

Programación Imperativa

Lenguaje C

Algoritmos y Estructuras de Datos I

Martín Ariel Domínguez

Lenguaje C

- Creado en 1972 por Dennis Ritchie
- Es un lenguaje imperativo.
- Es destacado por su nivel de eficiencia.
- Es débilmente tipado, de medio nivel.
- Linux está implementado en C.
- Es “Case Sensitive”.
- Se adopta el estándar (ISO/IEC 9899:1990) que lo hace multiplataforma (portable).

Lenguaje C - Introducción

- En haskell para ejecutar un programa:
 - Interpretado (ghci)
 - Compilado (ghc)
- Para el lenguaje C, sólo se puede compilar un programa para ejecutarlo, utilizando el comando de consola: gcc

```
$>gcc -Wall -Wextra -std=c99 hola.c -o holaPrograma
```

```
$>./holaPrograma
```

Así se ejecuta luego de compilar un programa en C

Escribiendo mi primer programa C

- En el lenguaje C, un programa se escribe a partir de una **función principal** y de funciones secundarias. Ésta **función principal**, se denomina `main()` y debe estar siempre presente, además, es la primera en ser llamada.
- Para realizar un programa en C, lo primero que se debe hacer es crear el programa fuente (con extensión ".c") de la siguiente manera:

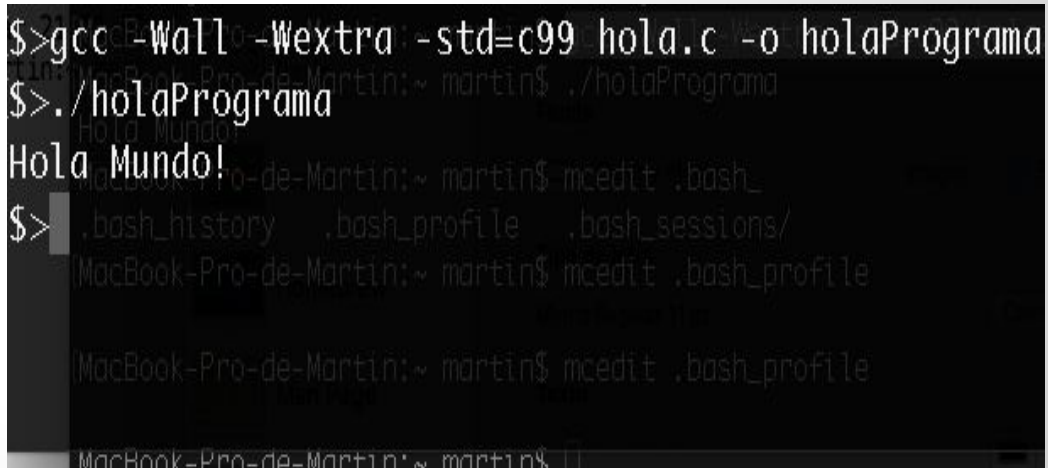
Primer programa en C

Edición del archivo fuente:

Archivo fuente: **hola.c**

```
#include <stdio.h>
/*inclusión biblioteca
estandar */
int main(void)
/* main devuelve un int, no
toma argumentos */
{
    printf("Hola Mundo!\n");
    return 0; /* resultado
de la función */
}
```

Compilación y ejecución

A terminal window with a dark background and light-colored text. The prompt is '\$>'. The first command is 'gcc -Wall -Wextra -std=c99 hola.c -o holaPrograma', which is followed by a new line. The second command is './holaPrograma', which outputs 'Hola Mundo!'. The prompt then changes to 'MacBook-Pro-de-Martin:~ martin\$'. The user then enters 'mcedit .bash_'. The prompt changes to 'MacBook-Pro-de-Martin:~ martin\$ mcedit .bash_'. The user then enters 'history', which lists several files including '.bash_history', '.bash_profile', and '.bash_sessions/'. The prompt changes to 'MacBook-Pro-de-Martin:~ martin\$ mcedit .bash_profile'. The user then enters 'mcedit .bash_profile' again. The prompt changes to 'MacBook-Pro-de-Martin:~ martin\$ mcedit .bash_profile'. The user then enters 'mcedit .bash_profile' again. The prompt changes to 'MacBook-Pro-de-Martin:~ martin\$'.

```
$>gcc -Wall -Wextra -std=c99 hola.c -o holaPrograma
$>./holaPrograma
Hola Mundo!
MacBook-Pro-de-Martin:~ martin$ mcedit .bash_
$> .bash_history .bash_profile .bash_sessions/
MacBook-Pro-de-Martin:~ martin$ mcedit .bash_profile
MacBook-Pro-de-Martin:~ martin$ mcedit .bash_profile
MacBook-Pro-de-Martin:~ martin$ mcedit .bash_profile
MacBook-Pro-de-Martin:~ martin$
```

