Introducción a C sobre GNU/Linux

Gorka Guardiola, Enrique Soriano

GSYC

25 de septiembre de 2023

Universidad
Rey Juan Carlos

Características

- Programación imperativa estructurada.
- Relativamente de “bajo nivel” (dentro de los lenguajes de *alto nivel*, es de los de menos nivel de abstracción).
- Lenguaje simple, la funcionalidad está en las bibliotecas.
- Básicamente maneja números, caracteres y direcciones de memoria.
- No tiene tipado fuerte.

Hola mundo: un vistazo rápido.

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

/* Comentario */

int
main(int argc, char *argv[])
{
     printf("hola mundo");
     exit(EXIT_SUCCESS);
}
```

Antes de nada: cómo compilar

La compilación se compone de tres fases. El programa gcc se encarga de las tres, en realidad es un *front-end* que invoca a otros programas:

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

Antes de nada: cómo compilar

- Preprocesado y compilación:

  ```
  gcc -c -Wall -Wshadow -Wvla -g hello.c
  ```

  OJO: ¡los warnings hay que tratarlos como errores!

- Enlazado:

  ```
  gcc -o hello hello.o
  ```

Antes de nada: cómo conseguir ayuda

- Las páginas de manual se pueden consultar con el comando `man`: `man sección asunto`
  Por ejemplo: `man 1 gcc`
- Secciones de interés: comandos (1), llamadas al sistema(2), llamadas a biblioteca(3).
- Para buscar sobre una palabra: `apropos`.
  Por ejemplo: `apropos gcc`.

Hola mundo: disección.

```
#include <stdlib.h>                /* Instrucciones para el preprocesador */
#include <stdio.h>

/* Comentario */

int
main(int argc, char *argv[])    /* Definición de función.
                                   Punto de entrada.*/
{                               /* Inicio de bloque */
     printf("hola mundo");      /* Sentencia, llamada a función */
     exit(EXIT_SUCCESS);        /* 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 o *sentencia compuesta* es un grupo de sentencias que se trata sintácticamente como una única sentencia. Los bloques se determinan mediante llaves ($\{\}$). Las sentencias de una función se engloban en un bloque.

Cosas importantes del hola mundo

```
int
main(int argc, char *argv[])    /* 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, retorna un valor, que es el status del programa (más adelante veremos las funciones).
- La llamada de biblioteca `exit()` indica si el programa ha acabado bien o no. La constante `EXIT_SUCCESS` significa que se ha acabado bien. Podemos usar `EXIT_FAILURE` para indicar lo contrario.

Tipos de datos fundamentales

Dejamos fuera los tipos reales, nos quedamos con los enteros:

- `char` : carácter con signo (1 byte), p.e. 'a' , 12
- `int` : entero con signo (4 bytes), p.e. 77 -11
- `unsigned char, uchar` : carácter sin signo (1 byte), usado para operar sobre bits.
- `unsigned int, uint` : entero sin signo (4 bytes), p.e. 77
- `long`: entero largo, p.e. 431414341L
- `long long`: entero más largo, p.e. 432423432423LL

Y otro:

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

Tamaño

- C tiene tipado débil y no se queja si intentas asignar una variable a otra de distinto tamaño.
- Si se asigna a una de menor tamaño, se trunca.
- Si se desborda una variable, nadie te lo va a decir.
- Puedes asignar una con signo a una sin signo (y viceversa).
- Que el tipo tenga signo sólo se tiene en cuenta en las comparaciones.

Declaración e inicialización de variables

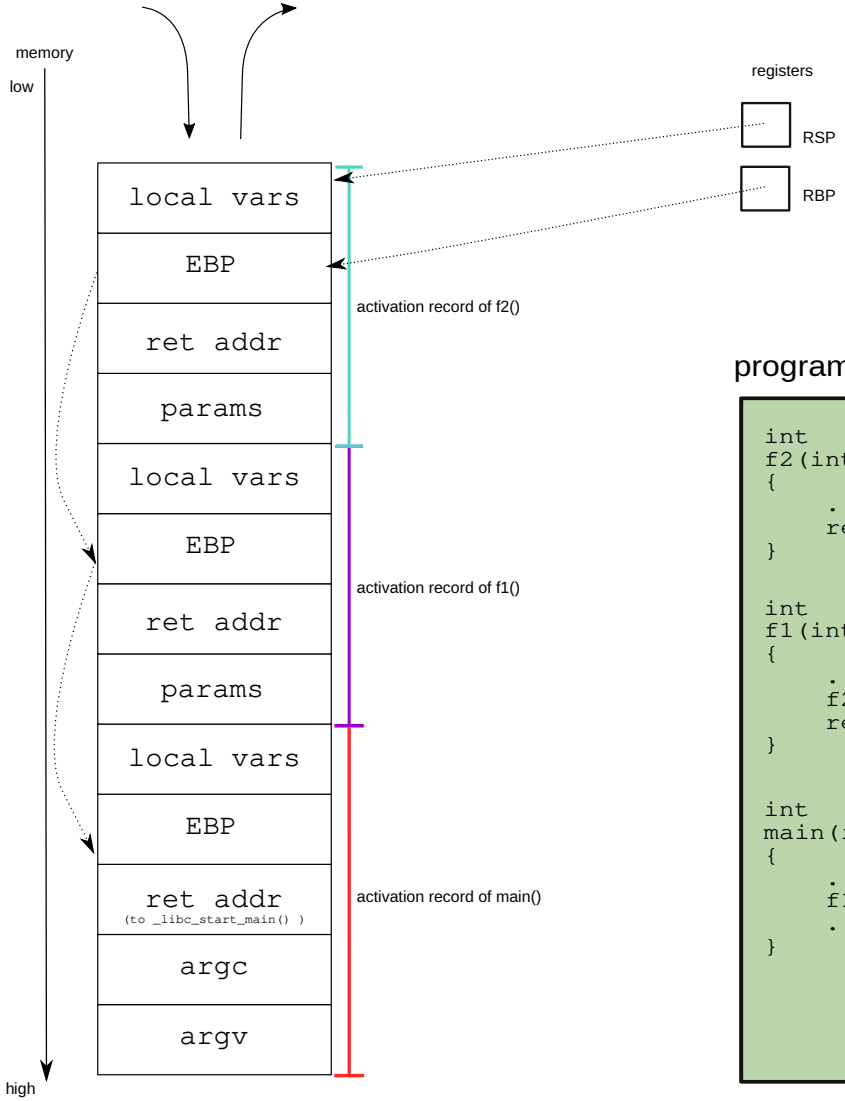
```
#include <stdlib.h>
#include <stdio.h>

