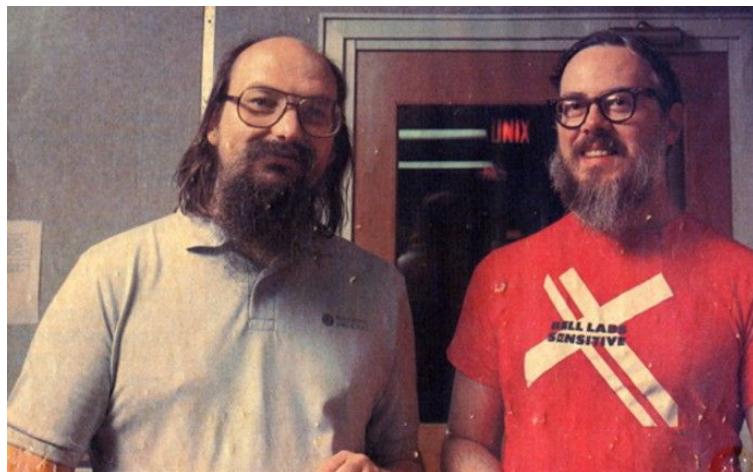


# Programación II

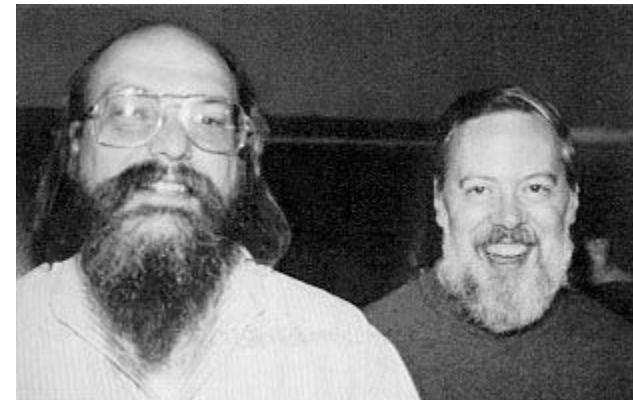
---

## Primeros pasos en C

C es un lenguaje de programación creado a principios de los 70s por Dennis Ritchie y Ken Thompson. Su desarrollo está asociado con el del sistema operativo Unix. En la segunda versión de Unix aparece el lenguaje C para escribir algunas utilidades.



Thompson - Ritchie

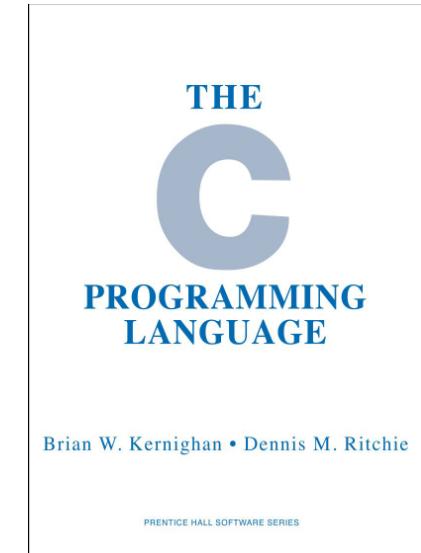


Thompson - Ritchie

En 1978 se publica el libro “The C Programming Language” (El lenguaje de programación C) escrito por Brian Kernighan y Dennis Ritchie. Este libro, conocido como K&R, se toma como una especificación informal del lenguaje de programación y es la bibliografía usualmente recomendada.



Kernighan - Ritchie





# Características del Lenguaje

---

C cuenta con ciertas características que hacen que su uso, luego de Python, nos pueda (sería lógico) resultar difícil. C es un lenguaje procedural, de estilo estructurado, con una sintaxis estricta.

Además de la sintaxis, C presenta una diferencia fundamental con respecto a Python y DrRacket. C es un lenguaje compilado.



## Características del Lenguaje

---

¿Qué significa esto? Que no es interpretado, es decir, no contamos con un intérprete para ir escribiendo nuestro código y que nos lo vaya evaluando.

¿Cómo se usa entonces? Es muy simple. Escribimos nuestro programa en uno o más archivos con extensión .c (también pueden ser .h pero eso lo veremos más adelante). Luego, usamos un programa llamado Compilador que toma, como entrada, nuestros archivos con código C y genera como salida un archivo ejecutable sobre el sistema operativo y arquitectura donde se compiló.



