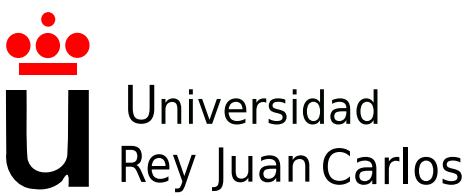
Curso de Introducción a C en Plan 9

Enrique Soriano

Laboratorio de Sistemas
Grupo de Sistemas y Comunicaciones
URJC

3 de febrero de 2010



Características de C

- Programación imperativa.
- Lenguaje simple, la funcionalidad está en las bibliotecas.
- Básicamente maneja números, caracteres y direcciones de memoria.
- No tiene tipado fuerte.

Antes de nada: cómo compilar

La compilación se compone de tres fases:

- Preprocesado: incluye ficheros de cabecera (#includes), quita comentarios, etc.
- Compilación: genera el código objeto a partir del código fuente, pasando por código ensamblador (aunque no nos demos cuenta).
- Enlazado: a partir de uno o varios ficheros objeto y bibliotecas, genera un único binario ejecutable.

Antes de nada: cómo compilar

- Preprocesado y compilación:

```
8c -FVw hello.c
```

- Enlazado:

```
8l -o hello hello.8
```

Antes de nada: cómo conseguir ayuda

- Las páginas de manual se pueden consultar con el comando `man`:
  `man sección asunto`
  P.ej.: `man 1 8c`
- Una introducción al sistema:
  `man 1 0intro`
- Secciones de interés: comandos (1), llamadas al sistema(2).
- Para buscar sobre una palabra: `lookman`.
  Por ejemplo: `lookman 8c`.

Hola mundo: disección.

```
#include <u.h>                  /* Instrucciones para el preprocesador */
#include <libc.h>

/* Comentario */

void
main(int , char **)                 /* Definición de función.
                                       Punto de entrada.            */
{                                   /* Inicio de bloque             */
     print("hola mundo");           /* Sentencia, llamada a función */
     exits(nil);                    /* Sentencia, llamada a función */
}                                   /* Fin de bloque                */
```

Cosas importantes del hola mundo

- Los `#include` tienen que seguir un orden, especificado en la página de manual correspondiente.
- Los comentarios no pueden estar anidados.
- Todas las sentencias acaban con un ";".
- Un bloque es un conjunto de sentencias que se tratan sintácticamente como una única sentencia.
- Las sentencias de una función se engloban en un bloque.
- Los bloques se determinan con llaves $\{\}$.

Cosas importantes del hola mundo

```
void
main(int , char **)    /* Definición de función.
                                    Punto de entrada.*/
```

- `main()` es la función por la que se comienza a ejecutar el programa, es lo que se denomina "punto de entrada".
- Recibe dos parámetros (por ahora no los usamos).
- La llamada de biblioteca `exits()` indica si el programa ha acabado bien o no. La constante `nil` significa que se ha acabado bien.

Tipos de datos fundamentales

Dejamos fuera los tipos reales, nos quedamos con los enteros. En Plan 9 sobre x86 32-bits tienen estos tamaños:

- `char` : *carácter* con signo (1 byte), p.ej. 'a' , 12
- `int` : entero con signo (4 bytes), p.ej. 77 -11
- `long` : entero largo con signo (4 bytes), p.ej. 77 -11
- `uchar` : entero sin signo (1 byte), p.ej. 33
- `uint` : entero sin signo (4 bytes), p.ej. 77
- `ulong` : entero largo sin signo (4 bytes), p.ej. 7

Nota: en esta plataforma, `long` e `int` son iguales.

- `void` : vacío (ya veremos para qué sirve).

Declaración e inicialización de variables

```
#include <u.h>
#include <libc.h>

int x = 1;                  /* variables globales */
int k;


void
main(int, char **)
{
     int i, q=1, u=12;      /* variables locales */
     char c;
     char p = 'o';

     c = 'z';
     i = 13;
     exits(nil);
}
```

Conversión de tipos

- En una asignación, siempre se convierte al tipo de la variable a la que se asigna el valor.
- En una operación, se convierte al tipo de *mayor* tamaño antes de realizar la operación.
- Las conversiones de un tipo menor a uno mayor se realizan insertando ceros en los bits de mayor peso.
- Las conversiones de un tipo mayor a uno menor se realizan truncando los bits de mayor peso. Ojo: ¡por lo general no quieres que te pase esto! ten cuidado con los tipos de datos.

