Taller de programación: C - Parte 1

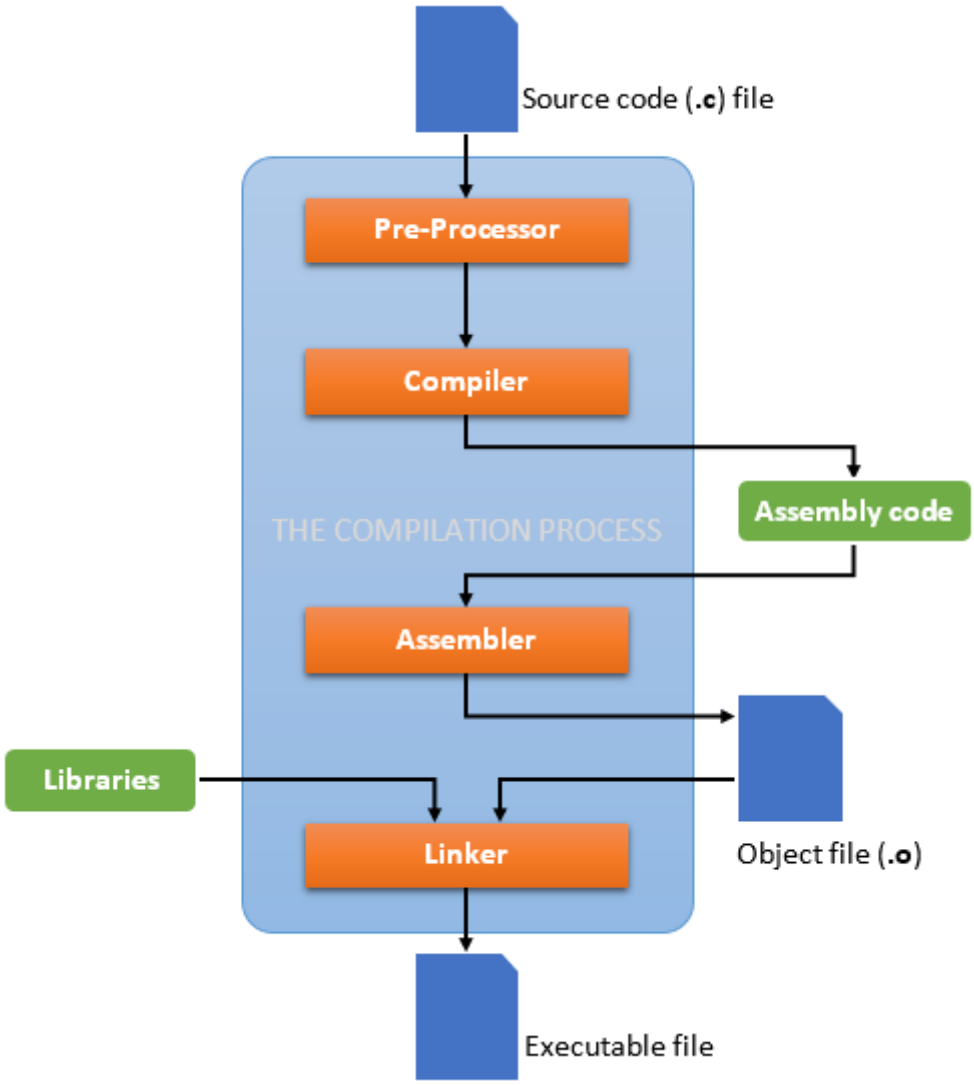
Autor: Luis A. Muñoz - UPC- 2020

Este documento presenta un resumen de los principales elementos del lenguaje de programación C, así como ejemplos de código sencillos y buenas prácticas de programación en C.

¿Por qué C?

C es un lenguaje de programación de propósito general, originalmente desarrollado por Dennis Ritchie entre 1969 y 1972 en los Laboratorios Bell, como evolución del anterior lenguaje B, a su vez basado en BCPL.

Es un lenguaje *compilado*, lo que significa que el código escrito en C se convierte en código binario por medio de un proceso llamado *compilación*. El resultado es un archivo ejecutable por el sistema operativo en *Assembler*, por lo que tendrá una velocidad de ejecución muy rápida.



Por otro lado, es un lenguaje de nivel-medio, lo que significa que no es tan complicado como un lenguaje de bajo-nivel (Assembler), pero no tan sencillo como un lenguaje de alto-nivel (Python). La ventaja es que permite construcciones de código que resultan en bloques de ejecución de rápida velocidad y eficientes en el uso de los recursos (como RAM y registros internos del CPU).

Saber programar en lenguaje C permite conocer los detalles de como funciona un programa desde el punto de vista de los recursos del sistema. Haciendo una analogía, es como conocer, además de la mecánica de un automóvil, la forma como la energía química del combustible se convierte en energía mecánica. Esto no solo nos hará mejores pilotos, sino que seremos capaces de comprender como funciona un motor de avión, dado el caso. Así, aprender a programar en lenguaje C es equivalente a tener el conocimiento necesario para comprender posteriormente cualquier lenguaje de programación.