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


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

     c = 'z';
     i = 13;
     exit(EXIT_SUCCESS);
}
```

Declaración e inicialización

Declaración de variables:

- Las **variables globales o externas** se declaran fuera de cualquier función. Se "ven" desde cualquier función del fichero, su ámbito es todo el fichero. 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 y sólo se pueden "ver" 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.

La pila



program

```
int
f2(int x)
{
    ...
    return y;
}

int
f1(int x)
{
    ...
    f2(3);
    return y;
}

int
main(int argc, char *argv[])
{
    ...
    f1(43);
    ...
}
```

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. Para enterarnos, usamos el modificador -Wshadow del compilador.
- Para inicializar, usamos valores constantes o *literales*:
  - Entero Decimal: `777`
  - Entero Hexadecimal: `0x777`
  - Entero Octal: `0777`
  - Carácter: `'a'`, `'\92'`

Constantes

Declaración de constantes enteras mediante el uso de tipos enumerados con `enum`. Si no se da valor, se adjudican valores consecutivos desde el último definido:

```
enum{
        Lun,
        Mar,
        Mier,
        Jue,
        Vier,
        Ndias,
        SalarioBase = 2580,
};
```

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

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

Recuerda: 0 si es falso, cualquier otro valor si es verdadero:

`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 o post)
--	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

OJO: estas operaciones sobre tipos con signo pueden sorprendernos!

`&`	(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

Precedencia y asociatividad de operadores

`() [] -> .`	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

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, entonces ponemos un argumento sin nombre de tipo `void`.

Librerías estándar

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

`<stdio.h>`	Entrada y salida estándar `printf()`, `sprintf()`, `perror()`, ...
`<stdlib.h>`	Librería estándar de C `exit()`, `atoi()`, `getenv()`, ...
`<string.h>`	Operaciones con cadenas de caracteres `strlen()`, `strcat()`, `strcpy()`, ...
`<unistd.h>`	Llamadas al sistema de UNIX `fork()`, `read()`, `write()`, `close()`, ...
`<fcntl.h>`	Control de ficheros `open()`, `creat()`, ...

(Ahora sí) printf

```
int printf(const char * restrict 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 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 (las veremos más tarde)... más info en `man 3 printf`.

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.

Switch

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

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

While/do-while

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

y también

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

- Se itera hasta que la expresión evalúa a 0.
- La sentencia `break` rompe un bucle (funciona con `for` también).
- La sentencia `continue` pasa a la siguiente iteración sin acabar la actual (funciona con `for` también).

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.