## Características del Lenguaje

---

Esto último significa que el archivo ejecutable generado no va a funcionar en cualquier máquina. Sólo en una con un sistema operativo y arquitectura similar.

El compilador que vamos a usar en la cátedra se llama gcc, hay otros pero este va a ser nuestro estándar. En Linux suele venir instalado mientras que en Windows hay que descargarlo. se puede hacer desde <http://www.mingw.org/> y descargando o, directamente cliqueando en: [download/installer](#).



# Características del Lenguaje

Una vez que lo tenemos instalado, vamos a escribir nuestro programa que imprima hola mundo. Para eso basta con escribir en un archivo, con extensión .c, el siguiente código:

```
#include<stdio.h>

int main() {
    printf("Hola Mundo!\n");
    return 0;
}
```



# Características del Lenguaje

La primera línea carga una librería. `stdio.h` contiene las funciones de entrada y salida que vamos a precisar para imprimir en pantalla.

Luego, se declara una función llamada `main`. Esta función es la que se ejecuta cuando se corre el programa. Debe existir en el programa a ejecutar. Es decir, sin importar qué contiene nuestro archivo, si la función `main` está vacía, el programa terminará automáticamente. Se puede notar que la función `main` retorna un entero. Este entero representa el estado de retorno del programa. Tradicionalmente 0 representa una terminación correcta, cualquier otro valor es señal de terminación anormal (error o excepción).

```
#include<stdio.h>

int main() {
    printf("Hola Mundo!\n");
    return 0;
}
```



# Características del Lenguaje

Dentro de la función `main` encontramos a la función `printf`. La misma se usa para imprimir en pantalla, como se podía suponer.

Podemos notar que cada sentencia lleva un `;` al final. Esto es parte de la sintaxis del lenguaje. Análogamente los bloques son limitados por `{ }`. Por ese motivo es que el código de la función `main` incluye la sentencia `printf` y `return`.

```
#include<stdio.h>

int main() {
    printf("Hola Mundo!\n");
    return 0;
}
```



## Características del Lenguaje

---

Una vez que ya tenemos escrito el programa lo vamos a compilar y ejecutar. En este caso, el archivo con el código anterior se llama `ejemplo1.c`.

Para poder compilar, si estamos en el directorio en que se encuentra nuestro archivo, la sentencia es:

`gcc ejemplo1.c`

Ahora vamos a ver esto puesto en práctica en ambos sistemas.



# Características del Lenguaje

En este caso, en Windows, se puede notar que obtuvimos un archivo ejecutable (con extensión .exe). El nombre, por defecto, es **a.exe**.

```
D:\Fede\Facultad\Programacion II\C\Ejemplos>gcc ejemplo1.c
D:\Fede\Facultad\Programacion II\C\Ejemplos>dir
El volumen de la unidad D es Nuevo vol
El n mero de serie del volumen es: 28D8-4EFE

Directorio de D:\Fede\Facultad\Programacion II\C\Ejemplos

17/11/2020  08:30    <DIR>    .
17/11/2020  08:30    <DIR>    ..
17/11/2020  08:30                54.024 a.exe
17/11/2020  08:30                61 ejemplo1.c
                  2 archivos          54.085 bytes
                  2 dirs   20.388.200.448 bytes libres

D:\Fede\Facultad\Programacion II\C\Ejemplos>a.exe
Hola Mundo!
```



# Características del Lenguaje

En Linux es similar, con la diferencia que el ejecutable se llama **a.out**.

```
fede@DESKTOP-NUTBUG5:/mnt/d/fede/Facultad/Programacion II/C/Ejemplos$ gcc ejemplo1.c
fede@DESKTOP-NUTBUG5:/mnt/d/fede/Facultad/Programacion II/C/Ejemplos$ dir
a.out ejemplo1.c
fede@DESKTOP-NUTBUG5:/mnt/d/fede/Facultad/Programacion II/C/Ejemplos$ ./a.out
Hola Mundo!
```



