3  p += 1;
4  p++;
```

Punteros y arrays

- El nombre de un array no es más que un puntero *constante* que apunta al primero de un conjunto de elementos.

```
1  /* punterosarrays.c */
2  #include<stdio.h>
3
4  int main()
5  {
6    int a[5] = {1,2,3,4,5}; //a apunta al primer elemento
7    int *pa;
8    int x;
9
10   pa = a; //pa apunta al primer elmento de a
11
12   x = *pa;     // equivale a x=a[0]
13   x = *(pa+1); // equivale a x=a[1]
14
15   pa = &a[0]; // equivale a pa = a
16   pa = &a[1]; // apunta a "2"
17   x = *pa; // equivale a x = a[1]
18
19   x = *(a+3); // equivale a x = a[3]
20   pa = a+3;   // apunta a a[3]
21 }
```

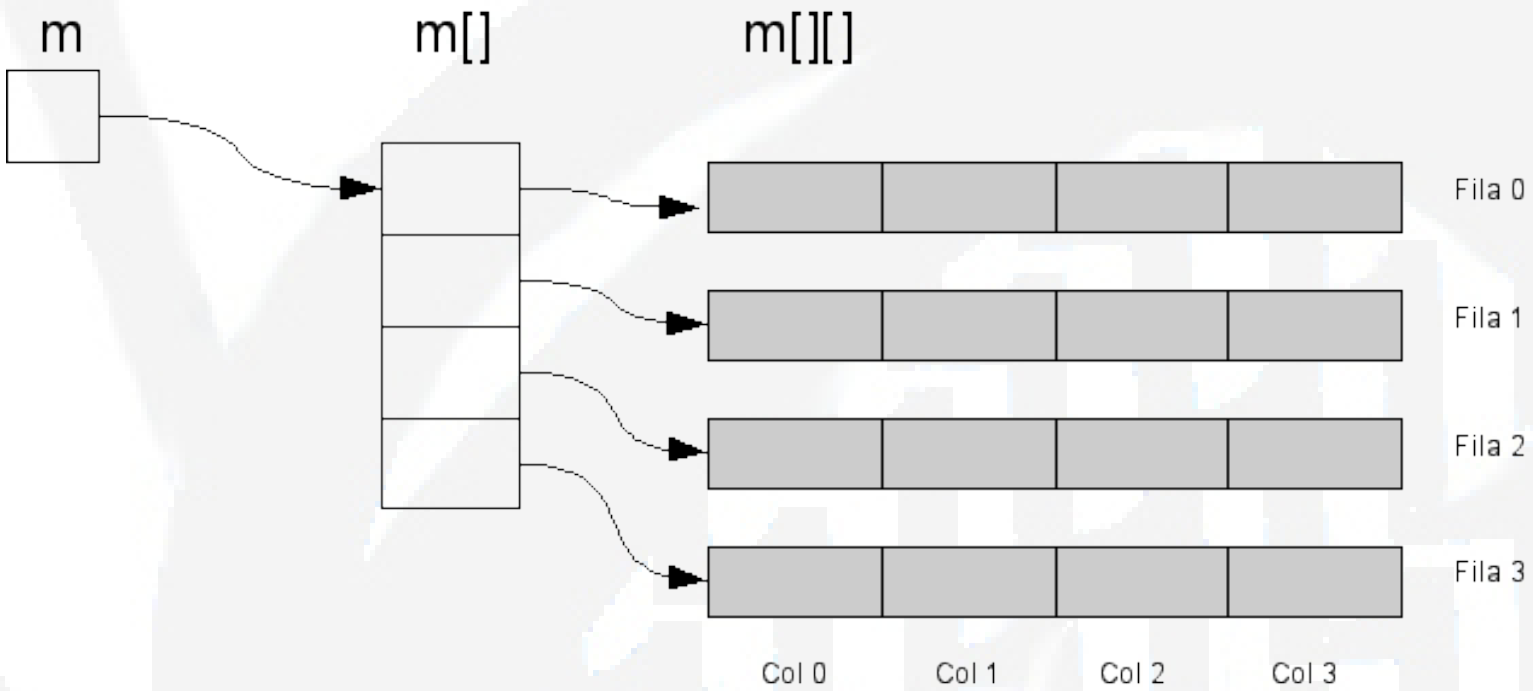
- Al ser `a` un puntero, obedece a la aritmética de punteros. Por tanto, `a[i]` y `*(a+i)` son equivalentes.
- Análogamente, `&a[i]` y `a+i` son equivalentes.

Punteros y matrices

- Supongamos la siguiente declaración:

