Sistemas Operativos

Universidad Complutense de Madrid

2020-2021

Revisión: programación en C

Juan Carlos Sáez

El lenguaje C

- Creado por Dennis Ritchie en 1972
  - Bell Laboratories - AT&T
  - SO UNIX fue reescrito en "C" (Ken Thompson, 1973)
- Usado ampliamente hoy en día para programación de sistemas:
  - Sistemas operativos, como GNU/Linux
  - Microcontroladores: automóviles y aviones
  - Dispositivos móviles/empotrados
  - DSPs (Digital Signal Processors): audio y video digital

El lenguaje C

- Permite programación de alto y bajo nivel
- Mayor rendimiento que lenguajes con mayor nivel de abstracción (Java, C++...)
  - Java/C++ ocultan muchos detalles necesarios para escribir código relacionado con el SO

- Es necesario asumir la responsabilidad de la gestión de memoria
  - No hay recolector de memoria
- Comprobación explícita de errores
- No hay valores de inicialización "por defecto"

Objetivos de esta introducción

- Introducir/Revisar algunos conceptos básicos de C
  - Los detalles se irán descubriendo con el uso
- Advertir sobre los fallos típicos de programación
  - Evitar pérdidas de tiempo en la realización de las prácticas

Ejemplo simple (I)

```c
#include <stdio.h>

int main(void)
{
    printf("Hello World.\n\t and you !\n");
    /* print out a message */
    return 0;
}
```

Output

```
Hello World.
        and you !
```

Ejemplo simple (II)

- `#include <stdio.h>`
  - Incluir fichero de cabecera `stdio.h`
  - No es necesario “;” al final
  - Sólo letras minúsculas
    - C es sensible a mayúsculas/minúsculas
- `int main(void) { ... }`
  - Código a ejecutar
- `printf(" /* message */ ");`
  - `'\n'` salto de línea
  - `'\t'` tabulador
  - `'\'` secuencia de escape

Tipos de datos simples

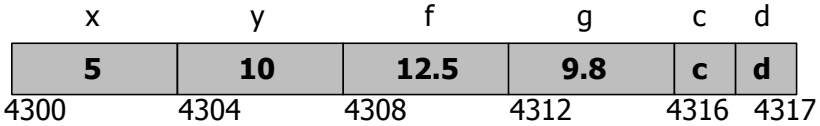
Tipo	Tamaño (bytes)	Rango de valores	Formato[1]
`int`	4	$[-2^{-31}, 2^{-31}-1]$	`%d`
`char`	1	$[-128, 127]$	`%c`
`float`	4	$3{,}4E+/-38$	`%f`
`double`	8	$1{,}7E+/-308$	`%lf`
`long`[2]	8	$[-2^{-63}, 2^{-63}-1]$	`%l`
`short`	2	$[-2^{15}, 2^{-15}-1]$	`%d`

[1] Hay varios modificadores de formato para cada tipo de datos (link)

[2] Un long ocupa lo mismo que una dirección de memoria (específico de plataforma). Usar `sizeof(long)` en un programa de C para obtener el tamaño.

Disposición de los datos

```
int x = 5, y = 10;
float f = 12.5, g = 9.8;
char c = 'c', d = 'd';
```



Otro ejemplo

```c
#include <stdio.h>

int main(void)
{
    int nstudents = 0; /* Initialization, required */
    printf("How many students does Cornell have ?:");
    scanf("%d", &nstudents); /* Read input */
    printf("Cornell has %d students.\n", nstudents);
    return 0;
}
```

Output

```
How many students does Cornell have ?: 20000 (enter)
Cornell has 20000 students.
```

Operadores y Sintaxis Básica

Operadores

- Aritméticos:
  - `int i = i+1; i++; i--; i *= 2;`
  - `+, -, *, /, %,`
- Relacionales y lógicos:
  - `<, >, <=, >=, ==, !=`
  - `&&, ||, &, |, !`

Sintaxis

- `if ( ) { } else { }`
- `while ( ) { }`
- `do { } while ( );`
- `for (i=1; i <= 100; i++) { }`
- `switch ( ) {case 1: … }`

Arrays de 1 dimensión

```c
#include <stdio.h>
int main(void)
{
    int number[12]; /* 12 cells, one cell per student */
    int index, sum = 0;

    /* Always initialize array before use */
    for (index = 0; index < 12; index++) {
        number[index] = index;
    }
    /* now, number[index]=index; will cause error:why ?*/

    for (index = 0; index < 12; index = index + 1) {
        sum += number[index]; /* sum array elements */
    }
    return 0;
}
```