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 */`
- En una sentencia, el operador * (*dereference*) delante de una variable de tipo puntero significa que queremos operar sobre el contenido de la dirección a la que apunta.
- Con el operador & (*address of*) delante de una variable obtenemos su dirección de memoria.
- No se puede usar un puntero que no apunta a ningún sitio (NULL)

Aritmética de punteros

- Los punteros se pueden sumar, restar, etc. P.e. para conseguir el tamaño de un buffer.
- 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 */
```

Los *Arrays* en C

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

Los *Arrays* en C

- El operador `sizeof` sobre un array devuelve el tamaño de la memoria reservada (NO el número de elementos!).
- 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
```

Pasando direcciones de memoria como argumento

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

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

int
main(int argc, char *argv[])
{
    char c = 'b';

    dameletra(&c);
    printf("c es %c\n", c);
    exit(EXIT_SUCCESS);
}
```

Cadenas de caracteres (string)

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

- snprintf: similar a printf, pero imprime en una cadena. Escribe como mucho el número de bytes que se especifica en el segundo argumento, contando 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.

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 del programa que se invoca. Desde UNIX V7, siempre va terminado con un NULL.

Registros

```
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.

Registros

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

typedef struct Coordenada Coordenada; /* definición de tipo Coordenada */
Coordenada c = {13,31};               /* declaración e inicialización  */
```

- 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:

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

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

Programas con varios ficheros fuente

- Las variables globales que declara un fichero externo tiene que definirse como `extern`.
- Una variable tiene que estar declarada en un fichero fuente.
- Una función o variable global declarada como `static` no es visible desde otros ficheros. Si no se especifica, sí son visibles.
- Las variables, tipos de datos, constantes, etc. compartidas por los ficheros fuente de un programa deberían estar en un fichero de cabeceras.

Programas con varios ficheros fuente

- Cada fichero fuente debe incluir los ficheros de cabeceras que necesite. No es buena idea 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 entre los directorios del sistema)
- No se deben incluir dos veces un mismo fichero de cabecera.

gdb

El comando gdb es un depurador que nos permite:

- Inspeccionar un programa (p. ej. desensamblar).
- Inspeccionar un proceso (p. ej. ver los valores de la memoria).
- Inspeccionar un *core*: es la *foto* de un proceso en un fichero.
- En la mayoría de las ocasiones, **no es la forma más eficiente** de depurar un programa.

gdb

1. Arrancamos gdb con el ejecutable:
   `gdb ejecutable`
2. Ejecutamos dentro de gdb:
   `run argumento1 argumento2 ...`
3. ... fallo en ejecución ...
4. `bt` #vuelca la pila
5. `frame 3` #selecciono el registro de activación que deseo inspeccionar
6. `info locals` #veo el valor de las variables locales
7. `info args` #veo el valor de los argumentos
8. `whatis z` #veo el tipo de la variable z

gdb

Meter punto de ruptura y ejecutar paso a paso:

1. Arrancamos gdb con el ejecutable:
   `gdb ejecutable`
2. `break f1` #mete punto de ruptura en la función f1
3. Ejecutamos dentro de gdb:
   `run argumento1 argumento2 ...`
4. se para en f1, ahora podemos inspeccionar como en el ejemplo anterior.
5. `stepi` #ejecuta una instruccion
6. ...
7. `continue` # sigue ejecutando normalmente.

Análisis dinámico: Valgrind

- Ayuda para encontrar errores y problemas automáticamente
- Hace análisis dinámico del código en ejecución (instrumentado o sin instrumentar)
- Leaks, corrupción de memoria, patrones incorrectos de uso de los recursos
- `http://valgrind.org`
- Para asegurarte de que no hay leaks y ver dónde están: `valgrind --leak-check=full programa`

Análisis dinámico: Valgrind

```
$ valgrind ./a.out lespaul sg telecaster stratocaster
==6491== Memcheck, a memory error detector
==6491== Copyright (C) 2002-2017, and GNU GPL'd, by Julian Seward et al.
==6491== Using Valgrind-3.15.0 and LibVEX; rerun with -h for copyright info
==6491== Command: ./a.out lespaul sg telecaster stratocaster
==6491==
[1] lespaul
[2] sg
[3] telecaster
[4] stratocaster
The largest argument is: stratocaster
==6491==
==6491== HEAP SUMMARY:
==6491==     in use at exit: 19 bytes in 2 blocks
==6491==   total heap usage: 4 allocs, 2 frees, 1,056 bytes allocated
==6491==
==6491== LEAK SUMMARY:
==6491==    definitely lost: 19 bytes in 2 blocks
==6491==    indirectly lost: 0 bytes in 0 blocks
==6491==      possibly lost: 0 bytes in 0 blocks
==6491==    still reachable: 0 bytes in 0 blocks
==6491==         suppressed: 0 bytes in 0 blocks
==6491== Rerun with --leak-check=full to see details of leaked memory
==6491==
==6491== For lists of detected and suppressed errors, rerun with: -s
==6491== ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 0 from 0)
$
```