Sobre su aplicación, actualmente en índice TIOBE (https://www.tiobe.com/tiobe-index/) (que mide mensualmente la popularidad de los lenguajes de programación) coloca a C en una posición priviegiada, en parte porque es el lenguaje de progamación de los videojuegos, así como de las aplicaciones de IoT (Internet de las Cosas) ya que están basados en microcontroladores y sistemas embebidos.

¿Qué se necesita para programar en lenguaje C?

C es un lenguaje muy económico: solo se necesita un editor de texto y el programa compilador. Así tambien se puede utilizar un IDE (Integrated Development Environment) que combina los dos anteriores y herramientas adicionales en un solo paquete. Ejemplos de esto son Dev-C, Code::Blocks y Visual Studio. El compilador de C es un programa llamado `gcc.exe` que se puede descargar. Una vez que se tiene el código de un programa, este se guarda con el formato de nombre `archivo.c` y luego se compila con la instrucción:

```
gcc archivo.c -o archivo.exe
```

Esto generará un archivo `archivo.exe` que se podrá ejecutar desde el sistema operativo. Los IDEs realizan este proceso de manera automática (normalmente con una opción llamada *Build*) y también pueden ejecutar el programa (opción *Run*).

El IDE recomendado para programar en C es Code::Blocks con la inclusión del compilador `gcc` para Windows llamado *MinGW* (http://sourceforge.net/projects/codeblocks/files/Binaries/20.03/Windows/codeblocks-20.03mingw-setup.exe (http://sourceforge.net/projects/codeblocks/files/Binaries/20.03/Windows/codeblocks-20.03mingw-setup.exe)).

Pero lo que realmente se necesita para aprender a programar en lenguaje C es:

- Programar, programar, programar....
- Entender los detalles de bajo nivel de las operaciones
- Saber que es posible, pero que no será fácil.

Sobre los lenguajes de programación

Todos los lenguajes de programación hacen las mismas cosas y estas son:

- Son capaces de recibir datos del mundo real y almacenarlos en algun recurso
- Son capaces de mostrar resultados bajo un formato y dispositivo
- Son capaces de realizar operaciones logíco-matemáticas
- Son capaces de tomar decisiones en función de condicones lógicas y modificar las operaciones a realizar
- Son capaces de repetir seccciones de operaciones de forma controlada.

Esto es todo. Cuando se aprende un lenguaje de programación es importante tener esto en mente para ordenar lo que se va apeendidndo bajo esta esquema.

Radiografía de un programa en C: *hola mundo*

- Todo programa en C inicia con las *directivas del preprocesador*, una línea que tiene la forma `#include <header_file>` . Esto le indica al compilador que debe de incluir el código almacenado en el archivo `header_file` .
- Un programa en C esta basado en una función llamada `main()` . Una función es una agrupación de acciones lógicas. o instrucciones de programación bajo un sólo nombre. Generalmente toman valores de entrada llamados *parametros* y retornan resultados. Lo que hace de `main()` especial es que es la función que se llamada cuando se ejecuta el programa y debe de tener ese nombre.
- Las instrucciones que forman parte de un programa estan agrupadas en un bloque entre `{}` utilizando una sangría de nivel (de 2 ó 4 espacios) que permite organizar el código.
- Las instrucciones en C terminan con el caracter `;` y son necesarios para indicar el final de una instrución (¡y olvidar esto es la problema número uno de todos los estudiantes de lenguaje C!)
- La función `main` retorna una resultado entero (por eso el `int` al principio de la definición de la función), por lo que la última línea de la función es `return 0;` , que retorna el valor `0` al finalizar la función (suele indicarse con este valor es el programa termino sin errores con un número positivo, o si se presento un error con un valor negativo).

In [1]:

```
#include <stdio.h>

int main()
{
    printf("Hello f*** world!");

    return 0;
}
```

```
Hello f*** world!
```

Instrucciones

Una instrucción es una operación que realiza una acción aritmética o lógica en el programa y suelen terminar con `;` . Por ejemplo:

```
puts("Hola mundo");
```

es una instrucción. Pero:

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

no es una instrucción. Esta distinción es importante para entender las reglas de escritura del código en lenguaje C.

Variables

Las variables son etiquetas que hacen referencia a posiciones de memoria donde se almacenan los datos. Por lo tanto, una variable tiene:

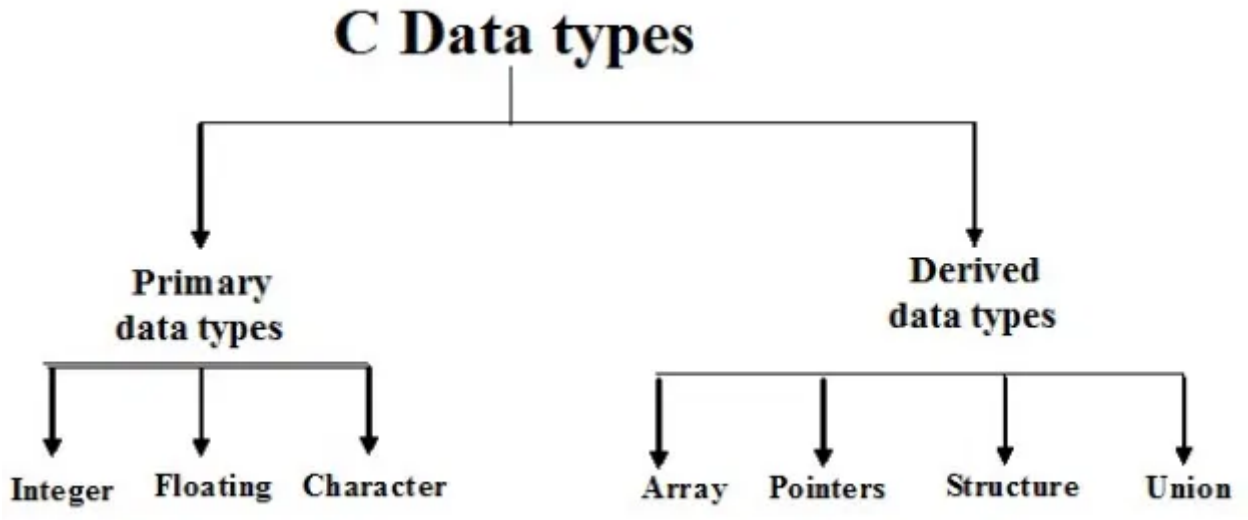
- Una etiqueta
- Un tipo
- Una dirección
- Un valor

Todos estos detalles son importantes para comprender las operaciones y la idiosincracia detrás del lenguaje C.

En el siguiente programa se *declaran* tres variables de los tres tipos principales:

- `int` , para almacenar valores enteros
- `float` , para almacenar valores reales
- `char` , para almacenar caracteres (es decir, letras). Es especifican con comillas simples('')

Posteriormente se pueden *asignar* valores a las variables seún el tipo declarado; no se puede asigar una letras a una variable del tipo *float*, y viceversa.

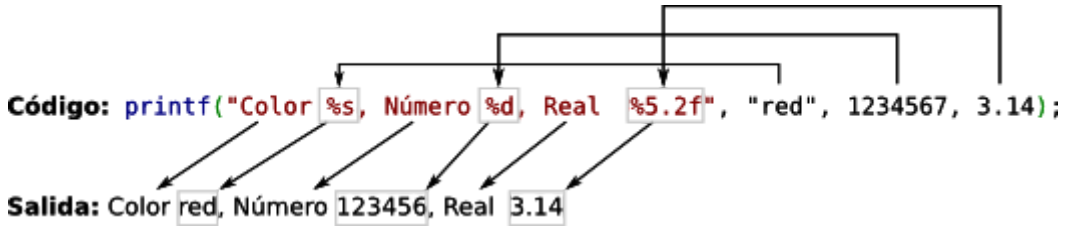


printf: impresión con formato

Podemos imprimir en la salida estandar `stdin` (que normalemnte es la consola del sistema operativo) con la función `puts()` (abreviatura de *put string*) una cadena o *string* (una cadena es una secuencia de chars, o sea una palabra o incluso una oración).

Otra forma de imprimir en la consola es utilizando la función `printf()` de la biblioteca `stdio.h` que *imprime con formato*; esto quiere decir que incluye algunos elementos a la cadena de impresión que pueden contener formato de impresión. Para esto último será necesario utilizar los *caracteres de conversión* que permitirán insertar los el valor de las variables en determinadas posiciones en el *string*. Estos pueden ser:

- `%d` (reemplazar con un valor tipo int. Puede incluír el tamaño de la celda de impresión, por ejemplo `%3d`).
- `%f` (reemplazar con un valor tipo float. Puede incluír el número decimales, por ejemplo `%.4f` para 4c decimales).
- `%c` (reemplazar con un valor tipo char).
- `%s` (reemplazar por un string). Un string se especifica con comillas dobles (" ")



En en el caso de valores tipo `float` , `printf` nos permite especificar el número decimales a mostrar. La especificación `%5.2f` indica que que quiere visualizar un número en un espacio de 5 caracteres (incluyendo el punto decimal) con 2 decimales a mostrar. Esto también se puede especificar de la forma `%.2f` , indicando solo el número de decimales a mostrar; `printf` se encargará de darle espacio a la impresión del número para que ocupe el espacio necesario de forma automática.

Adicinonalmente también existen *caracteres de escape*, que permiten "imprimir" acciones sobre la consola, como saltar a una nueva línea (`\n`) y insertar un tabulador (`\t`).

Caracter de escape	Propósito
\n	Inserta una nueva línea
\t	Mueve el cursor al siguiente tabulador
\r	Mueve el cursor al inicio de la línea
\	Inserta el caracter \
\"	Inserta el caracter "
\'	Inserta el caracter '

Asi también, se muestra como comentario las funciones que se pueden utilizar como parte de la inclusión en el código de las diferentes librerías.

