Introducción al lenguaje C

Javier Gatón Herguedas,
Pablo Martínez López,
Manuel de Castro Caballero

GUI
Grupo Universitario de Informática
Escuela de Ingeniería Informática, Universidad de Valladolid

Hour of Code, 2020
25 de Noviembre de 2020

Origen de C

- 1967, Martin Richards · BCPL. Diseño de Sistemas Operativos.
- 1970, Ken Tompson · Primera versión de UNIX. Usa lenguaje B.
- 1972, Denis Ritchie · Lenguaje C.
  - Lenguaje de alto nivel · independiente de máquina.
  - Con instrucciones sencillas, próximas al código máquina.
- Muchas compañías implementan su propio C por lo que surgen discrepancias.
- 1983, ANSI estableció un comité para crear una definición no ambigua del lenguaje C e independiente de la máquina, que pudiera utilizarse en todos los tipos de C.

Figura: Logo de C

¿Por qué C?

- Es rápido, control muy directo de la máquina.
- **Ayuda a comprender todo lo que está sucediendo en la máquina.**
- La mayoría de las partes de los sistemas operativos están escritos en C.
- Los drivers están escritos en C muchas veces.
- C es la base de otros lenguajes (Python, Rust).

Sintaxis-Hola Mundo!

```c
#include <stdio.h>
int main(int argc, char *argv[])
{
      printf("Hola mundo");
      return 0;
}
```

Compilar y Ejecutar

Para poder ejecutar nuestro programa utilizamos un compilador.

Compilación

gcc $< nombrePrograma > -Wall -o < nombreEjecutable >$

Tras esto, pasamos a la ejecución:

Ejecución

$./ < nombreEjecutable >< argumentos >$

¿Qué es una variable?

- Valor contenido en posición de memoria.
- Los tipos básicos más utilizados
  - int/short/long
  - float/double
  - char
  - No hay booleans. 0=false, 1=true.
  - Strings (Son arrays de chars, ya lo veremos)
- Comentarios: //[...] o multilinea /* [...] */

printf

Tipo	Codificación
Caracter	%c
Entero decimal	%d
Entero largo (long int)	%ld
Entero largo largo (long long)	%Ld
Flotante se representa con exponente	%e
Flotante se representa sin exponente	%f
double	%lf
long double	%Lf
Entero octal, sin el cero inicial	%o
Entero decimal sin signo	%u
Entero representado en hexadecimal sin 0x	%x
Strings	%s

¡Cuidado con la precisión de los floats!

Entrada y salida

Programa que toma un parámetro de entrada y lo imprime por pantalla.

```
#include <stdio.h>
int main(int argc, char *argv[])
{
      double a;
      char cadena[80];
      scanf("%d",&a);
      scanf("%s",&s);
      printf("El double es %d, y la cadena es %s\n",a,cadena);
}
```

Estructuras condicionales if/if-else

Podemos utilizar estructuras condicionales para controlar la ejecución nuestro código. Esto nos permite que un fragmento de código se ejecute si se dan unas condiciones impuestas por nosotros mismos.

Estructuras condicionales if/if-else

```
int condicion = 1;
int condicion2 = 1;
int value;
if (condicion){
    value = 23;
}else if(condicion2){
    value = 33;
}else{
    value = 43;
}
printf("%d\n",value);
```

La estructura *switch*

- Un *Switch* es otra estructura de control, que nos permite gestionar en varios casos lo que queremos comprobar.
- El último caso deberá ser *default* y este se ejecutará siempre que el resto de condiciones hayan fallado.
- Además, después de cada caso tenemos que poner un *break* para salir del *switch*.

switch código

```
[...]
int number;
scanf("%d", &number);
switch (number){
    case 1:
        printf("Nmero 1");
        break;
    case 2:
        printf("Nmero 2");
        break;
    default:
        printf("Otro nmero");
        break;
}
[...]
```

Estructuras *while* y *do-while*

El bucle while nos permite ejecutar un bucle mientras una condición sea cierta. También existe la variante do-while.

```
int n = 10;
while(n > 10)
{
  printf("%d\n", n);
  n--;
}
```

```
int n = 10;
do {
  printf("%d\n", n);
  n--;
} while (n > 0);
```

Estructura *for*

El bucle for es un bucle controlado por un iterador, resultando que antes de comenzar ya sabemos el límite de ejecuciones que tiene.

```
for (int i = 0; i < value; i++)
{
    printf("%d\n",i);
}
```

Métodos

Los métodos son fragmentos de código destinados a realizar una tarea concreta. Las funciones deben escribirse antes de ser usadas. En caso contrario se debe utilizar un prototipo al inicio.

```
int multiplicar(int a, int b);
[...]
main(...)
[...]
int multiplicar(int a, int b)
{
  return a*b;
}
```

Arrays

- Un array puede compararse con un vector. Es una asignación de celdas en las que podemos guardar informacion de un tipo concreto.
- Los arrays tienen tipo, y deben declararse e inicializarse como las variables. También podemos hacer operaciones con lo que hay guardado en sus celdas. tipo nombre[N];
- Se pueden declarar arrays de n dimensiones, pero en el caso que nos ocupa sólo vamos a ver los de 1 dimensión.
- Visto esto, un String es un array de *chars* que finaliza con el caracter $\backslash 0$.

Ejercicio de repaso

Escribid un programa que tome un número por teclado. Dicho número representará el tamaño de un vector de números enteros que tendréis que rellenar por valores recogidos por teclado.