# Características del Lenguaje

Cuando tenemos un archivo que no contiene `main`, como se indicó en el [slide 9](#), se obtiene un error al querer compilar para generar un ejecutable.

```
#include<stdio.h>

int main1() {
    printf("Hola Mundo!\n");
    return 0;
}
```

```
fede@DESKTOP-NUTBUG5:/mnt/d/fede/Facultad/Programacion II/C/Ejemplos$ gcc ejemplo1.c
/usr/lib/gcc/x86_64-linux-gnu/7/../../../../x86_64-linux-gnu/Scrt1.o: In function `__start':
(.text+0x20): undefined reference to `main'
collect2: error: ld returned 1 exit status
```



# Tipos básicos

Los tipos básicos en C son:

TIPO	Rango
char	-128 a 127
short	-32768 a 32767
int	-32768 a 32767
long	-2147483648 a 2147483647
float	3.4E-38 a 3.4E+38
double	1.7E-308 a 1.7E+308

Como se puede ver, son todos rangos numéricos. Sin embargo, el tipo **char** representa un carácter. La forma de declararlo y asignarle un valor es:

```
char c = 'h';
```

Se usan las comillas simples.



# Tipos básicos

Los tipos básicos anteriores se pueden combinar o, agregar el atributo `unsigned`. Esto permite extender el rango superior, ya que sólo se representan valores positivos.

TIPO	Rango
<code>unsigned char</code>	0 a 255
<code>unsigned int</code>	0 a 65535
<code>short int</code>	-32768 a 32767
<code>unsigned short int</code>	0 a 65535
<code>long int</code>	-2147483648 a 2147483647
<code>unsigned long int</code>	0 a 4294967295
<code>unsigned long</code>	0 a 4294967295
<code>long double</code>	1.7E-308 a 1.7E+308 ó 3.4E-4932 a 1.1E+4932



# Tipos básicos

Vamos a crear un par de variables a modo de ejemplo, de diferentes tipos. Se puede ver que en la primera línea se declaran dos variables, inicializando sólo una de ellas.

```
int a, b = 4;  
unsigned int i = 10;  
char c = 'h';  
double d = 1.3;
```

Ahora bien, si se muestra el valor de `a`, ¿qué valor les parece que mostrará? La respuesta es que no se puede saber. Toda variable, en C, tiene un valor pero si no se le asigna dicho valor, se dice que es basura. Este valor no es otra cosa que lo que hay en memoria en el lugar que se le dio a la variable.



## Tipos básicos

---

En C se pueden declarar constantes. Declarar una constante es similar a declarar una variable, excepto que el valor no puede ser cambiado.

La palabra clave `const` se usa para declarar una constante, como se muestra a continuación:

```
const int a = 1;
```

Se puede usar `const` antes o después del tipo y, es usual inicializar una constante con un valor, ya que no puede ser cambiada de alguna otra forma.



# Tipos básicos

Si se intenta cambiar el valor de una constante surge un error al compilar.

```
#include<stdio.h>
int main() {
    const int a = 1;
    a = a + 1;
    return 0;
}
```

```
fede@DESKTOP-NUTBUGS:/mnt/d/Fede/Facultad/Programacion II/C/Ejemplos$ gcc ejemplo1.c
ejemplo1.c: In function 'main':
ejemplo1.c:10:4: error: assignment of read-only variable 'a'
    a = a + 1;
           ^
```

Como se puede ver, el compilador indica que la variable es de sólo lectura. Es decir, es una constante.



# Operadores - Operadores de Asignación

En C se puede hacer asignaciones, como en Python, pero también permite realizar una operación al momento de asignar. O sea, estas dos sentencias son iguales:

```
a += 5;
```

```
a = a + 5;
```

Así como se muestra con el operador `+` se puede realizar con otros operadores:

Operador	Acción	Ejemplo	Resultado
<code>=</code>	Asignación Básica	<code>X=6</code>	X vale 6
<code>*=</code>	Asigna Producto	<code>X*=5</code>	X vale 30
<code>/=</code>	Asigna División	<code>x/=2</code>	X vale 3
<code>+=</code>	Asigna Suma	<code>X+=4</code>	X vale 10
<code>-=</code>	Asigna Resta	<code>X-=1</code>	X vale 5
<code>%=</code>	Asigna Modulo	<code>X%=5</code>	X vale 1



# Operadores - Operadores de Asignación

---

Una característica con la que cuenta el operador de asignación es que permite este tipo de sentencias:

```
int a, b, c;  
a = b = c = 5;
```

Esto se debe a que el operador de asignación retorna el valor asignado. Entonces esta expresión se divide en:

1. se le asigna a **c** el valor 5 y se retorna 5;
2. a **b** se le asigna el valor resultante de 1 y se retorna dicho valor;
3. a **a** se le asigna el valor resultante de 2 y se retorna dicho valor (no es usado)



# Operadores - Operadores Aritméticos

---

Los operadores aritméticos para los tipos numéricos son:

Operador	Nombre	Ejemplo	Resultado
+	Suma	=10+5	15
-	Resta	=10-5	5
-	Negación	=-10	-10
*	Multiplicación	=10*5	50
/	División	=10/5	2



# Operadores - Operadores Aritméticos

En C también existen los operadores `++` y `--`. Dichos operadores son equivalentes, hasta cierto punto, a sumarle 1 o restarle 1 a la variable.

```
int i = 10;  
i++; //aquí i vale 11  
i--; //aquí i vale 10 nuevamente
```

Estos operadores se pueden poner antes o después de la variable. Si bien el resultado es el mismo varía su precedencia. Esto lo vamos a explicar con más detalle en la presentación siguiente.

```
int i = 10;  
++i; //aquí i vale 11  
--i; //aquí i vale 10 nuevamente
```



## Salida Estándar

---

Es importante tener en cuenta el tipo de los datos a la hora de usar los operadores en C. Vamos a ilustrar esto con un pequeño código de ejemplo que además nos servirá para entender cómo usar la función `printf` cuando se desea mostrar el valor de alguna variable.



# Salida Estándar

Supongamos que se tiene el siguiente código:

```
#include<stdio.h>
int main() {
    int a = 5, b = 2;
    int c = a/b;
    double d = a/b;

    printf("El valor de c es %d\n", c);
    printf("El valor de d es %lf\n", d);

    return 0;
}
```



# Salida Estándar

Las primeras tres líneas declaran e inicializan variables. Tres son de tipo `int` y, una es de tipo `double`.

El objetivo de las líneas siguientes es mostrar los valores que dichas variables almacenan. Allí podemos notar que la sintaxis del `printf` es distinta de la que se venía usando para imprimir en pantalla.

Vamos a explicarla.

```
#include<stdio.h>
int main() {
    int a = 5, b = 2;
    int c = a/b;
    double d = a/b;

    printf("El valor de c es %d\n", c);
    printf("El valor de d es %lf\n", d);

    return 0;
}
```



# Salida Estándar

---

La entrada y salida en C es formateada. Qué significa esto? Que hay que explicitar cómo hay que mostrar o leer un dato. Para esto hay una tabla de formatos:

Formateador	Salida
%d ó %i	entero en base 10 con signo (int) printf ("el numero enteron base 10 es: %d" , -10);
%u	entero en base 10 sin signo (int)
%o	entero en base 8 sin signo (int)
%x	entero en base 16, letras en minúscula (int)
%X	entero en base 16, letras en mayúscula (int)
%f	Coma flotante decimal de precisión simple (float)
%lf	Coma flotante decimal de precisión doble (double)
%ld	Entero de 32 bits (long)
%lu	Entero sin signo de 32 bits (unsigned long)
%e	La notación científica (mantisa / exponente), minúsculas (decimal precisión simple ó doble)
%E	La notación científica (mantisa / exponente), mayúsculas (decimal precisión simple ó doble)
%c	carácter (char)
%s	cadena de caracteres (string)



## Salida Estándar

En nuestro código de ejemplo se tiene una variable entera, cuyo formato es `%d` y, una double cuyo formato es `%lf`.

Entre comillas va la cadena que se quiere imprimir y en la posición donde queremos que aparezca el valor de la variable, se ubica el formato correspondiente.

```
printf("El valor de c es %d\n", c);
printf("El valor de d es %lf\n", d);
```

Ahora bien, ¿se podrán mostrar el valor de varios datos en la misma sentencia?



## Salida Estándar

La respuesta es sí. Esta sentencia es equivalente a las dos anteriores:

```
printf("El valor de c es %d\nEl valor de d es %lf\n", c, d);
```

¿Cómo está funcionando esto? Se puede ver que tenemos dos patrones (o marcadores) que se van a reemplazar por los datos que correspondan a dicha posición. O sea, el primer formato se va a aplicar al primer dato y, el segundo formato se va a aplicar al segundo dato.



## Salida Estándar

---

Entonces podemos concluir que se necesitan tener la misma cantidad de formatos que de datos.