In [3]:

```c
#include <stdio.h>   // printf, scanf, puts, sizeof
#include <stdlib.h>  // NULL, rand, rand, exit, atof, atoi
#include <string.h>  // string funcs
#include <time.h>    // time

int main()
{
    // Declaracion de variables
    int peso;
    float altura;
    char sexo;

    // Asignacion de variables
    peso = 67;
    altura = 1.65;
    sexo = 'M';

    puts("Paciente:");
    printf("\t* Sexo: %c\n\t* Peso: %d kg\n\t* Altura: %.2f\n\t* IMC: %.2f\n",
           sexo, peso, altura, peso / (altura * altura));

    return 0;
}
```

```
Paciente:
        * Sexo: M
        * Peso: 67 kg
        * Altura: 1.65
        * IMC: 24.61
```

scanf: ingreso de datos desde el teclado

Se puede recibir información del mundo exterior de muchas formas, pero la más tradicional es a traves de el teclado. Esto se lográ en C con las instrucción `scanf` de la biblioteca `stdio.h` que permite "escanear" el teclado y reconocer los caracteres ingresados bajo un formato especificado. Modifiquemos el programa anterior para que se puedan ingresado los datos desde el teclado:

In [ ]:

```c
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <time.h>

int main()
{
    // Declaracion de variables
    int peso = 0;
    float altura = 0.0, IMC = 0.0;
    char sexo = ' ';

    /* Caracteres de conversion:
       %d: int
       %f: float
       %c: char
    */
    printf("Ingrese sexo [M/F]: ");
    scanf("%c", &sexo);
    printf("Ingrese su peso [kg]: ");
    scanf("%d", &peso);
    printf("Ingrese altura [m]: ");
    scanf("%f", &altura);

    IMC = peso / (altura * altura);

    /* Caracteres de escape:
      \n: New Line
      \t: Tab
      \b: Backspace
      \a: Alarm
    */
    printf("Paciente:\n\t* Sexo: %c\n\t* Peso: %d kg\n\t* Altura: %.2f\n\t* IMC: %.2f\n
",
           sexo, peso, altura, IMC);

    return 0;
}
```

Observe la línea:

```c
scanf("%c", &sexo);
```

Esto espera leer desde el teclado un caracter ('M'/'F') y por eso el caracter de conversión `%c`. Lo importante es el uso del operación *dirección de* `&`. Esto devuelve la dirección en memoria de una variable. Por lo tanto, `scanf` recibirá un dato del teclado (una vez presionado ENTER) e interpretará ese valor como un `char` (o sea, un valor de 1 byte) y lo almacenará en la dirección de memoria separada especialmente para almacenar un caracter (por eso la declaración de la variable son anterioridad) con la etiqueta *sexo*. Un error muy común en programación en C es olvidar el operador `&` en `scanf`. ¡Mucho ojo! Hágalo y observe que el compilador no mostrará un error por lo que el programa se ejecutará, pero en la consola no se verá nada.

Operaciones en C y orden de precedencia

Todos los lenguajes de programación puede realizar las siguintes operaciones:

- Operaciones aritméticas (+, -, *, /, %).
- Operaciones de relación (>, <, >=, <=, ==, !=)
- Operaciones lógicas (&&, ||, !)

Y en el caso de C, además:

- Operaciones de asignación (=, +=, -=, *=, /=, %=)
- Operaciones de incremento y decremento (var++, ++var, --var, var--)

Operaciones aritméticas

Lenguaje C soporta 5 operaciones aritméticas:

- Suma (+)
- Resta (-)
- Multiplicación (*)
- División (/)
- Módulo (%)

In [19]:

```
#include <stdio.h>

int main()
{
    int iNum1 = 18, iNum2 = 12;

    printf("%d + %d = %d\n", iNum1, iNum2, iNum1 + iNum2);
    printf("%d - %d = %d\n", iNum1, iNum2, iNum1 - iNum2);
    printf("%d * %d = %d\n", iNum1, iNum2, iNum1 * iNum2);
    printf("%d / %d = %d\n", iNum1, iNum2, iNum1 / iNum2);
    printf("%d %% %d = %d\n", iNum1, iNum2, iNum1 % iNum2);   // "%%" permite imprimir
 "%"

    return 0;
}
```

```
18 + 12 = 30
18 - 12 = 6
18 * 12 = 216
18 / 12 = 1
18 % 12 = 6
```

Los resultados son bastante sencillos de interpretar en los casos de +, -, y . *En el caso de / lo que retorna es una división entera o* truncada* que no incluye la parte decimal porque todos los valores involucrados en la división son enteros. Si modifica el caracter de conversión por `%f` el problema persiste:

In [20]:

```c
#include <stdio.h>

int main()
{
    int iNum1 = 18, iNum2 = 12;

    printf("%d / %d = %f\n", iNum1, iNum2, iNum1 / iNum2);

    return 0;
}
```

```
/tmp/tmpzqm_0siv.c: In function ‘main’:
/tmp/tmpzqm_0siv.c:7:24: warning: format ‘%f’ expects argument of type ‘do
uble’, but argument 4 has type ‘int’ [-Wformat=]
    7 |     printf("%d / %d = %f\n", iNum1, iNum2, iNum1 / iNum2);
      |                       ~^                   ~~~~~~~~~~~~
      |                        |                          |
      |                        double                     int
      |                       %d
```

```
18 / 12 = 0.000000
```

El compilador lanza un error que indica que el resultado de la operación es un valor entero por lo que requiere `%d` para su interpretación. Esto no es un error, es un *warning*, lo que no evita que el programa se ejecute. Se observa que el `%d` fuerza la impresión de valores decimales, pero no con los valores correctos.