```
int m[4][4], (*p)[4];
```

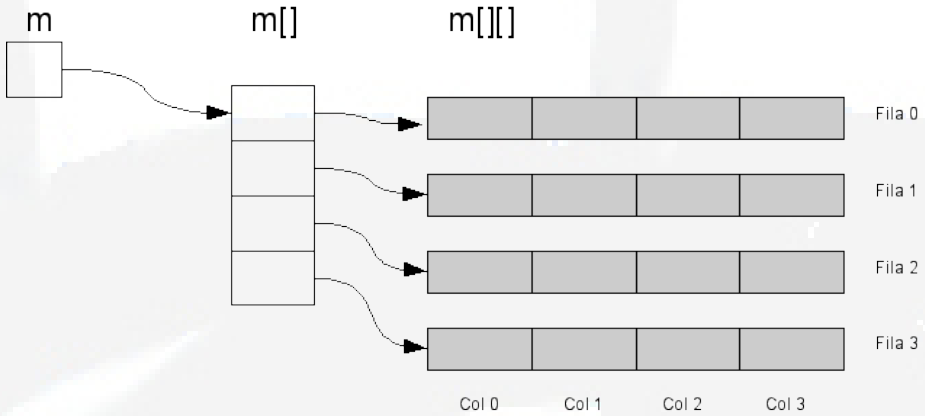
- `m` define una matriz, pero también se comporta como un *puntero a puntero*: apunta al primer elemento de un vector de punteros `m[]`, cuyos elementos contienen las direcciones del primer elemento de cada fila de la matriz.



Punteros y matrices

- `m[]` se crea automáticamente al crearse la matriz.
- Algunas equivalencias:

```
m es igual a &m[0]
m[0] es igual a &m[0][0]
m[1] es igual a &m[1][0]
m[2] es igual a &m[2][0]
(m+i) apunta a m[i]
```



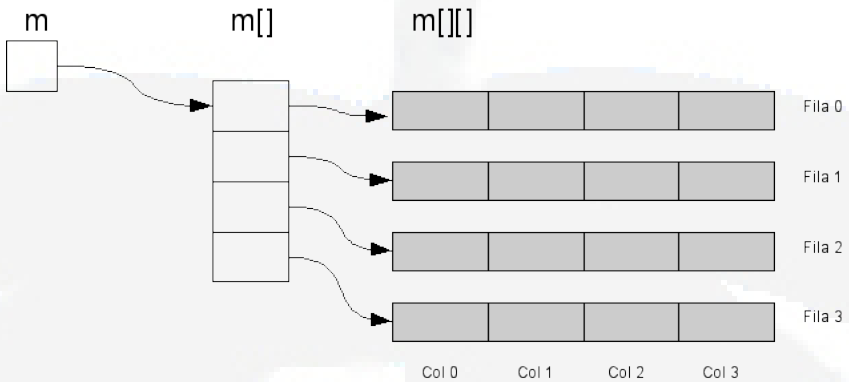
- `int (*p)[4]` es un *puntero a arrays de 4 elementos*. Es decir, sólo puede apuntar a arrays de longitud 4 cuyo contenido sean `int`. **No confundir con** `int *p[4]`, que sería un array de 4 punteros a `int` ([ ] tiene preferencia sobre *)
- Si hacemos `p = m` tendremos varias formas de acceder a la matriz:

```
*p es el valor de m[0] (dirección de la primera fila)
**p es el valor de m[0][0] (elemento [0,0] de la matriz)
*(p+1) es el valor de m[1] (dirección de la segunda fila)
**(p+1) es el valor del primer valor apuntado por m[1] (m[1][0])
```

Punteros y matrices

- **Más ejemplos:**

```
*(*(p+2)+3)
```



- `p+2` apunta a `m[2]` (Fila 2)
- `*(p+2)` es el contenido de `m[2]`: toda la Fila 2. En realidad, la Fila 2 es un array unidimensional, por lo que `*(p+2)` apunta realmente al primer elemento de ese array.
- `*(p+2)+3` apunta al cuarto elemento del array Fila 2
- `*(*(p+2)+3)` es el contenido del cuarto elemento del array Fila 2: `m[2][3]`

- `(*p)[i]` apunta a `m[0][i]`. En general, `(*p)[i]` equivale a `*((*p)+i)`. Para `i=0`, tenemos `*((*p)) = **p` Es decir, el primer elemento de la matriz

Punteros y matrices

- Supongamos otra declaración:

```
int m[4][4], *q;
```

- Podemos hacer `q = *m`, porque el contenido de `m` es de tipo (* int), es decir, puntero a entero. Tendremos nuevas formas de acceder a la matriz:

```
q (igual a *m) apunta a m[0] (dirección de la primera fila)
*q (igual a **m) es el valor de m[0][0] (elemento [0,0] de la matriz)
*(q+3) es el valor de m[0][3]
*(q+4) es el valor de m[1][0]

Para una matriz NxM: *(q + M*i + j) es el valor de m[i][j]
```

Punteros a funciones

- Es posible definir punteros a funciones, lo que permite pasar el nombre de una función como argumento de otra función.
- Por ejemplo, puede ser útil pasarle a una función de reordenamiento de arrays no sólo el array a reordenar, sino también un puntero a la función que contiene el algoritmo de reordenamiento. De ese modo, podemos tener diferentes algoritmos e invocar a nuestra función de reordenamiento con el que mejor convenga en un momento dado.
- También es posible almacenar punteros a funciones en arrays. Por ejemplo, se podría utilizar un array cuyos índices se correspondiesen con las teclas de un teclado. Según la tecla pulsada, el programa llamaría a la función apuntada por el puntero contenido en dicha posición del array.

Punteros a funciones

- **Ejemplo:**