Conversión de tipos ¡cuidado!

Ejemplo:

```
int i;
char c;

i = 10025;
c = i;

/* i es 00000000 00000000 00100111 001010001 == 10025*/
/* c es                            001010001 == 41 */
```

Declaración e inicialización

Declaración de variables:

- Las variables globales o externas.
  - Se declaran fuera de una función.
  - Son accesibles desde cualquier función.
  - Se localizan en el segmento de datos.
  - Si no se inicializan explícitamente, se inicializan a 0.
- Las variables locales o automáticas.
  - Se declaran dentro de una función.
  - Sólo son accesibles desde dentro de su ámbito, que es el bloque en el que se han declarado y sus bloques anidados.
  - Se localizan en la pila.
  - Si no se inicializan explícitamente, tienen un valor indeterminado.

Declaración e inicialización

- Una variable declarada dentro de una función como `static` conserva el valor entre distintas invocaciones. Es así porque no se localizan en la pila, sino en el segmento de datos.
- Unas variables pueden ocultar a otras, y el compilador no nos avisa.
- Para inicializar, usamos valores constantes o *literales*:
  - Entero Decimal: `777`
  - Entero Hexadecimal: `0x777`
  - Entero Octal: `0777`
  - Carácter: `'a'`

ver sombra.c

Constantes

Declaración de constantes enteras mediante el uso de tipos enumerados con `enum`:

```
enum{
        Nelementos = 10,
        Nobjetos = 5,
};
```

Ventajas

- Ofrece algo de tipado (son enteros).
- Ahorra problemas con el preprocesador: p.ej. mantiene el ámbito, mensajes de error, etc.

Nota: El tipo de los enumerados depende de la implementación, y puede ser con signo o sin signo.

Operadores aritméticos

+	Suma. Operandos enteros o reales
-	Resta. Operandos enteros o reales
*	Multiplicación. Operandos enteros o reales
/	División. Operandos enteros o reales.
%	Módulo. Operandos enteros

Nota: Para la división entera, ambos operandos deben ser enteros.

Operadores lógicos

Las operaciones lógicas devuelven un entero. Cualquier valor distinto a 0 significa TRUE, 0 significa FALSE.

`a && b`	AND. 1 si "a" y "b" son distintos de 0
`a || b`	OR. 0 si "a" y "b" son iguales a 0
`!a`	NOT. 1 si "a" es 0, 0 si es distinto de 0

Operadores de relación

`a < b`	“a” menor que “b”
`a > b`	“a” mayor que “b”
`a <= b`	“a” menor o igual que “b”
`a >= b`	“a” mayor o igual que “b”
`a != b`	“a” distinto que “b”
`a == b`	“a” igual que “b”

Operadores de asignación

++	Incremento pre: el valor de la expresión refleja el incremento post: el valor de la expresión no refleja el incremento
--	Decremento (pre o post)
=	Asignación simple
*=	Multiplicación y asignación
/=	División y asignación
%=	Módulo y asignación
+=	Suma y asignación
-=	Resta y asignación

Operadores bit a bit y otros

`&`	(unario) Dirección-de. Da la dirección de su operando
`*`	(unario) Indirección. Acceso a un valor, teniendo su dirección
`~`	(unario) Complemento
`&`	AND de bits
`^`	XOR de bits
`|`	OR de bits
`<<`	Desplazamiento binario a la izquierda
`>>`	Desplazamiento binario a la derecha
`?:`	Operador ternario
`sizeof`	Operador de tamaño

Nota: por lo general, para operaciones bit a bit debemos usar tipos sin signo.
P.ej.: el desplazamiento por la izq. conserva el signo para que sirva como una división entera por 2.

ver bits.c

Precedencia y asociatividad de operadores

Precedencia	Asociatividad
() [] -> .	izquierda a derecha
! ++ -- * & ~ sizeof (unarios)	derecha a izquierda
* / %	izquierda a derecha
+ -	izquierda a derecha
<< >>	izquierda a derecha
< <= > >=	izquierda a derecha
== !=	izquierda a derecha
& ^ |	izquierda a derecha
&&	izquierda a derecha
||	izquierda a derecha
?:	derecha a izquierda
= += -= *= /= %= &= ...	derecha a izquierda
,	izquierda a derecha

Precedencia: prioridad por clase.
Asociatividad: prioridad para los de misma precedencia.