Para solucionar esto se cuenta con dos soluciones. Primero hacer que los números no sean `int`; esto es en lugar de realizar la instrucción 18/12, realizar `18.0 / 12`, o `18 / 12.0`, o `18.0 / 12.0`. Todas estas operaciones retornarán un valor flotante.

La otra solución es hacer un *typecast*; esto es, hacer una conversión de tipos en esa sección del código. Para esto se coloca `(type)` antes de una de las variables con el nuevo tipo:

In [22]:

```c
#include <stdio.h>

int main()
{
    int iNum1 = 18, iNum2 = 12;

    printf("%d / %d = %.2f\n", iNum1, iNum2, (float)iNum1 / iNum2);

    return 0;
}
```

```
18 / 12 = 1.50
```

El operador `%` retorna el residuo de la división entre dos numeros. Esto suele ser muy útil para restringir un resultado numérico en un rango. Debe entenerse las operaciones modulares como *matemática de reloj*: cuando decimos "las 15 horas" estamos especificando la hora en formato militar y para obtener el valor en un reloj de manecillas hacemos la operación *hora % 12*, de forma tal que dividimos 15 entre 12 y nos quedamos con el residuo: las 3 horas.

Esto permite entender porque *5 % 12* resulta en *5*, y sobre todo porque *-2 % 12* resulta en *10* (¿qué hora sera 2 horas antes de las 12?).

Operadores de relación

Otros operadores son los llamados de relación ya que establecen valores lógicos sobre una consulta relacional:

- Mayor que (>)
- Menor que (<)
- Mayor o igual (>=)
- Menor o igual (<=)
- Igual (==)
- Diferentes (!=)

In [25]:

```c
#include <stdio.h>

int main()
{
    printf("\n8 > 5 = %d", 8 > 5);
    printf("\n8 < 5 = %d", 8 < 5);
    printf("\n5 >= 5 = %d", 5 >= 5);
    printf("\n3 <= 5 = %d", 3 <= 5);
    printf("\n6 == 6 = %d", 6 == 6);
    printf("\n6 != 6 = %d", 6 != 6);

    return 0;
}
```

```
8 > 5 = 1
8 < 5 = 0
5 >= 5 = 1
3 <= 5 = 1
6 == 6 = 1
6 != 6 = 0
```

Los resultados de las operaciones relacionales son valores de verdadero o falso (`true` o `false`) expresados como valores de tipo entero. En términos generales, un valor 0 es igual a `false` , miestras que un valor diferentes de 0 es igual a `true` .

Operadores lógicos

Otros operadores llamados lógicos realizan operaciones de tipos AND (&&), OR(||) y NOT(!):

In [35]:

```c
#include <stdio.h>

int main()
{
    printf("Operador AND (&&):\n");
    printf("\t0 && 0 = %d\n", 0 && 0);
    printf("\t0 && 1 = %d\n", 0 && 1);
    printf("\t1 && 0 = %d\n", 1 && 0);
    printf("\t1 && 1 = %d\n", 1 && 1);

    printf("\nOperador OR (||):\n");
    printf("\t0 || 0 = %d\n", 0 || 0);
    printf("\t0 || 1 = %d\n", 0 || 1);
    printf("\t1 || 0 = %d\n", 1 || 0);
    printf("\t1 || 1 = %d\n", 1 || 1);

    printf("\nOperador NOT (!):\n");
    printf("\t!0= %d\n", !0);
    printf("\t!1= %d\n", !1);

    return 0;
}
```

```
Operador AND (&&):
        0 && 0 = 0
        0 && 1 = 0
        1 && 0 = 0
        1 && 1 = 1

Operador OR (||):
        0 || 0 = 0
        0 || 1 = 1
        1 || 0 = 1
        1 || 1 = 1

Operador NOT (!):
        !0= 1
        !1= 0
```

Operadores de asignación

Hemos utilizado el operador de asignación = que permite asignar un valor a una variable. Sin embargo hay algunas modificaciones del operador asignación que asigna el resultado de una operación de forma resumida:

In [45]:

```c
#include <stdio.h>

int main()
{
    int iNum = 1;

    printf("iNum = %d\n", iNum);
    printf("iNum = iNum + 1 : %d\n", iNum + 1);

    iNum = 1;

    printf("\niNum = %d\n", iNum);
    printf("iNum += 1 : %d\n", iNum += 1);

    return 0;
}
```

```
iNum = 1
iNum = iNum + 1 : 2

iNum = 1
iNum += 1 : 2
```