```
1  #include <stdio.h>  /* puntafunc.c */
2  #include <stdlib.h>
3
4  double Suma( double x, double y )  { return x + y; }
5  double Resta( double x, double y ) { return x - y; }
6  double Mult( double x, double y )  { return x * y; }
7  double Div( double x, double y )   { return x / y; }
8
9  // Array de 5 punteros a funciones que toman dos parametros
10 // double y devuelven un double:
11 double (*tablaFunc[5])(double, double) = { Suma, Resta, Mult, Div};
12
13 // Array de punteros a strings para la salida
14 char *tablaMsg[5] = { "Suma", "Resta", "Producto", "Cociente"};
15
16 int main( )
17 {
18   int i;
19   double x = 0, y = 0;
20
21   printf( "Introduzca dos operandos:\n" );
22   if ( scanf( "%lf %lf", &x, &y ) != 2 )
23     printf( "Num. de elementos incorrecto.\n" );
24
25   for ( i = 0; i < 4; ++i )
26     printf( "%10s: %6.2f\n", tablaMsg[i], tablaFunc[i](x, y) );
27   return 0;
28 }
```

Funciones que devuelven punteros

- Recordemos que una función devuelve resultados al programa o función que la ha llamado por medio del valor de retorno y de los argumentos que hayan sido pasados por referencia.
- El uso de punteros como valor de retorno permite que una función pueda devolver algo más que un único valor de retorno. Se puede devolver un puntero al primer elemento de un array, una matriz, a una estructura, etc., lo que equivale a devolver múltiples valores.
- También se puede utilizar un puntero a `void` como valor de retorno, de forma que pueda asignarse a un puntero de cualquier tipo (como ya se ha visto):

```
Definición:  void *calloc(size_t nmemb, size_t size);

int *iPtr = calloc( 1000, sizeof(int) );
```

Gestión dinámica de memoria

- Habitualmente no sabremos la cantidad de datos que vamos a manejar o memoria que vamos a utilizar en un programa.
- La gestión dinámica de memoria permite reservar, en tiempo de ejecución, la memoria que se necesite en un momento dado.
- La librería estándar de C proporciona cuatro funciones para gestión de memoria: `malloc`, `calloc`, `realloc` y `free`, todas ellas declaradas en `stdlib.h`.

```
void *malloc( size_t size );
```

  - Reserva un bloque contiguo de memoria cuyo tamaño en bytes es, al menos, de `size` bytes. Tras la reserva, el contenido del bloque es indeterminado.

```
void *calloc( size_t count, size_t size );
```

  - Reserva un bloque contiguo de memoria cuyo tamaño en bytes es, al menos, de `count` x `size` bytes. Es decir, puede alojar un array de `count` elementos, cada uno de los cuales ocupa `size` bytes. Cada byte del bloque es inicializado a 0.
- Ambos devuelven un puntero a `void` con la dirección de inicio del bloque reservado.

Gestión dinámica de memoria

- Para liberar el bloque de memoria apuntado por `ptr`:

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

- Para liberar el bloque de memoria apuntado por `ptr` y reservar un nuevo bloque de memoria de tamaño `size`, devolviendo un puntero al inicio del nuevo bloque:

```
void *realloc( void * ptr, size_t size );
```

- Una función útil y relacionada con las anteriores, declarada en `string.h`, es `memset`:

```
void *memset(void *s, int c, size_t n);
```

Rellena los primeros `n` bytes del área de memoria apuntada por `s` con el byte `c`, y devuelve un puntero al área de memoria `s`.

Gestión dinámica de memoria

- **Ejemplo:** Producto matriz-vector