Más sobre arrays ... (I)

Cadenas de caracteres

```
char message[6]={'h','e','l','l','o','\0'}; /* '\0'= end */
printf("%s", message); /* print until '\0' */
```

- Inicialización equivalente:

```
char message[] = "hello";
```

- Funciones de manejo de cadenas(<string.h>)
  - strlen, strcpy, strncpy, sprintf, strcmp, strncmp, strcat, strncat, strstr, strchr

Operaciones sobre cadenas de caracteres

- =, == y + están sobrecargados para *strings* de C++ pero no para cadenas de C
- Comparación de cadenas

Incorrecto

```
char* string1=...;
char string2[]=...;

if (string1==string2)
    printf("Equal addresses\n");
```

Correcto

```
char* string1=...;
char string2[]=...;

if (strcmp(string1,string2)==0)
    printf("Equal strings\n");
```

- Copia de Cadenas

Incorrecto

```
char* old="Cool string";
char* copy;

copy=old;
/* copy now points to old */
```

Correcto

```
char* old="Cool string";
char* copy;
copy=malloc(strlen(old)+1);
strcpy(copy,old);
/* Don't forget to free up
   the pointer later */
```

Más sobre arrays ... (II)

Arrays multi-dimensionales

```
int points[3][4]; /* NOT points[3,4] */
points [1][3] = 12;
printf("%d", points[1][3]);
```

Más sobre arrays ... (III)

Conclusiones

- ¡Es necesario inicializarlos antes de usarlos!:

```
int number[12];
printf("%d", number[20]); /* ¡¡esto está permitido!! */
```

- Las cadenas de caracteres deben acabar con el caracter '\0':
  - Si no, las funciones estándar de manejo de cadenas fallarán

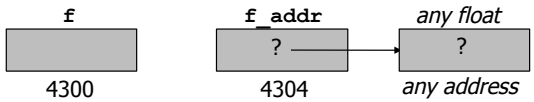
```
char message[6]={'h','e','l','l','o','\0'}; /* '\0'= end */
printf("%s", message); /* print until '\0' */
```

- Usar `strcmp()`, `strcpy()` y `strcat()` para comparar, copiar y concatenar cadenas, respectivamente.

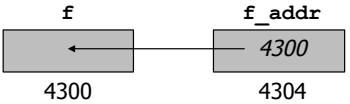
Punteros

- Puntero: tipo de datos de C para almacenar una dirección de memoria
  - Una variable de tipo puntero puede almacenar la dirección de cualquier otra variable
  - … o cualquier dirección, aunque no estemos *autorizados* para acceder a ella

```
float f;          /* data variable */
float *f_addr;    /* pointer variable */
```




```
f_addr = &f;      /* '&' address operator */
```



Punteros: Ejemplo

```c
#include <stdio.h>

int main(void) {
    int j;
    int *ptr;

    ptr=&j;   /* initialize ptr before using it */
              /* *ptr=4 does NOT initialize ptr */
    *ptr=4;   /* j <- 4 */
    j=*ptr;   /* j <- MEM[dir(ptr)] */
    return 0;
}
```

Gestión de memoria en C

- Dos fuentes de asignación dinámica de memoria:
  1. **Pila** (gestión implícita)
     - Variables locales
     - Almacenar parámetros, dirección de retorno, punteros a marcos de activación, etc.
  2. **Heap** (gestión explícita)
     - Para reservar grandes cantidades de memoria
     - Reserva de memoria que se referencia desde función distinta a la que reservó la memoria

Asignación dinámica de memoria

- El programador debe reservar/liberar memoria explícitamente del *heap*

```
#include <stdio.h>
void my_function(void) {
    char c;
    int *ptr;

    /* allocate space to hold an int */
    ptr = malloc(sizeof(int));

    /* do stuff with the space */
    *ptr=4;

    /* free up the allocated space */
    free(ptr);
}
```

Gestión de errores

- A diferencia de Java, no existen “excepciones”
- Es necesario realizar la comprobación de errores de manera manual
  - Siempre que se use una función que no se haya escrito
  - Los errores pueden aparecer en cualquier sitio
    - Cuidado con el manejo de punteros y cadenas de caracteres!!