Asi, tenemos los siguientes operadores de asignación:

- `var += num` es igual a `var = var + num`
- `var -= num` es igual a `var = var - num`
- `var *= num` es igual a `var = var * num`
- `var /= num` es igual a `var = var / num`
- `var %= num` es igual a `var = var % num`

Operador incremento

La operación `var += 1` o `var -= 1` son bastante comunes en programación y se conocen como *incremento* o *decremento* de una variable. En C tienen un operador especial: `var++` y `var--` . Esto exste en dos variaciones: *pre* y *post*:

In [50]:

```c
#include <stdio.h>

int main()
{
    int iNum = 5;

    printf("iNum++ = %d\n", iNum++);
    printf("++iNum = %d\n", ++iNum);

    return 0;
}
```

```
iNum++ = 5
++iNum = 7
```

En el código anterior, la operación `iNum++` es del tipo *post* y re realiza luego de la asignación. Así, se reemplaza el caracter `%d` por el valor de *5* y luego se incrementa al valor *6*. Luego, en la siguiente línea la operación `++iNum` es del tipo *pre* por lo que el valor de *6* se incrementa a *7* y luego se reemplaza por el caracter `%d` .

En general, esta diferencia no suele ser de particular importancia en un código de programación. Sin embargo, si los resultados no son los esperados al momento de utilizar el operador de incremento o decremento, puede deberse al uso de la versión *pre* o *post*.

Orden de precedencia

Las operaciones tienen un orden de precedencia que debe de considerarase.

1. Operaciones entre paréntesis
2. Signo mas o signo menos (operación unaria), incremento o decremento (++, --), NO (!) y typecast (tipo)
3. mutiplicación (*), division (/) y módulo (%)
4. Suma (+) y resta (-)
5. Menor, menor igual, mayor, mayr o igual (<, <=, >, >=)
6. Igual, diferente (==, !=)
7. AND (&&)
8. OR (||)
9. Asignación (==, +=, -=, *=, /=, %=)

No intente memorizar este orden: solo recuerde utilizar paréntesis para agrupar las operaciones que desea agrupar (sobre todo al combinar AND y OR).

El caracter `&` es el operador dirección y retorna la dirección en memoria de una variable. Esto quiere decir que `scanf` toma un valor ingresado desde el teclado y lo almacena en una posición de memoria etiqueta con un nombre de variable. Un error muy común entre los estudiantes de C es olvidar el caracter `&` en `scanf` .

Variables: almacenamiento de datos en memoria

Es importante insistír en que una variable no es más que una etiqueta que hace referencia a una posición de memoria, en especial en C, ya que este lenguaje de programación trabaja a un nivel-medio y puede realizar operaciones sobre las direcciones de memoria (es decir, accede directamente a modificar datos en la memoria RAM y por eso su uso en los sistemas embebidos o microprocesados).

In [ ]:

```c
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <time.h>

int main()
{
    int anio = 2020;
    const float PI = 3.1415;
    char letra = 'a';
    char nombre[] = "Elvio Lado";   //  \0: NULL

    puts("var: anio");
    printf("\tTipo: int\n");
    printf("\tValor: %d\n", anio);
    printf("\tDireccion de memoria: %d\n", &anio);
    printf("\tNum bytes: %d\n\n", sizeof(anio));

    puts("var: PI");
    printf("\tTipo: float\n");
    printf("\tValor: %f\n", PI);
    printf("\tDireccion de memoria: %d\n", &PI);
    printf("\tNum bytes: %d\n\n", sizeof(PI));

    puts("var: letra");
    printf("\tTipo: char\n");
    printf("\tValor: %c\n", letra);
    printf("\tDireccion de memoria: %d\n", &letra);
    printf("\tNum bytes: %d\n\n", sizeof(letra));

    puts("var: nombre");
    printf("\tTipo: char[] - String\n");
    printf("\tValor: %s\n", nombre);
    printf("\tDireccion de memoria: %d\n", &nombre);
    printf("\tNum bytes: %d\n\n", sizeof(nombre));

    return 0;
}
```