Tras ello, se debe crear un método que tome dicho vector y devuelva la suma de sus elementos. Imprimid dicho valor por pantalla. Podéis elegir utilizar o no un prototipo.

Recordad compilar con la opción -Wall.

Punteros y direcciones de memoria

- Las variables tienen el valor del contenido de una dirección de memoria.
- Un puntero es una variable entera sin signo que almacena la dirección de memoria de otra variable o dato.
- Un puntero se define con un asterisco (*) antes del nombre de la variable (int *a).
- De la misma forma se utiliza un asterisco para obtener la variable apuntada por un puntero (int b = *a) (operador de indirección/de-referencia).
- Con el símbolo *ampersand* (&) obtenemos el puntero que apunta a la variable (int *a = &b) (operador de dirección).
- Si asignamos una variable el valor de otra (int b = a), tendremos dos variables y "un solo puntero".

Operaciones con punteros

```
[...]
float x[3] = {1.1, 2.2, 3.3};
float *punt = x;

printf ("Direccion de memoria del primer elemento de x: %x y
    del segundo: %x\n",x,x+1);

printf ("Direccion de memoria del primer elemento de x: %x y
    del segundo: %x\n",punt,(punt+1));

printf ("Valor almacenado en la direccion de memoria %x: %f y
    en %x: %f\n",x,*x,x+1,*(x+1));

printf ("Valor almacenado en la direccion de memoria %x: %f y
    en %x: %f\n",x,x[0],x+1,x[1]);
[...]
```

Malloc() y calloc()

- malloc() permite reservar memoria de manera dinámica. calloc() por su parte reserva memoria y la inicializa toda a 0.
- La diferencia entre usar malloc y reservar una cantidad variable de memoria en la declaración del array[x], es que en el segundo caso si x es una variable se almacena en la pila, donde es posible alcanzar algún máximo del sistema operativo.
- Un malloc siempre reseva memoria en ejecución, pero si en array[x] 'x' es una constante, como array[5], se reserva en compilación, siendo esto más rápido.
- Está en la librería <stdlib.h>
- char *str = (char *)malloc(15);

Estructuras

- Una estructura es una variable diseñada por nosotros mismos, la cual contiene varios atributos.
- Se deben definir al principio del programa.
- No son objetos
- Para acceder al campo de una estructura: $estructura.atributo$
- Para acceder desde un puntero: $puntero- > atributo$

Ejemplo de struct

```
[...]
struct Persona{
  char nombre[30];
  int edad;
}; //Importante, no olvidar este punto y coma
[...]
struct Persona persona;
strcpy(persona.nombre, "Pepe");
persona.edad=30;
printf("Nombre: %s\n", persona.nombre);
printf("Edad: %d\n", persona.edad);
[...]
```

Definición de variables struct en el inicio

```
[...]
struct Persona{
  char nombre[30];
  int edad;
} persona;
[...]
strcpy(persona.nombre, "Pepe");
persona.edad=30;
printf("Nombre: %s\n", persona.nombre);
printf("Edad: %d\n", persona.edad);
[...]
```

Definición de struct por typedef

```
[...]
typedef struct{
  char nombre[30];
  int edad;
} Persona;
[...]
Persona persona;
strcpy(persona.nombre, "Pepe");
persona.edad=30;
printf("Nombre: %s\n", persona.nombre);
printf("Edad: %d\n", persona.edad);
[...]
```

Uniones

- Tipo especial que almacena diferentes tipos de datos en el mismo espacio de memoria (no a la vez)
- Tipo de dato "mutable"
- Utilidad avanzada para pseudopolimorfismo, eficiencia de memoria.
- Se declaran como los structs, pero solo está en memoria el último dato introducido.

```
union Uni {
    int i;
    float f;
};
[...]
union Uni dato;
dato.i = 13;
dato.f = 250.5;
printf( "dato.i : %d\n", dato.i); // Imprime 1132101632
printf( "dato.f : %f\n", dato.f); // Imprime 250.5
[...]
```

Preprocesador

- Los DEFINES se utilizan para sustituir un fragmento de código que se repite. Su valor se sustituye en compilación.
- Se suelen utilizar para representar constantes o pequeñas funciones.
- Se especifican al comienzo del programa.

```
[...]
#define max(A,B) ( (A)>(B) ? (A) : (B) )
#define NUMBER 1024
[...]
int x=max(1025,NUMBER);
[...]
```

Podemos aumentar la eficiencia del programa compilando con la opción: **-O3**

Ejercicio de repaso

Escribid un programa que utilize una estructura que defina a las personas, concretamente, su nombre, dígitos del DNI y letra del DNI. Podéis elegir declararla con o sin typedef. Rellenad sus campos e imprimidlos. Para rellenar al menos uno de los tres campos deberéis de utilizar el preprocesador.

También deberéis declarar un vector de cinco elementos. Utilizad un puntero para recorrer el vector e imprimir sus valores.

Recordad compilar con la opción -Wall.

Gracias por vuestra atención

Esperamos que el taller os haya resultado útil, si tenéis alguna cuestión, no dudéis en preguntar.

Repositorio con código visto en el taller:
https://github.com/HylianPablo/TallerC_HoC2020

Contacto en Telegram: **@HylianPablo @javgat @bomilk**

Enlaces de interés: **https://infor.uva.es/~cevp/introduccion_C/**