Compilando

```
gcc -Wall -Wextra -std=c99 archivo.c -o ejecutable
```

- **gcc**: Compilador de C
 - **GNU Compiler Collection**: compilador libre del proyecto GNU
 - las opciones que vienen después se llaman "flags":
 - **-Wall**: Chequea todas las advertencias
 - **-Wextra**: chequeos extras
 - **-std=c99**: chequea que se programe de acuerdo al estándar del 1999
 - **archivo.c**: el (o los) archivos de entrada:
 - **-o NOMBRE_EJECUTABLE: (ejecutable)** Nombre del archivo ejecutable (a.out por defecto)

Tipos básicos en C

Los tipos básicos son:

- `char`, carácter
- `int`, entero
- `float`, real de simple precisión
- `double`, real de doble precisión
- ¿Los Booleanos? No hay booleanos nativos :-(
 - Para poder utilizar booleanos usamos la librería `<stdbool.h>` :
 - simplemente es un **renombramiento de tipos**, algo similar a lo que hacíamos en Haskell con `type`.

Instrucciones Básicas de C

Declaración de variables

Formalismo	Lenguaje C
[var x,y:Num]	<code>int x, y;</code>

Instrucciones Básicas de C

Asignación

Formalismo	Lenguaje C
<code>x := 10</code>	<code>x = 10;</code>

Instrucciones Básicas de C

Alternativas

Formalismo	Lenguaje C
<pre>if (x<y) → b:=True □ (x≥y) → b:=False fi</pre>	<pre>if (x<y) { b=true; } else { b=false; }</pre>

Instrucciones Básicas de C

Ciclo

Formalismo	Lenguaje C
do $(x \geq y) \rightarrow$ $x := x - y$ $i := i + 1$ od	while $(x \geq y) \{$ $x = x - y;$ $i = i + 1;$ $\}$

Entrada Salida: `printf`

- Como pudimos ver en el primer programa para poder imprimir por pantalla “Hola Mundo” se utiliza:

```
printf("Hola Mundo!\n");
```

- Pero además, puedo utilizar `printf` para imprimir texto mezclado con variables, por ejemplo:

instrucción	Estado de variables	Se imprime en pantalla
<code>printf("a*b=%d\n", c);</code>	$\{ \dots, c \mapsto 3, \dots \}$	<code>a*b=3</code>
<code>printf("3*x=%d\n", x*3);</code>	$\{ \dots, x \mapsto 5, \dots \}$	<code>3*x=15</code>

Entrada Salida: `printf`

- Los caracteres , “\” **barra invertida** y `n` juntos, provocan un salto de línea (return) en pantalla.
- Puedo poner tantos `%d` como sea necesario, si luego, pongo la misma cantidad de expresiones para que se inserten en el texto.
- De manera general el `printf` se utiliza como sigue:

```
printf(cadena_de_control, lista_de_arg) ;
```

Entrada Salida: `scanf`

- Esta es la función para entrada estándar, se utiliza de un modo parecido a la anterior.

```
scanf ("%d" , &x) ;
```

- `%d` indica que se lee con formato entero. En el ejemplo el caracter `"&"` se usa para la función `scanf` pueda guardar dentro de la variable `x` el valor entero ingresado por pantalla".

Entrada Salida: `scanf`

- En general al solicitar una variable entera al usuario, se utilizan combinadas `printf` y `scanf` como sigue:

```
printf("Ingrese un valor para x\n");  
scanf("%d", &x);
```

- De lo contrario el usuario no se entera que es lo que se espera de él :-).