Ahora bien, ¿qué sucederá si se ponen más datos que formatos? Podemos suponer que el programa va a funcionar igual porque las que sobren no van a ser usadas. Vamos a verificarlo.



# Salida Estándar

Escribimos este código:

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

    int a = 5, b = 2;
    int c = a/b;
    double d = a/b;

    printf("El valor de c es %d\n", c);
    printf("El valor de d es %lf\n", d);

    printf("El valor de c es %d\nEl valor de d es %lf\n", c, d, a);

    return 0;
}
```

y al compilar nos encontramos con esta salida:

```
fede@DESKTOP-NUTBUG5:/mnt/d/Fede/Facultad/Programacion II/C/Ejemplos$ gcc ejemplo1.c
ejemplo1.c: In function 'main':
ejemplo1.c:15:9: warning: too many arguments for format [-Wformat-extra-args]
    printf("El valor de c es %d\nEl valor de d es %lf\n", c, d, a);
          ^~~~~~
```

¿Qué significa esto?



# Salida Estándar

Nuestro programa generó un **warning**. Un **warning** es un aviso de que algo posiblemente no esté bien pero, a pesar de esto, se genera el programa para ejecutar. Es posible indicarle al compilador que no genere el ejecutable en caso de que haya **warnings**. Esto se indica agregando **-Werror** a la sentencia de compilación. De esta forma, los **warnings** pasan a ser tratados como errores.

```
fede@DESKTOP-NUTBUGS:/mnt/d/Fede/Facultad/Programacion II/C/Ejemplos$ gcc -Werror ejemplo1.c
ejemplo1.c: In function 'main':
ejemplo1.c:11:9: error: too many arguments for format [-Werror=format-extra-args]
    printf("El valor de c es %d\nEl valor de d es %lf\n", c, d, a);
          ^
cc1: all warnings being treated as errors
```



# Salida Estándar

Volviendo a nuestro **warning**, en este caso podemos ejecutarlo y funciona, tal cual se esperaba:

```
fede@DESKTOP-NUTBUG5:/mnt/d/Fede/Facultad/Programacion II/C/Ejemplos$ ./a.out
El valor de c es 2
El valor de d es 2.000000
El valor de c es 2
El valor de d es 2.000000
```

Sin embargo la salida es un poco confusa. ¿Por qué la división entre 5 y 2 es 2? Esto sucede porque se están dividiendo dos enteros y, en C, el resultado de dicha división es un entero sin importar el tipo de la variable en la que se almacene el resultado.



# Salida Estándar

Ahora bien, nos queda responder: ¿qué sucederá si se ponen menos datos que formatos? Podemos suponer que va a pasar algo similar a lo anterior. Veamos este código:

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

    int a = 5, b = 2;
    int c = a/b;
    double d = a/b;

    printf("El valor de c es %d\nEl valor de d es %lf\n", c);
    return 0;
}
```

y al compilar obtenemos

```
fede@DESKTOP-NUTBUG5:/mnt/d/Fede/Facultad/Programacion II/C/Ejemplos$ gcc ejemplo1.c
ejemplo1.c: In function ‘main’:
ejemplo1.c:8:50: warning: format ‘%lf’ expects a matching ‘double’ argument [-Wformat=]
    printf("El valor de c es %d\nEl valor de d es %lf\n", c);
                                         ^~~^
```



# Salida Estándar

¿Qué sucede al ejecutarlo? La realidad es que no se puede asegurar cuál será la salida. Al correrlo se obtiene:

```
fede@DESKTOP-NUTBUG5:/mnt/d/Fede/Facultad/Programacion II/C/Ejemplos$ ./a.out
El valor de c es 2
El valor de d es 0.000000
```

Sin embargo, con este código:

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

    int a = 5, b = 2;
    int c = a/b;
    double d = a/b;

    printf("El valor de c es %d\n", c);
    printf("Un número double es %lf\n", 0.25);

    printf("El valor de c es %d\nEl valor de d es %lf\n", c);
    return 0;
}
```



# Salida Estándar

Tenemos la siguiente salida:

```
fede@DESKTOP-NUTBUG5:/mnt/d/Fede/Facultad/Programacion II/C/Ejemplos$ gcc ejemplo1.c
ejemplo1.c: In function 'main':
ejemplo1.c:11:50: warning: format '%lf' expects a matching 'double' argument [-Wformat=]
    printf("El valor de c es %d\nEl valor de d es %lf\n", c);
                                         ^
fede@DESKTOP-NUTBUG5:/mnt/d/Fede/Facultad/Programacion II/C/Ejemplos$ ./a.out
El valor de c es 2
Un número double es 0.250000
El valor de c es 2
El valor de d es 0.250000
```

Como se puede notar el valor a mostrar va a depender de lo que suceda, previamente, en el programa.



## Salida Estándar

---

La conclusión de esto debería ser que los `warnings` están avisando de potenciales errores en tiempo de ejecución y **deben** ser revisados. Nuestro programa debería compilar sin `warnings`. Ni errores, por supuesto.



# Operadores - Operadores Aritméticos

Antes de continuar nos queda algo pendiente. ¿Cómo hacemos para obtener el resultado exacto de la división entre dos enteros?

Lo que hay que hacer es convertir a alguno de los dos argumentos a double. Esto se denomina **cast** o conversión de tipos.

En este caso, estamos “**casteando**” la variable **a**.

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

    int a = 5, b = 2;
    int c = a/b;
    double d = (double) a/b;

    printf("El valor de c es %d\n", c);
    printf("El valor de d es %lf\n", d);

    return 0;
}
```



# Operadores - Operadores Aritméticos

Si se compila y ejecuta este programa el resultado será:

```
Fede@DESKTOP-NUTBUG5:/mnt/d/Fede/Facultad/Programacion II/C/Ejemplos$ gcc -Werror ejemplo1.c
Fede@DESKTOP-NUTBUG5:/mnt/d/Fede/Facultad/Programacion II/C/Ejemplos$ ./a.out
El valor de c es 2
El valor de d es 2.500000
```

Si se hubiera “casteado” la variable **b**:

```
double d = a/(double)b;
```

obtendríamos lo mismo.



# Operadores - Operadores de Comparación

Nombre del operador	Sintaxis
Menor que	<code>a &lt; b</code>
Mayor que	<code>a &gt; b</code>
Menor o igual que	<code>a &lt;= b</code>
Mayor que o igual que	<code>a &gt;= b</code>
Igual que	<code>a == b</code>
Diferente que/No igual que	<code>a != b</code>
Negación lógica (NOT)	<code>! a</code>
AND logic	<code>a &amp;&amp; b</code>
OR logic	<code>a    b</code>

Si revisamos los tipos de datos vemos que no existe el tipo booleano. Esto es porque en C se asume que 0 es **false** y 1 es **true**.



# Funciones matemáticas

La librería `math.h` cuenta con las constantes y funciones matemáticas. Para usarlas se debe poner `#include<math.h>` al comienzo del archivo, con el resto de las librerías.

Constant	Description
<code>M_E</code>	2.7182818...
<code>M_PI</code>	3.1415926...

Nombre	Descripción
<code>acos</code>	arcocoseno
<code>asin</code>	arcoseno
<code>atan</code>	arcotangente
<code>atan2</code>	arcotangente de dos parámetros
<code>floor</code>	función suelo
<code>cos</code>	coseno
<code>cosh</code>	coseno hiperbólico
<code>exp(double x)</code>	función exponencial, computa $e^x$
<code>fabs</code>	valor entero
<code>ceil</code>	menor entero no menor que el parámetro
<code>fmod</code>	residuo de la división de flotantes
<code>frexp</code>	fracciona y eleva al cuadrado.
<code>ldexp</code>	tamaño del exponente de un valor en punto flotante
<code>log</code>	logaritmo natural
<code>log10</code>	logaritmo en base 10
<code>modf</code>	obtiene un valor en punto flotante íntegro y en partes
<code>pow</code>	eleva un valor dado a un exponente, $x^y$
<code>sin</code>	seno
<code>sinh</code>	seno hiperbólico
<code>sqrt</code>	raíz cuadrada
<code>tan</code>	tangente
<code>tanh</code>	tangente hiperbólica

Como se mencionó en el [slide 10](#) los bloques son marcados con el uso de { }. Sin embargo se recomienda continuar indentando el código para que el mismo sea legible. Esta es una práctica que no se puede dejar de lado por el hecho de que ya no sea una característica propia del lenguaje.