```
1  // prodmat.c: Multiplica una
2  // matriz a (NxN) por un
3  // vector x. Resultado en y
4  #include<stdio.h>
5  #include<stdlib.h>
6  #include<string.h>
7
8  int **inicializaMatriz(int);
9  int *inicializaVector(int);
10 void muestraMatriz(int **, int);
11 void muestraVector(int *, int);
12 void producto(int **, int*, int *, int);
13
14 int main()
15 {
16   int **a; //matriz
17   int *x,*y; //vectores
18   int N = 5; //tamano matriz
19
20   a = inicializaMatriz(N);
21   muestraMatriz(a,N);
22
23   x = inicializaVector(N);
24   //Queremos los valores de x a -1
25   memset(x,-1,N*sizeof(int));
26   muestraVector(x,N);
27   y = inicializaVector(N);
29   muestraVector(y,N);
30
31   producto(a,x,y,N);
32   printf("\nResultado del producto:");
33   muestraVector(y,N);
34 }
35
36 int **inicializaMatriz(int n)
37 {
38   int **mat;
39   register int i,j;
40
41   /* Reservamos memoria para un bloque que
42    contendra punteros a punteros a int. */
43   mat = (int **)malloc( n * sizeof(int *));
44   /* Reservamos memoria para cada una
45    de las filas */
46   for(i=0;i<n;i++)
47     mat[i] = (int *)malloc(n * sizeof(int));
48
49   //Rellenamos matriz
50   for(i=0;i<n;i++)
51     for(j=0;j<n;j++)
52       mat[i][j] = (int)(i+j);
53
54   return mat;
55 }
```

Gestión dinámica de memoria

- **Ejemplo:** Producto matriz-vector (cont.)

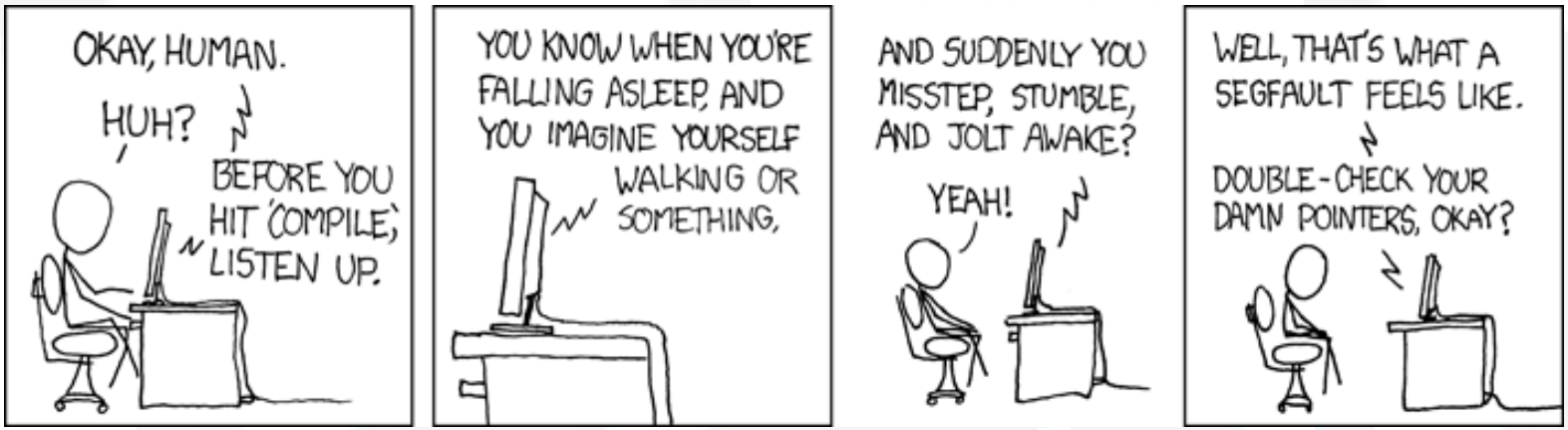
```
56
57  int *inicializaVector(int n)
58  {
59   int *v;
60   register int i;
61
62   /* Reservamos memoria para un bloque
63      que contendra punteros a int.
64      v apunta al inicio del bloque
65      El bloque se inicializa a 0 */
66   v = (int *)calloc( n, sizeof(int));
67
68   return v;
69  }
70
71
72  void producto(int **mat, int *x, int *y,
          int n)
73  {
74   register int i,j;
75
76   for(i=0;i<n;i++)
77     for(j=0;j<n;j++)
78        y[i] += mat[i][j] * x[j];
79  }
```

```
81
82  void muestraMatriz(int **mat, int n)
83  {
84    register int i,j;
85
86    for(i=0;i<n;i++)
87    {
88      //nueva fila
89      printf("| ");
90      for(j=0;j<n;j++)
91        printf(" %i\t",mat[i][j]);
92      printf("|\n");
93    }
94
95  }
96
97  void muestraVector(int *v, int n)
98  {
99    register int i;
100
101   printf("\n| ");
102   for(i=0;i<n;i++)
103   {
104    printf(" %i\t",v[i]);
105   }
106   printf("|\n");
107 }
```

Gestión dinámica de memoria

- `inicializaVector()` devuelve un puntero a un bloque de enteros. Concretamente, `calloc` devuelve un puntero a un bloque de `n*sizeof(int)` bytes, inicializado a 0.
- Aunque el bloque de memoria haya sido asignado dentro de una función, dicho bloque sigue siendo accesible, tras la finalización de la función, gracias al puntero devuelto por la función a `main`.
- `inicializaMatriz()` devuelve un puntero a punteros a `int`: alternativa a las matrices.
  - **Línea 42:** Reserva memoria para un array (bloque de memoria) de `n` *punteros a* `int` (filas de la matriz).
  - **Línea 46:** Con un bucle `for` se reserva memoria para `n` bloques de memoria de `n` elementos de tamaño `int` cada uno. El puntero a cada bloque devuelto por malloc se asigna a cada uno de los punteros elementos del array de punteros.
  - Tenemos, por tanto, un array de punteros cuyos elementos apuntan a arrays de enteros: **una matriz**.
  - Como ventaja, es posible definir así una matriz cuyas filas con columnas de diferente tamaño.
  - Como desventaja, no es seguro que las filas se ubiquen en memoria en posiciones consecutivas.
- A la función `producto()` se le pasa la "matriz" como `**mat` y se trabaja internamente como una matriz normal, accediendo a sus elementos con `[ ]`.

Laboratorio 6



(Tomada de http://xkcd.com/)

Índice

1. Introducción
2. Tipos, declaraciones y expresiones
3. Control de flujo
4. Funciones
5. Entrada/salida
6. Arrays y Punteros
7. Tipos avanzados
8. Librerías
9. El preprocesador de C

Estructuras

- Una **estructura** permite agrupar un conjunto de datos de distinto tipo bajo un mismo nombre o identificador para facilitar su manejo.
- La palabra clave `struct` introduce su declaración. Opcionalmente, puede seguir un nombre de estructura (no es el nombre de la variable) que se usará posteriormente para abreviar la estructura. Al final de la declaración pueden ir los nombres de las variables:

```
1  struct fecha {
2     int dia;
3     int mes;
4     int year;
5     char nombre_del_mes[10];
6  } fecha1, fecha2, fechanacimiento;
```

```
1  struct Cancion {
2    char titulo[64];
3    char artista[32];
4    char compositor[32];
5    short duracion;
6    struct Date publicacion;
7  };
```

- Posteriormente se puede usar el nombre de la estructura para declarar nuevas variables:

```
struct fecha fechahoy;
```

- También se pueden realizar asignaciones iniciales en la declaración:

```
struct fecha fechanacimiento = {25, 12, 1970, ``Enero''};
```

Estructuras

- Una estructura no puede contenerse a sí misma como miembro (ya que su definición aún no está completa). Sí pueden contener punteros a estructuras del mismo tipo (por ejemplo, en listas enlazadas):

```
1  struct ElementoCancion {
2     struct Cancion cancion;    // Informacion sobre la cancion.
3     struct ElementoCancion *pNext;  // Puntero a la siguiente cancion.
4  };
```

- **Acceso a un miembro de la estructura:** se usa el "." para conectar el nombre de la estructura con el del miembro:

```
1   /* structSongs.c */
2   #include <stdio.h>
3   #include <string.h>
4
5   struct Date{
6     int mes;
7     int anyo;
8   };
9
10  struct Cancion {
11   char titulo[64];
12   char artista[32];
13   char compositor[32];
14   short duracion;
15   struct Date publicacion;
16  };
```

```
18  int main()
19  {
20    struct Cancion cancion1,cancion2;
21    struct Cancion *pCancion = &cancion1;
22                  // puntero a cancion1.
23    strcpy( cancion1.titulo, "Havana Club" );
24    strcpy( cancion1.compositor, "Ottmar Liebert" );
25    cancion1.duracion = 251;
26    cancion1.publicacion.anyo = 1998;
27
28    if ( (*pCancion).duracion > 180 )
29      printf( "La cancion %s es mas larga de 3
            minutos.\n", (*pCancion).titulo );
30
31    cancion2 = cancion1; //Copiamos la cancion
32  }
```

Estructuras

- Si se tiene un puntero a una estructura, se utiliza el operador ”->“ para acceder a los miembros. Así, `p->m` es equivalente a `(*p).m`. En el ejemplo anterior podemos reescribir:

```
1  if ( pCancion->duracion > 180 )
2    printf( "La cancion %s es mas larga de 3 minutos.\n", pCancion->titulo );
```

- Generalmente, las estructuras se pasan a las funciones por referencia, debido a que pueden tener gran tamaño.

```
1  void listarCancion( const struct Cancion *pCancion )
2  {
3    int m = pCancion->duracion / 60,
4        s = pCancion->duracion % 60;
5    printf( "----------------------------------------\n"
6            "Titulo:           %s\n"
7            "Artista:          %s\n"
8            "Compositor:        %s\n"
9            "Duracion:    %d:%02d\n"
10           "Fecha:            %d\n",
11           pCancion->titulo, pCancion->artista, pCancion->compositor, m, s,
12           pCancion->publicacion.anyo );
13 }
```

Estructuras

- Se pueden declarar arrays de estructuras:

```
struct Cancion miColeccion[100];
```

- Y se podría acceder al elemento 10, por ejemplo, de la siguiente forma:

```
miColeccion[9].publicacion.anyo = 2009;
```

Uniones

- Una **unión** es una variable que puede contener, en distintos momentos, objetos de tipos y tamaños distintos.
- Sirve para manipular distintos datos con distintos tipos en la misma zona de memoria en instantes diferentes de tiempo.
- Su declaración y acceso a sus miembros es similar a la de las estructuras:

```
1  union{
2    char c;
3    int i;
4    float f;
5  } conversion;
```

- La unión `conversion` es lo suficientemente grande como para contener el mayor de los tres tipo. Es decir, no contiene los tres a la vez, sino cualquiera de ellos en un cierto instante.

Uniones

- **Ejemplo:**

```c
/* union.c */
#include <stdio.h>

int main()
{

 union Conversion{
   char c;
   int i;
   float f;
 } conversion;

 conversion.c = 'a';
 printf("Caracter: %c\n",conversion.c);

 conversion.i = 15;
 printf("Entero: %d\n",conversion.i);

 conversion.f = 3.14159;
 printf("Float: %f\n",conversion.f);
}
```

```
./union
Caracter: a
Entero: 15
Float: 3.141590
```

Typedef

- **`typedef`** permite crear nuevos tipos de datos:

```
1  /* typedef.c */
2  #define MAX_NOM      30
3  #define MAX_ALUMNOS 400
4
5  struct alumno {
6        char nombre[MAX_NOM];
7        short edad;
8  };
9  typedef struct alumno alumno_t;
10 typedef struct alumno *alumno_ptr;
11
12 struct clase {
13           alumno_t alumnos[MAX_ALUMNOS];
14           char   nom_profesor[MAX_NOM];
15 };
16 typedef struct clase clase_t;
17 typedef struct clase *clase_ptr;
18
19 int anade_a_clase(alumno_t un_alumno, clase_ptr clase)
20 {
21      alumno_ptr otro_alumno;
22      otro_alumno = (alumno_ptr) malloc(sizeof(alumno_t));
23      otro_alumno->edad = 23;
24      ...
25      clase->alumnos[0]=alumno;
26      ...
27      return 0;
28 }
```

Typedef

- En el ejemplo anterior, se crean los tipos `alumno_t` y `clase_t` (estructuras) y los tipos `alumno_ptr` y `clase_ptr` (punteros a estructura)
- `typedef` ayuda a controlar problemas de portabilidad y facilita la documentación del programa.

Enumeraciones

- El tipo **enum** permite definir los posibles valores **de tipo `int`** de ciertos identificadores o variables que únicamente pueden tomar un número discreto de valores:

```
enum color { negro, rojo, verde, amarillo, azul, blanco=7, gris};
```

- `color` sólo puede tomar uno de esos siete valores constantes de tipo `int`.
- Cada constante de un tipo `enum` representa un cierto valor que es determinado, bien implícitamente por su posición en la lista, o explícitamente mediante inicialización. Una constante sin inicialización tiene el valor 0 si es la primera en la lista, o el valor de la constante que le precede más 1. En el caso anterior, las constantes tienen los valores 0, 1, 2, 3, 4, 7, 8.

Enumeraciones

- Ejemplo de uso de `enum`:

```
enum color bgColor = azul,
fgColor = amarillo;
void setFgColor( enum color fgc );
```

- Diferentes constantes en una enumeración pueden tener el mismo valor:

```
enum { OFF, ON, STOP = 0, GO = 1, CLOSED = 0, OPEN = 1 };
```

- Si no definimos un nombre para un tipo `enum`, como en el caso anterior, nos limitamos a definir una serie de constantes.

Laboratorio 7

(Tomada de http://tux.crystalxp.net/)

Índice

1 Introducción
2 Tipos, declaraciones y expresiones
3 Control de flujo
4 Funciones
5 Entrada/salida
6 Arrays y Punteros
7 Tipos avanzados
8 Librerías
9 El preprocesador de C

La librería estándar de C

- Las librerías son conjuntos de funciones compiladas, normalmente con una finalidad análoga o relacionada, que se guardan bajo un determinado nombre, listas para ser utilizadas por cualquier usuario.
- Existe una *librería estándar de C* que implementa un conjunto de funciones predefinidas con el fin de facilitar la vida al programador.
- Existen también una serie de ficheros de cabecera que definen un subconjunto de esas funciones predefinidas, macros y tipos relacionados. Por ejemplo: *stdio.h* para funciones de entrada/salida, *math.h* para funciones matemáticas, etc. En Linux se pueden encontrar en `/usr/include`.
- Para utilizar el conjunto de funciones definido en un fichero de cabecera, se utiliza la directiva `include`:

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

- El estándar de C define 24 ficheros de cabecera estándar (los marcados con un * fueron añadidos en la versión C99):

assert.h	inttypes.h*	signal.h	stdlib.h
complex.h*	iso646.h	stdarg.h	string.h
ctype.h	limits.h	stdbool.h*	tgmath.h*
errno.h	locale.h	stddef.h	time.h
fenv.h*	math.h	stdint.h*	wchar.h
float.h	setjmp.h	stdio.h	wctype.h

La librería estándar de C

- Durante la fase de enlazado, el compilador debe añadir no sólo los ficheros objeto que forman nuestro programa, sino también el código de las funciones de la librería estándar que hayamos usado.
- No se enlaza toda la librería, sino sólo los ficheros objeto que contienen las funciones que hemos usado en nuestro programa.
- En UNIX/Linux, las funciones de la librería estándar se encuentran en *libc.a* (.a por *archive*) y en la versión equivalente para enlazado dinámico *libc.so* (.so por *shared object*). Generalmente, ambas se alojan en `/lib/` o `/usr/lib/`.
- Ciertas funciones están contenidas en ficheros de librería separados, tales como la librería matemática de punto flotante. Por ejemplo, si tenemos un código con las siguientes líneas:

```
1  #include <math.h>
2  ...
3  main{
4    ...
5    printf(" %f\n",sin(1.0));
6    ...
7  }
```

necesitaremos compilar llamando a la librería matemática, donde está definida la función `sin`. La libería se llama `libm.a` (o `libm.so`), por lo que el nombre con el que se la invocará será el suyo quitando ”lib“ del principio y usando la opción **`-l`**:

```
gcc -o ejmath ejmath.c -lm
```

La librería estándar de C

- Cuando se incluye una librería, C busca un fichero con ese nombre en los directorios estándar para librerías, como `lib` y `/usr/lib`. Si la librería no está en ninguna de esas ubicaciones, existen tres formas de enlazarla:

  1. Si tenemos, por ejemplo, una librería llamada `libpropia.a` ubicada en `/usr/local/lib`, se añadirá la ruta completa después de llamar a los ficheros (fuente u objeto) que la necesiten:

     ```
     gcc -o ejmath ejmath.c /usr/local/lib/libpropia.a
     ```

  2. Se puede utilizar la opción `-L` para indicarle al compilador una ruta adicional donde buscar librerías:

     ```
     gcc -o ejmath ejmath.c -L/usr/local/lib -lpropia
     ```

     Si necesitamos añadir también rutas no estándar donde el compilador deba buscar ficheros de cabecera (`*.h`), utilizaremos la opción `-I`.

  3. La tercera forma consiste en añadir la ruta donde se encuentra la librería a incluir en la variable de entorno `LIBRARY_PATH`.

Librerías compartidas y enlazado dinámico

- Las **librerías compartidas** (*shared libraries*) son ficheros objeto especiales que pueden enlazarse a un programa en tiempo de ejecución. De este modo, al compilar un programa se obtendrá un ejecutable más pequeño, y las librerías compartidas podrán ser utilizadas por varios programas simultáneamente, lo que permite una gestión más eficiente de la memoria.
- Para crear una librería compartida, se debe utilizar la opción `-shared` y usar un fichero objeto ya existente:

```
gcc -c funciones.c
gcc -shared -o libfunciones.so funciones.o
```

Se puede enlazar como un fichero objeto normal o como una librería:

```
gcc -o prog_principal prog_principal.c libfunciones.so
```

Las funciones contenidas en `funciones.so` se enlazarán dinámicamente en tiempo de ejecución. Para ello, es necesario asegurarse de que el programa puede encontrar dicho fichero. Deberá estar en alguno de los directorios estándar (ej: `/usr/lib`) o bien deberá indicarse el directorio en el que está contenido con la variable de entorno `LD_LIBRARY_PATH`.

- Al usar una librería, el compilador buscará primero la versión dinámica (`.so`). Si no la encuentra, buscará la versión estática. Si se tienen las dos versiones de una librería y se quiere utilizar la versión estática debe indicarse al compilar el flag `-static`.

Librerías estáticas

- Ya se comentó previamente que las librerías estáticas son un conjunto de ficheros objeto empaquetados en un único archivo con la extensión `.a`.
- Para crear una librería estática:
  1. Creamos los ficheros objeto a incluir en la librería:

```
gcc -c funciones.c
gcc -c funciones2.c
...
```

  2. Se empaquetan los ficheros obtenidos con la utilidad `ar`:

```
ar rcs libfunciones.a funciones.o funciones2.o
```

  donde:

  - r : Reemplazar los ficheros si ya existían en el paquete.
  - c : Crear el paquete si no existe.
  - s : Construir un índice del contenido.

- `ar` ofrece otras opciones útiles:
  - t : Lista el contenido de una librería.
  - x : Extrae un fichero de una librería.

Laboratorio 8

Índice

1 Introducción
2 Tipos, declaraciones y expresiones
3 Control de flujo
4 Funciones
5 Entrada/salida
6 Arrays y Punteros
7 Tipos avanzados
8 Librerías
9 El preprocesador de C

Directivas

- Cuando se invoca al compilador, existe una fase previa a la compilación real, en la que el *preprocesador de C* prepara el código fuente para ser compilado y evalúa unas **directivas de preprocesado**.
- Las directivas permiten insertar código de otros ficheros, identificar secciones de código que deben ser compiladas únicamente bajo ciertas condiciones y definir macros (identificadores que el preprocesador reemplaza con otro texto).
- Cada directiva debe aparecer en una línea, comenzando por el carácter `#`, que únicamente puede ir precedido por espacios o tabulaciones.
- El preprocesador de C entiende las siguientes directivas:

```
#define, #undef
#if, #ifdef, #ifndef, #endif, #else, #elif
#include
#pragma
#error
```

Directiva *define*

- Permite definir *macros*. Con `define` se le puede dar un nombre a cualquier texto que se desee, ya sea una constante o una sentencia, y el preprocesador sustituirá todas sus ocurrencias en el código durante la compilación. Ejemplos de macros sin parámetros:

```
#define TAM_MAX_NOMBRE  256  // Constante
#define SIZE_BUFFER  (4*256)
#define RANDOM  (-1.0 + 2.0*(double)rand( ) / RAND_MAX)
```

- Ejemplos de macros con parámetros:

```
#define getchar( )   getc(stdin)  //Incluida en stdio.h
#define putchar(x)  putc(x, stdout) //Incluida en stdio.h
#define DISTANCIA( x, y )  ((x)>=(y) ? (x)-(y) : (y)-(x))
```

- `#undef` permite cancelar la definición de una macro.

Directivas de compilación condicional

- Permite incluir u omitir determinadas partes del código fuente dependiendo de determinadas condiciones. Una sección condicional comienza con las directivas `#if`, `#ifdef` o `#ifndef`, y termina con la directiva `#endif`.
- Una sección con `if` tiene la siguiente forma:

```
#if expresion1
  [ grupo1 ]
[#elif expresion2
  [ grupo2 ]]
...
[#elif expresion(n)
  [ grupo(n) ]]
[#else
  [ grupo(n+1) ]]
#endif
```

- La macro `defined` devuelve 1 si el identificador especificado como parámetro es una macro. Es decir, si ha sido definido previamente:

```
#if defined( _  _unix_  _ ) && defined( _  _GNUC_  _ )
/* ... */
#endif
```

- Para comprobar (en expresiones cortas) si una macro ha sido definida previamente, se pueden utilizar:

```
#ifdef identificador
#ifndef identificador
```

Directiva *include*

- Indica al preprocesador que incluya el contenido de un determinado fichero de cabecera:

```
#include <stdio.h>  // Incluye header de la librería estándar
#include "funciones.h" // Incluye header propio
```

- Puede ocurrir que `funciones.h` llame a su vez a `stdio.h`. Para evitar múltiples inclusiones del mismo fichero de cabecera, se puede hacer lo siguiente, suponiendo un fichero `miProyecto.h`:

```
#ifndef MIPROYECTO_H_
#define MIPROYECTO_H_
/* ... Los contenidos de miProyecto.h van aquí ... */
#endif   /* MIPROYECTO_H_ */
```

En la primera ocurrencia de una directiva que incluya `miProyecto.h`, la macro `MIPROYECTO_H_` no está aún definida. El preprocesador inserta, por tanto, los contenidos del bloque entre `#ifndef` y `#endif` incluyendo la definición de la macro `MIPROYECTO_H_`. Si hay inserciones posteriores de `miProyecto.h`, la condición `#ifndef` es falsa, y el preprocesador descarta el bloque hasta `#endif`.

Directivas *error* y *pragma*

- Indica al preprocesador que muestre un mensaje de error. El compilador deja de procesar el fichero fuente y, tras mostrar el mensaje, termina.

```
#ifndef _ _GNUC_ _
#error "Este código no puede ser compilado con GCC"
#endif
```

- La directiva `#pragma` permite proporcionar información adicional al compilador. Tiene la forma:

```
#pragma [tokens]
```

Si el primer *token* es `STDC`, entonces la directiva es un pragma estándar. Si el preprocesador reconoce los tokens especificados, realiza la acción especificada.

Laboratorio 9

Algunas referencias

1. Javier García de Jalón et al. *Aprenda lenguaje ANSI C como si estuviera en primero*. Universidad de Navarra, Febrero 1998.

   Libro clásico para iniciarse en la programación en C. Muy didáctico, aunque no demasiado exhaustivo, tiene algunas inconsistencias.

2. Steve Oualline. *Practical C Programming*. 3rd Edition. O'Reilly Media, Inc., August 1997.

   Bastante didáctico, incluye muchos ejemplos. No sólo cubre el lenguaje, sino que enseña "buenas costumbres" de programación.

3. Peter Prinz and Tony Crawford. *C in a Nutshell*. Copyright 2006 O'Reilly Media, Inc., 0-596-00697-7.

   El libro de consulta de C por excelencia. Denso y exhaustivo: cubre todo el lenguaje, incluyendo la revisión C99. Apto para consultar cualquier duda, pero no tanto para iniciarse.

Programación en C

Juan Ángel Lorenzo del Castillo
juanangel.lorenzo@usc.es

Grupo de Arquitectura de Computadores
Departamento de Electrónica y Computación
Universidad de Santiago de Compostela

CESGA, 11 - 15 de Julio 2011