ver operadores.c

Definición y declaración de funciones

- Una función tiene que estar *declarada* antes de poder usarla en el código, pero puede estar *definida* después. Los tipos del *prototipo* y de la definición tienen que coincidir.
- Los argumentos siempre son por valor. Si queremos argumentos por referencia, tendremos que usar un puntero (lo veremos más adelante).
- Si una función no devuelve nada, es de tipo `void`.
- Si una función no tiene argumentos, debemos poner `void` en lugar de los argumentos.

ver funciones.c

Bibliotecas estándar

Las bibliotecas son colecciones de ficheros objeto pre-compilados que podemos usar. Las bibliotecas con funciones estándar:

- `<u.h>` Definiciones de tipos y constantes dependientes de la arquitectura
- `<libc.h>` Librería estándar de C, funciones de strings, llamadas al sistema, etc.

Para buscar el prototipo de una función podemos usar el comando `sig`.

(Ahora sí) print

```
int print(char * format, ...);
```

- Tiene un número variable de parámetros.
- El primer parámetro indica en una cadena de caracteres el *formato* de lo que se quiere imprimir por pantalla.
- Cada % en el formato se sustituirá con el valor del parámetro que ocupa ese lugar **después** del formato.
- Lo que viene después del % es la forma en la que se quiere imprimir el dato: %d es un entero, %c es un carácter, %x un entero hexadecimal sin signo, %o un entero octal sin signo, %s una cadena de caracteres o string...

If

```
if ( expresión ) {
     sentencias1...
} else  {
     sentencias2...
}
```

- Si la expresión evalúa a un entero distinto de 0, no se entra al if, se entra a `else` (si lo hay).
- Los paréntesis son obligatorios.
- Si sólo hay una sentencia, podemos prescindir de las llaves.

ver par0.c

Switch

```
switch ( expresión ) {
    case valor1:
                sentencias1...
    case valor2:
                sentencias2...
    default:
                sentencias3...
}
```

- El flujo pasa por el `case` que corresponde al valor de la expresión.
- Debe haber un `case` por valor.
- La sentencia `break` rompe un bucle o un switch. Si no se rompe al final de un case, se entra a otros cases posteriores (*fall-through*).
- Si no entra por ningún `case`, entrará en `default` (si lo hay).

ver par1.c

While

```
while ( expresión ) {
    sentencias...
}
```

- Se itera hasta que la expresión evalúa a 0.
- La sentencia `break` rompe un bucle.

ver buclewhile.c

For

```
for ( inicialización ; condición ; actualización) {
    sentencias...
}
```

- La inicialización sólo se ejecuta una vez antes de la primera iteración y antes de evaluar la condición.
- Se itera en el bucle hasta que se deja de cumplir la condición.
- Al final de cada iteración se ejecuta la actualización.

ver buclefor.c

Punteros

- Las variables son una o varias direcciones de memoria contiguas con los bytes correspondientes.
- Un puntero es una variable que contiene una dirección de memoria.

Punteros

- Para declarar una variable puntero a un tipo:
  `tipodedato * nombre;`
  Por ejemplo:
  `int * ptr; /* un puntero a entero */`
- El operador unario * (*dereference*) delante de una variable de tipo puntero significa que queremos operar sobre el contenido de la dirección a la que apunta.
- El operador unario & (*address of*) delante de una variable significa que queremos operar sobre su dirección de memoria.
- No se puede atravesar un puntero que no apunta a ningún sitio (nil).

Aritmética de punteros

- Los punteros se pueden sumar, restar, etc.
- Las operaciones se hacen en múltiplos del tamaño en bytes del tipo de datos al que apunta el puntero.

```
char * cptr; /* los char ocupan 1 byte */
int * iptr;  /* los int ocupan 4 bytes */

...

cptr = cptr + 4; /* la dirección de memoria + 4 posiciones */
iptr = iptr + 4; /* la dirección de memoria + (4*4) posiciones */
```

ver punteros.c

Los *Arrays* en C

- No son más que *azúcar sintáctico* para los punteros.
- El índice para N elementos va de 0 a N-1.
- No se comprueban los límites.
- `int lista[N];` → "reserva la memoria necesaria para tener N objetos de tipo `int` contiguos y guarda la dirección en la variable `lista`".
- `lista[NUM] = 3;` → "escribe el entero 3 en la posición de memoria (NUM * tamaño de `int`) a partir de la dirección `lista`".

Los *Arrays* en C

Diferencias entre un array y un puntero:

- Un array tiene un *tamaño* asociado, un puntero no. Un array reserva automáticamente su memoria.
- Un array no puede estar a la izquierda de una asignación (no es un *l-value*). Se puede ver como un *puntero estático*.