## Entrada estándar

Para poder tomar la entrada estándar se debe usar la función `scanf`. La misma tiene el uso de formatos igual a lo que vimos en `printf`. Un primer ejemplo es:

```
int i;  
printf("Ingrese un número: ");  
scanf("%d", &i);
```

Algo que debemos notar es el uso del símbolo & delante de la variable. Esto siempre es así para la lectura de datos enteros y caracteres.



# Entrada estándar

Si no lo ponemos, para estos casos, el compilador nos va a avisar con un **warning**.

```
#include<stdio.h>

int main() {

    int i;
    char c;

    printf("Ingrese un carácter y un número: ");
    scanf("%c%d", c, i);

    printf("Ingresó %c y %d", c, i);

    return 0;
}
```

```
fede@DESKTOP-NUTBUGS:/mnt/d/Fede/Facultad/Programacion III/C/Ejemplos$ gcc ejemplo1.c
ejemplo1.c: In function ‘main’:
ejemplo1.c:10:10: warning: format ‘%c’ expects argument of type ‘char *’, but argument 2 has type ‘int’ [-Wformat=]
    scanf("%c%d", c, i);
           ~^
ejemplo1.c:10:12: warning: format ‘%d’ expects argument of type ‘int *’, but argument 3 has type ‘int’ [-Wformat=]
    scanf("%c%d", c, i);
           ~^
```



# Entrada estándar

Es posible leer varias variables en la misma sentencia. En ese caso, en el ingreso los datos se pueden separar por un espacio blanco, un tabulador o un salto de línea.

```
#include<stdio.h>

int main() {
    int i;
    char c;

    printf("Ingrese un carácter y un número: ");
    scanf("%c%d", &c, &i);

    printf("Ingresó %c y %d\n", c, i);
    return 0;
}
```

```
fede@DESKTOP-NUTBUGS:/mnt/d/Fede/Facultad/Programacion II/C/Ejemplos$ gcc ejemplo1.c
fede@DESKTOP-NUTBUGS:/mnt/d/Fede/Facultad/Programacion II/C/Ejemplos$ ./a.out
Ingrese un carácter y un número: h 5
Ingresó h y 5
```

En C la sintaxis es similar a la que se vio en Python:

```
if (condición1) {  
    bloque si se cumple condición1  
}  
else if (condición2) {  
    bloque si se cumple condición2  
}  
else {  
    bloque en caso contrario  
}
```



# Sentencia IF

---

Aquí se puede ver un ejemplo del uso de **if**, **else if** y **else**.

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

    int a, b, mayor;
    printf("Ingrese dos números:");
    scanf("%d%d", &a, &b);
    if (a > b) {
        printf("El mayor es %d\n", a);
    }
    else if (a < b) {
        printf("El mayor es %d\n", b);
    }
    else{
        printf("Son iguales!\n");
    }

    return 0;
}
```

La salida sería:

```
fede@DESKTOP-NUTBUG5:/mnt/d/Fede/Facultad/Programacion II/C/Ejemplos$ ./a.out
Ingrese dos números:5 2
El mayor es 5
fede@DESKTOP-NUTBUG5:/mnt/d/Fede/Facultad/Programacion II/C/Ejemplos$ ./a.out
Ingrese dos números:2 5
El mayor es 5
fede@DESKTOP-NUTBUG5:/mnt/d/Fede/Facultad/Programacion II/C/Ejemplos$ ./a.out
Ingrese dos números:2 2
Son iguales!
```



# Bucles While

---

La sintaxis del bucle `while` en C es la siguiente:

```
while (condición) {  
    sentencias  
}
```

El concepto es el mismo que se tenía en Python: mientras condición sea verdadera se ejecutan las sentencias.



# Bucles While

Un pequeño ejemplo de uso podría ser:

```
#include<stdio.h>

int main() {
    int i = 3;

    while (i>0) {
        printf("x = %d\n", i);
        i--;
    }

    return 0;
}
```

```
Fede@DESKTOP-NUTBUG5:/mnt/d/Fede/Facultad/Programacion II/C/Ejemplos$ gcc ejemplo1.c
Fede@DESKTOP-NUTBUG5:/mnt/d/Fede/Facultad/Programacion II/C/Ejemplos$ ./a.out
x = 3
x = 2
x = 1
```



# Funciones

---

Como vimos, las funciones en C se declaran de la siguiente forma:

```
tipo_retorno mi_funcion(tipo1 param1, tipo2 param2){  
    sentencias  
}
```

O sea, se debe indicar el tipo de retorno y, de cada argumento que reciba la función. El tipo **void** indica que nuestra función no retorna valor y, por eso mismo, es que no hay un **return**.