Funciones

- ¿Cuando usarlas?
  - Programa demasiado largo
  - Para facilitar:
    - Programación
    - Depuración
    - Reutilización de código
- ¿Cómo usarlas?
  - Paso de parámetros
    - Por valor
    - Por referencia
  - Valores de retorno
    - Por valor
    - Por referencia

Parámetros por valor

```c
#include <stdio.h>

/* function prototype at start of file */
int sum(int a, int b);

void main(void){
    int total = sum(4,5); /* call to the function */
    printf("The sum of 4 and 5 is %d", total);
}

int sum(int a, int b){ /* arguments passed by value*/
  return (a+b);       /* return by value */
}
```

Parámetros por referencia

```c
#include <stdio.h>

/* function prototype at start of file */
int sum(int *pa, int *pb);

void main(void){
    int a=4, b=5;
    int *ptr = &b;
    int total = sum(&a,ptr); /* call to the function */
    printf("The sum of 4 and 5 is %d", total);
}

int sum(int *pa, int *pb){
                /* args passed by reference */
return (*pa+*pb);   /* return by value */
}
```

¿Por qué se usan punteros? (I)


```c
#include <stdio.h>

void swap(int, int);

void main() {
    int num1 = 5, num2 = 10;
    swap(num1, num2);
    printf("num1 = %d and num2 = %d\n", num1,
        num2);
}

void swap(int n1, int n2) { /* passed by value */
    int temp;
    temp = n1;
    n1 = n2;
    n2 = temp;
}
```

¡¡Código incorrecto!!

¿Por qué se usan punteros? (II)

```c
#include <stdio.h>

void swap(int *, int *);

void main() {
int num1 = 5, num2 = 10;
    swap(&num1, &num2);
    printf("num1 = %d and num2 = %d\n", num1, num2);
}

void swap(int *n1, int *n2) { /* passed and returned by reference */
int temp;
    temp = *n1;
    *n1 = *n2;
    *n2 = temp;
}
```

¿Por qué es incorrecto este ejemplo?

```c
#include <stdio.h>

void dosomething(int *ptr);

void main() {
    int *p;
    dosomething(p);
    printf("%d", *p); /* will this work ? */
}

void dosomething(int *ptr){
   /* ptr passed and returned by reference */
    int temp=32+12;
    *ptr = temp;
}
```

Error en tiempo de ejecución

Solución 1

```c
#include <stdio.h>

void dosomething(int *ptr);

void main() {
    int a;
    int *p=&a;
    dosomething(p);
    printf("%d", *p);  /* will this work ? */
}

void dosomething(int *ptr){
     /* ptr passed and returned by reference */
    int temp=32+12;
    *ptr = temp;
}
```

Solución 2

```c
#include <stdio.h>

void dosomething(int *ptr);

void main() {
    int *p=malloc(sizeof(int));
    dosomething(p);
    printf("%d", *p);  /* will this work ? */
    free(p);
}

void dosomething(int *ptr){
     /* ptr passed and returned by reference */
    int temp=32+12;
    *ptr = temp;
}
```

Paso de arrays como parámetro


```c
#include <stdio.h>

void init_array(int array[], int size);

void main(void) {
    int i,list[5];
    init_array(list, 5);
    for (i = 0; i < 5; i++)
        printf("next: %d", list[i]);
}

void init_array(int array[], int size) { /* why size ? */
    /* arrays ALWAYS passed by reference */
    int i;
    for (i = 0; i < size; i++)
        array[i] = 0;
}
```

Prototipo alternativo

```c
void init_array(int* array, int size);
```

Estructuras

- La abstracción de C más similar a las clases Java/C++
  - .. pero solo con atributos (campos), no métodos

```
#include <stdio.h>
struct birthday{
    int month;
    int day;
    int year;
};

void main() {
    struct birthday mybday;  /* - no 'new' needed ! */

    mybday.day=1; mybday.month=1; mybday.year=1977;
    printf("I was born on %d/%d/%d\n",
           mybday.day,mybday.month,mybday.year);
}
```

Estructuras : “.” vs “->”

```
#include <stdio.h>

struct birthday{
    int month;
    int day;
    int year;
};

void main() {
    struct birthday mybday;
    struct birthday* ptrMybday=&mybday;

    ptrMybday->day=1; ptrMybday->month=1; ptrMybday->year=1977;
    printf"(I was born on %d/%d/%d\"n,
           mybday.day,mybday.month,mybday.year);
}
```