Traducción del Formalismo a C

```
[var x,y,i:Int]
i:=0;
do not (x<y) ->
    x:=x-y;
    i:=i+1;
od
```

Solicito las
variables **x**
e **y** al
usuario

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

```
    int x, y, i;
```

```
    printf("Ingrese un valor para x\n");
    scanf("%d", &x);
    printf("Ingrese un valor para y\n");
    scanf("%d", &y);
```

```
    i = 0;
```

```
    while (x>=y) {
```

```
        x = x - y;
```

```
        i = i + 1;
```

```
    }
```

```
    printf("El resultado es %d\n", i);
```

```
    return 0;
```

```
}
```


Operadores básicos

Aritméticos

- suma: $a + b$
- resta: $c - d$
- multiplicación: $e * f$
- división: g / h
- módulo/resto: $i \% 14$

Lógicos

- and: $w \&\& y$
- or: $w || y$
- negación: $! z$

Relacionales/Comparación

- mayor: $j > k$
- mayor o igual: $ll \geq m$
- menor: $n < op$
- menor o igual: $q \leq r$
- igual: $s == t$ (es doble signo ==, no confundir con = asignación)
- distinto: $u != v$

Leer y escribir booleanos

```
#include <stdbool.h> /* biblioteca estándar de booleanos*/  
.... Código ....  
int temp;  
bool x; /* declaración de variables*/  
x = true; /*true es una constante definida como 1*/  
x = false; /* es una constante definida como 0 */  
scanf("%d", &temp); /* en realidad pedimos un entero por  
teclado, no un booleano, usamos temp, ya que scanf no tiene  
definido un '%' para bool */  
x = temp; /* asignamos para que el valor quede en x */  
printf("x vale %d\n", x); /* en realidad imprimimos un entero,  
no un booleano, */  
.... Sigue Código ....
```

Leer y escribir booleanos

- Notar que al pedir un booleano, en realidad se pide un entero, y si se ingresa un valor **distinto de cero**, será interpretado como `true` y si es **igual a cero** se interpretará como `false`.

GDB

A veces, leer el código fuente no es suficiente. Hay herramientas para ayudarnos a entender un programa y arreglar los errores posibles:

- poner `printf` en todos lados (puede ser tedioso)
- usar un debugger como GDB (más general)

GDB

- Para usar GDB, pasar el flag **-g** a gcc:

```
$>gcc -Wall -Wextra -std=c99 -g main.c -o ejecmain
```

- Eso le agrega al ejecutable información para debuggear (depurar) el programa (código fuente)

Para correr el programa:

```
$>gdb ./ejecmain
```

- **gdb** es una interfaz interactiva, donde se ingresan comandos:

GDB



Se puede escribir con ese atajo de

<code>help COMANDO</code>	<i>explicar lo que hace COMANDO</i>
<code>list</code>	<i>listar código</i>
<code>break NUM_LINEA</code>	<i>poner un breakpoint en la línea NUM_LINEA</i>
<code>continue</code>	<i>seguir ejecutando hasta el próximo breakpoint</i>
<code>run</code>	<i>empezar ejecución del programa</i>
<code>step</code>	<i>ejecutar una línea de código</i>
<code>next</code>	<i>ejecutar una línea de código (sin entrar a funciones)</i>
<code>print VARIABLE</code>	<i>imprimir el valor de VARIABLE</i>
<code>display VARIABLE</code>	<i>imprimir el valor de VARIABLE cada vez que para la ejecución</i>
<code>q</code> o <code>CTRL+D</code>	<i>salir</i>

GDB: Ejemplo de uso

```
$>gcc -Wall -Werror -std=c99 -g proy3.c -o proy3_ejecutable
```

```
$>gdb proy3_ejecutable
```

carga gdb desde la terminal y nos espera un cursor (gdb).

un ENTER, sin comando, repite el último comando. =D funciona TAB para autocompletar, las flechas para retomar comandos anteriores.

```
(gdb) help [comando]
```

está bueno empezar pidiendo ayuda.

```
(gdb) file proy3
```

Quiero ver el programa

GDB: Ejemplo de uso

```
(gdb) l
1  #include <stdio.h>
2
3  int main(void)
4  {
5      int i=0,x=7,y=3;
6      while (x>=y){
7          x=x-y;
8          i=i+1;
9      }
10     return 0;
11 }
```


GDB: Ejemplo de uso

- Voy a poner un breakpoint, para luego correr el programa y ver el valor de las variables.

```
(gdb) b 7
```

```
Breakpoint 1 at 0x100000f4a: file ejemplo GDB.c,  
line 7.
```

GDB: Ejemplo de uso

```
(gdb) run
```

```
Starting program: ejemplo3.c
```

```
Breakpoint 1, main () at ejemplo GDB.c:9
```

```
7          x=x-y;
```

- Aquí ya corrimos el programa y se detuvo la ejecución en la línea 7 que pusimos el breakpoint.
- Ahora queremos ver el valor de las variables, podemos usar el comando `display` para verlas cada vez que avanzamos en la ejecución.