```
void mi_funcion(int param1, int param2) {  
    printf("%d %d", param1, param2);  
}
```



# Funciones

En C no existe la posibilidad de asignar valores por defecto para los parámetros, en caso que los mismos no sean pasados como argumento se produce un error al compilar:

```
#include<stdio.h>

void mi_funcion(int param1, int param2) {
    printf("%d %d", param1, param2);
}

int main() {
    mi_funcion(1);
    return 0;
}
```

```
Fede@DESKTOP-NUTBUG5:/mnt/d/Fede/Facultad/Programacion II/C/Ejemplos$ gcc ejemplo1.c
ejemplo1.c: In function 'main':
ejemplo1.c:10:2: error: too few arguments to function 'mi_funcion'
  mi_funcion(1);
  ^~~~~~
ejemplo1.c:4:6: note: declared here
  void mi_funcion(int param1, int param2) {
  ^~~~~~
```



# Funciones

Ahora bien, si tenemos el siguiente código:

```
#include <stdio.h>

int main(){
    int i = f();
    printf("%d", i);
}

int f() {
    return 1;
}
```

Al compilar se tiene la siguiente salida:

```
fede@DESKTOP-NUTBUG5:/mnt/d/Fede/Facultad/Programacion II/C/Ejemplos$ gcc -Wall ejemplo2.c
ejemplo2.c: In function 'main':
ejemplo2.c:6:10: warning: implicit declaration of function 'f' [-Wimplicit-function-declaration]
    int i = f();
           ^
```

Esto es porque en C no se pueden escribir las funciones en cualquier lugar del archivo. El compilador espera que estén antes del `main`.



# Funciones

Si se cambia el orden:

ahora sí compila sin problemas:

```
#include <stdio.h>

int f() {
    return 1;
}

int main(){
    int i = f();
    printf("%d", i);
}
```

```
fede@DESKTOP-NUTBUG5:/mnt/d/Fede/Facultad/Programacion II/C/Ejemplos$ gcc -Wall ejemplo2.c
```



# Funciones

---

Entonces, ¿todas las funciones tienen que estar declaradas antes del `main`? La respuesta es que sí, salvo que se definan los **prototipos** (o firma o signature) de las mismas antes del `main`.

Un **prototipo** de una función es el tipo de retorno, el nombre y los tipos de argumentos.

Por ejemplo, los siguientes son **prototipos** de funciones:

```
int f();  
int f1(int , double);  
void f2(int a);
```

Como se vé, se puede poner el identificador asociado al tipo o no.



# Funciones

Usando esto, entonces este código ahora sí compila.

```
#include <stdio.h>
int f();

int main(){
    int i = f();
    printf("%d", i);
}

int f() {
    return 1;
}
```

```
fede@DESKTOP-NUTBUG5:/mnt/d/Fede/Facultad/Programacion II/C/Ejemplos$ gcc -Wall ejemplo2.c
```



# Funciones Recursivas

Como ejemplo vamos a implementar la función recursiva factorial.

```
int factorial(int n) {
    if (n==0) {
        return 1;
    }
    else {
        return n * factorial(n-1);
    }
}
```

El concepto para plantear la recursión es similar a lo que se había visto en Python.



# Funciones Recursivas

El programa completo, que muestre su uso, podría ser:

```
#include<stdio.h>
int factorial(int n){
    if (n==0) {
        return 1;
    }
    else {
        return n * factorial(n-1);
    }
}
int main() {

    int num;

    printf("Ingrese un número positivo para calcular su factorial: ");
    scanf("%d", &num);

    printf("El resultado del factorial es %d\n", factorial(num));

    return 0;
}
```



# Funciones Recursivas

Otra variante sería guardar en una variable el resultado de la función y luego mostrarla.

```
#include<stdio.h>
int factorial(int n){
    if (n==0) {
        return 1;
    }
    else {
        return n * factorial(n-1);
    }
}
int main() {

    int num;

    printf("Ingrese un número positivo para calcular su factorial: ");
    scanf("%d", &num);

    int resultado = factorial(num);
    printf("El resultado del factorial es %d\n", resultado);

    return 0;
}
```