Estructuras como parámetros

```c
/* pass struct by value */
void display_year_1(struct birthday mybday) {
    printf("I was born in %d\n", mybday.year);
}               /* - inefficient: why ? */

/* pass struct by reference */
void display_year_2(struct birthday *pmybday) {
    printf("I was born in %d\n", pmybday->year);
    /* warning ! ‘’->, not ‘’., after a struct pointer*/
}

/* return struct by value */
struct birthday get_bday(void){
    struct birthday newbday;
    newbday.year=1971; /* ‘’. after a struct */
    return newbday;
}
```

typedef: alias de tipos

- Mayor claridad y facilidad de uso

sintaxis

```
typedef <tipo_existente> <nombre_alias_de_tipo>;
```

```
typedef int Employees;
Employees my_company;   /* same as int my_company; */

typedef struct person Person;
Person me;              /* same as struct person me; */

typedef struct person *Personptr;
Personptr ptrtome; /* same as struct person *ptrtome;*/
```

Más sobre punteros ...

```
int month[12];
/* month is a pointer to base address 430*/

int *ptr = month + 2;
/* mem(ptr + k) = mem(ptr) + k*sizeof(data_type)
ptr points to month[2], => ptr is now (430+2*4)= 438 */

month[3] = 7;
/* month address + 3 * int_size=> int at (430+3*4) is 7 */

ptr[5] = 12;
/* int at (438+5*4) is now 12. Thus, month[7]=12 */

ptr++;
/* ptr <- 438 + 1 * sizeof(int) = 442 */

(ptr + 4)[2] = 12;
/* accessing ptr[6] i.e., month[9] */
```

Cadenas de caracteres ...

```
#include <stdio.h>
void main() {

char msg[10];          /* array of 10 chars */
char *p;                     /* pointer to a char */
char msg2[]="Hello"; /* msg2 = 'H''e''l''l''o''\0' */

msg = "Bonjour"; /* ERROR. msg has a const address.*/
p   = "Bonjour"; /* address of ""Bonjour goes into p */
msg = p;  /* ERROR. Message has a constant address. */

p = msg;                     /* OK */

p[0] = 'H', p[1] = 'i',p[2]='\0'; /* *p and msg are now "Hi" */
}
```

Parámetros argc y argv


```c
#include <stdio.h>

/* program called with cmd line parameters */
int main(int argc, char *argv[]) {
    int ctr;

    for (ctr = 0; ctr < argc; ctr = ctr + 1) {
        printf("Argument #%d->|%s|\n", ctr, argv[ctr]);
   }
        /* ex., argv[0] == the name of the program */
    return 0;
}
```

Punteros a función

- Ventaja fundamental: flexibilidad
  - Permiten modelar "interfaces en C"

```
/* function returning integer */
int func(void);

/* function returning pointer to integer */
int *func(int a);

/* pointer to function returning integer */
int (*func)(void);

/* pointer to func returning ptr to int */
int *(*func)(int);
```

Punteros a función: ejemplo

```c
#include <stdio.h>
void myproc (int d);
void mycaller(void (* f)(int), int param);

void main(void) {
    myproc(10);         /* call myproc with parameter 10*/
    mycaller(myproc,10);    /* and do the same again ! */
}

void mycaller(void (* f)(int), int param){
    (*f)(param);         /* call function *f with param */
}

void myproc (int d){
    . . .                  /* do something with d */
}
```

Programas con varios ficheros fuente

main.c

```c
#include "mypgm.h"

void main(void)
{
    myproc();
}
```

mypgm.c

```c
#include <stdio.h>
#include "mypgm.h"

int mydata=0;
void myproc(void)
{
    mydata=2;
    . . . /* some code */
}
```

mypgm.h

```c
void myproc(void);
extern int mydata;
```

Declaraciones externas

```
#include <stdio.h>

extern char user2line [20]; /* globally defined in another file*/
char user1line[30];         /* global for this file */

void dummy(void);

void main(void) {
    char user1line[20];     /* different from earlier */
    . . .                    /* restricted to this func */
}

void dummy(){
    extern char user1line[]; /* the global user1line[30] */
    . . .
}
```

Por último ...

- SIEMPRE inicializar las variables antes de usarlas (especialmente los punteros)
- Solicitar memoria para las estructuras de datos dinámicas
- No devolver punteros a variables locales de la función
- No utilizar punteros después de liberarlos con `free()`
- Añadir código de tratamiento de errores en los programas
  - ¡¡Esto simplifica la depuración!!
- Un array es también un puntero, pero su valor es inmutable
- Prestad atención a los ejemplos proporcionados en cada práctica