Los *Arrays* en C

- Inicialización de arrays:

```
int lista1[5] = { 1, 2, 3, 4, 5 };  /* damos el tamaño e
                                       inicializamos            */
int lista2[] = { 1, 2, 3, 4, 5 };   /* tamaño == numero elementos
                                       en la inicialización     */
int lista3[5] = { 1, 2, 3 };        /* da igual si sobra huecos */
int lista2[4] = { 1, 2, 3, 4, 5 };  /* error!!! ATENCION!!! esto compila
```

ver arrays.c

Pasando direcciones de memoria como argumento

```
#include <u.h>
#include <libc.h>

void
dameletra(char *c)
{
    *c = 'A';
}

void
main(void)
{
    char c = 'b';

    dameletra(&c);
    print("c es %c\n", c);
    exits(nil);
}
```

ver referencia.c

Cadenas de caracteres (strings)

- Son *arrays* de caracteres acabados en un carácter `'\0'` (el carácter nulo).
- Si no se acaba en un carácter nulo, no es una string.
- Inicializar una string:

```
char str[] = "hola";                          /* inicializando una string */

char str2[] = {'h', 'o', 'l', 'a', '\0'}; /* equivalente a lo anterior */
```

Cadenas de caracteres (strings)

Funciones para manejo de cadenas (ver prototipos en las páginas de manual):

- `snprint`: similar a print, pero imprime en una cadena. Escribe como mucho size-1 caracteres mas el carácter nulo. Devuelve el número de caracteres escritos en la cadena.
- `strlen`: devuelve el tamaño de una cadena, sin contar el carácter nulo.
- `strcat`: concatena dos cadenas, la segunda al final de la primera, dejando el resultado en la primera. Devuelve un puntero a la cadena resultante. La primera cadena tiene que tener espacio suficiente como para que quepa la concatenación.

ver strings.c

Argumentos de main

- `argc`: variable entera que indica el número de argumentos que se le han pasado a `main`.
- `argv`: array de strings con cada uno de los argumentos. El primer argumento se corresponde con el nombre con el que se invoca al programa.

ver mainargs.c

Estructuras

```
struct Coordenada{
    int x;
    int y;
};

struct Coordenada c = {13, 33}; /* inicialización */
```

- El tamaño que ocupa en memoria no tiene porqué coincidir con la suma de los tipos de datos que contiene la estructura.
- Sólo se pueden hacer 3 cosas con ellas: copiarlas/asignarlas (esto incluye pasarlas como parámetro o retornarla), obtener su dirección (&), y acceder a sus campos.

Estructuras

- Se suele definir un tipo de datos nuevo con `typedef` para usarlas de forma más cómoda.
- Si tenemos un puntero a una estructura, el operador `->` sirve para acceder a sus campos:

$$\texttt{p->x} \equiv \texttt{(*p).x}$$

ver structs.c

Memoria dinámica

Free y malloc (leer la página de manual):

- `malloc`: sirve para pedir memoria en tiempo de ejecución. La memoria devuelta se localiza en el segmento *heap*.
- La memoria reservada con `malloc` puede tener cualquier contenido.
- Si no hay memoria en el sistema, devuelve `nil`.
- `free`: sirve para liberar la memoria pedida anteriormente. No se puede liberar memoria que no se ha solicitado con `malloc`.
- Hay que liberar la memoria cuando ya no nos hace falta.

ver malloc.c

Programas con varios ficheros fuente

- Las variables globales que están definidas en un fichero externo tienen que declararse como `extern`.
- Una función o variable global declarada como `static` no es visible desde otros ficheros. Si no se especifica, sí es visible aunque no estén en un fichero de cabecera incluido (aunque el compilador nos avisa).
- En el fichero de cabeceras sólo deben ir las variables, tipos de datos, constantes, etc. que formen parte de la interfaz del módulo.

Programas con varios ficheros fuente

- Cada fichero fuente debe incluir los ficheros de cabeceras que necesite. Por lo general, no se deben incluir ficheros de cabecera en otros ficheros de cabecera.
- Para incluir un fichero de cabeceras que no está en los directorios del sistema (/usr/include,...):

  ```
  #include "rutadelfichero"
  ```

  (si no lo encuentra, lo busca en los del sistema)
- No se pueden incluir dos veces un mismo fichero de cabecera.

ver manyfiles.c