GDB: Ejemplo de uso

```
(gdb) display x
```

```
1: x = 4
```

```
(gdb) display y
```

```
2: y = 3
```

```
(gdb) display i
```

```
3: i = 0
```

```
(gdb)
```

- Ahora en cada avance voy a ver el estado de las variables.

```
(gdb) n
```

```
8          i=i+1;  Línea en la que está detenido
```

```
1: x = 4
```

```
2: y = 3
```

```
3: i = 1
```

**Estado de
las
variables**

GDB: Ejemplo de uso

```
(gdb) n
```

```
9      }
```

```
1: x = 4
```

```
2: y = 3
```

```
3: i = 1
```

```
(gdb) n
```

```
6      while (x>=y){
```

```
1: x = 4
```

```
2: y = 3
```

```
3: i = 1
```

```
(gdb)
```

```
(gdb) n
```

```
9      }
```

```
1: x = 4
```

```
2: y = 3
```

```
3: i = 1
```

```
(gdb) n
```

```
6      while (x>=y){
```

```
1: x = 4
```

```
2: y = 3
```

```
3: i = 1
```

```
(gdb)n
```

GDB: Ejemplo de uso

```
Breakpoint 1, main ()  
at ejemplo GDB.c:7
```

```
7          x=x-y;
```

```
1: x = 4
```

```
2: y = 3
```

```
3: i = 1
```

```
(gdb)
```

```
8          i=i+1;
```

```
1: x = 1
```

```
2: y = 3
```

```
3: i = 1
```

```
(gdb)n
```

```
9          }
```

```
1: x = 1
```

```
2: y = 3
```

```
3: i = 2
```

```
(gdb)n
```

```
6          while (x>=y){
```

```
1: x = 1
```

```
2: y = 3
```

```
3: i = 2
```

```
(gdb) n
```

```
10         return 0;
```

```
...
```

¿Solo te gustan las ventanas?

- Estás de suerte :-). Podés usar un programa para hacer debugging visual y es cómodo para programar en C.
- Estos programas se denominan usualmente IDE (Integrated Development Environment), que significa Entorno de Desarrollo Integrado.
- Te presentamos uno que está disponible en el laboratorio:

[Presentación de Visual Code](#)

Funciones

- Para definir una función se debe escribir primero un **prototipo**, el cual declara los parámetros con sus tipos y el tipo que retorna la función
- A continuación se escribe el cuerpo de la función donde se explicita su definición.
- El **prototipo** tiene tres partes:
 - el tipo del valor que la función devuelve
 - el nombre de la función
 - la lista de argumentos de la función con sus tipos

Funciones:

```
TIPO_DEVOLUCION NOMBRE( TIPO_1 ARG_1, TIPO_2 ARG_2, ... ) /*  
protótipo */  
{  
    ... /* definición */  
}
```

- Las llaves { y } delimitan la definición de la función.
- Comparación con Haskell: $\text{TIPO_1} \rightarrow \text{TIPO_2} \rightarrow \dots \rightarrow \text{TIPO_DEVOLUCION}$.

Funciones: Ejemplo

```
int funcion( int a, int b, ...)  
{  
    int x;    /* declaración de variables al principio */  
    int y;  
  
    ...  
    /* existen las variables a, b, ... x, y, ... */  
    <instrucción>;  
    <instrucción>;  
  
    ...  
    return x; /* eligimos el valor que sirve como resultado */  
}
```

Funciones: El tipo void

- Cuando una función no devuelve ningún valor, escribimos que devuelve el tipo **void**:
- Cuando una función no recibe ningún argumento, escribimos **void** en lugar de su lista de argumentos:

```
void funcion( ... )  
{  
    ...  
}
```

```
... funcion( void )  
{  
    ...  
}
```

Funciones: llamada

- En el lenguaje C, la definición de una función debe estar antes de donde está llamada (acá, no se puede escribir la definición de `f` después la de `g` porque `g` llama a `f`).

```
int f(int a)
{
    return 2*a;
}
```

```
int g(void)
{
    int x;
    x = f(10);      /* siempre poner
                     parenthesis para llamar una
                     función */
    printf("x vale %d\n", x);
    return 1;
}
```

Funciones - Ejemplo

```
int f(int x)
{
    int i=2;
    while (x%i!=0 && x>1){
        i=i+1;
    }
    return (i==x);
}

int main(void) {
    int x;
    printf("Ingrese un valor para x\n");
    scanf("%d", &x);
    printf("El resultado es %d\n", f(x));
    return 0;
}
```

¿Qué hace esta función?

Si adivinaste, ponele un mejor nombre a `f`.