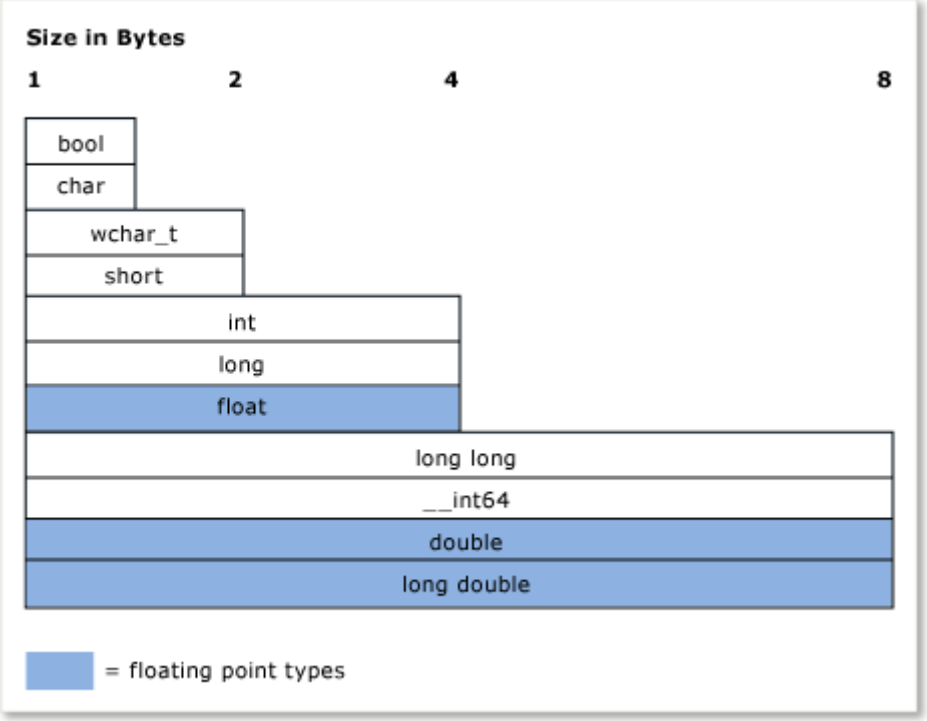
```
var: anio
    Tipo: int
    Valor: 2020
    Direccion de memoria: 140737367406500
    Num bytes: 4

var: PI
    Tipo: float
    Valor: 3.141500
    Direccion de memoria: 140737367406504
    Num bytes: 4

var: letra
    Tipo: char
    Valor: a
    Direccion de memoria: 140737367406516
    Num bytes: 1

var: nombre
    Tipo: char[] - String
    Valor: Elvio Lado
    Direccion de memoria: 140737367406611
    Num bytes: 11
```

Obsrve que no se esta imprimiendo el valor de las variables, sino las direcciones de memoria donde los datos estan guardados (se muestran como valores decimales, aunque son realmente binarios). Así también se observa de que tamaño es cada uno de ellos gracias a la función `sizeof()` . Hay algunas cosas a considerar:

- Un int y un float ocupan el mismo espacio (4 bytes), pero almacenan los datos de forma diferente.
- Un int y un char son datos ambos numéricos (sobre esto más adelante), pero un char tiene el tamaño necesario para guardar un letra, o sea 1 byte.
- Un string tendrá un numero variables de bytes en función de cuantos caractares lo conforman, mas uno adicional (llamado el caracter NULL).

Los tipos de datos tienen las siguentes especificaciones de uso en memoria:

Tipo de datos	Tamaño	Rango	Descripción
char	1 byte	-127 a 128	Caracter
int	2 o 4 bytes	-32,768 a 32,767 o -2,147,483,648 a +2,147,483,647	Entero
float	4 bytes	1.2e-38 a 3.4e38	Real



rand(): Generación de numeros aleatorios

Se pueden generar números aleatorios (realmente *pseudoaleatorios*) utilizando la función `rand()` de la biblioteca `stdlib.h` . La función `rand()` generá valores enteros entre 0 y un número muy grande alamcenado en un *macro*, que es una etiqueta que guarda un valor llamada `RAND_MAX` (distinga entre variables y macro: la última no tiene un valor almacenado en memoria y por lo tanto tampoco una dirección asociada). Este valor dependerá del compilador. Solo recuerde que es muy grande.

Para generar valores enteros en un rango especificado `[a, b>` se puede utilizar la instrucción:

```
a + rand() % (a + b - 1)
```

Para generar valores reales enteros en el mismo rango, se puede utilizar la instrucción:

```
(b - a) * (float)rand() / RAND_MAX + a
```

In [7]:

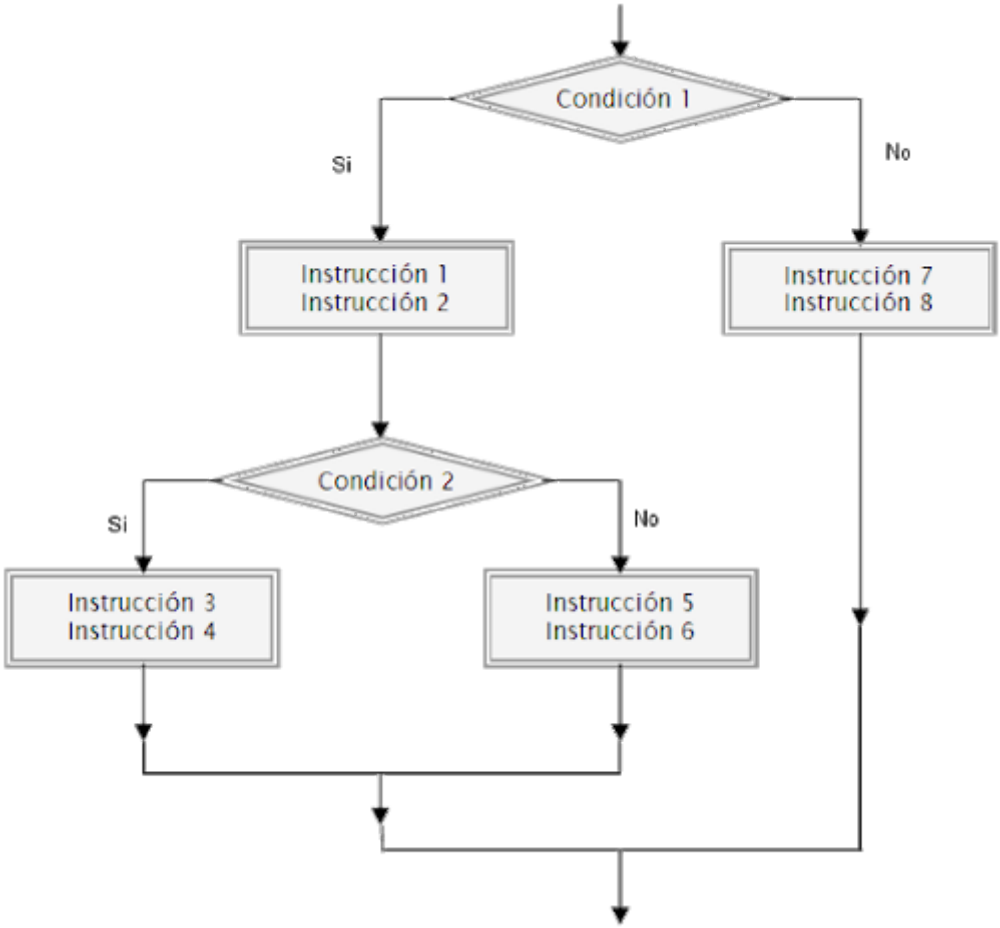
```c
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <time.h>

int main()
{
    srand(time(NULL));  // rand(): [0 - RAND_MAX>

    int dado;

    // a + rand() % (a + b - 1)
    dado = 1 + rand() % 6; printf("Dado: %d\n", dado);
    dado = 1 + rand() % 6; printf("Dado: %d\n", dado);
    dado = 1 + rand() % 6; printf("Dado: %d\n", dado);
    dado = 1 + rand() % 6; printf("Dado: %d\n", dado);
    dado = 1 + rand() % 6; printf("Dado: %d\n\n", dado);

    float nota;

    // (b - a) * (float)rand() / RAND_MAX + a
    nota = (20 - 5) * (float)rand() / RAND_MAX + 5; printf("Nota: %.1f\n", nota);
    nota = (20 - 5) * (float)rand() / RAND_MAX + 5; printf("Nota: %.1f\n", nota);
    nota = (20 - 5) * (float)rand() / RAND_MAX + 5; printf("Nota: %.1f\n", nota);
    nota = (20 - 5) * (float)rand() / RAND_MAX + 5; printf("Nota: %.1f\n", nota);
    nota = (20 - 5) * (float)rand() / RAND_MAX + 5; printf("Nota: %.1f\n", nota);

    return 0;
}
```

```
Dado: 6
Dado: 4
Dado: 6
Dado: 2
Dado: 5

Nota: 14.1
Nota: 9.3
Nota: 11.4
Nota: 15.8
Nota: 13.4
```

Note que el script anterior se han simuado 5 lanzamientos de un dato y se han generado 5 notas de estudiantes en un rango especificado. En un código no tiene sentido repetir lineas iguales: estas se agrupan en un *lazo de repetición*, pero volveremos luego a eso.

if y else: bloques condicionales

En C se puede especificar cual será el flujo de las operaciones de un programa según el resultado de la obervación de ciertas condiciones de operación con las instrucciones `if y else` combinadas con operaciones lógicas y de relación. Esto se entiende mucho mejor con un esquema gráfico del flujo de información en la ejecución de un script llamado *diagrama de flujo*.



Si se siguen las flechas del diagrama se puede observar que el flujo de la información seguirá caminos diferentes en función de los resultados de las operaciones condicionales (esquematizadas como rombos con dos salidas, *true* o *false*). Esto se lográ con la instrucción `if` que tiene la sigiente estructura:

```
if (condicion_verdadera) {
    // Realizar las operaciones en este bloque
}
```

Se puede considerar el caso que se tengan dos grupos de instrucciones a realizar, tanto para el caso que una condición se cumple como que no sea asi. En este caso se tiene:

```
if (condicion_verdadera) {
    // Realizar las operaciones en este bloque
} else {
    // Realizar las operaciones en caso la condicion sea falsa
}
```

Se pueden tener *if anidados*, esto es tener una operación `if` dentro de otra de forma indefinida de la forma:

```
if (condicion_1) {
    if (condicion_2) {
        if (condicion_3) {
            // Bloque de instrucciones...
        }
     } else {
         // Bloque se instrucciones en caso la condicion 1 sea falsa
     }
}
```

Como puede verificar, aunque este código es valido, no resulta légible, ni fácil de mantener ni de entender. Siempre es preferible utilizar operaciones lógicas AND (&&) y OR (||) para conectar las condiciones. Por ejemplo, lo anterior se puede convertír en:

```
if (condicion 1 && condicion 2 && condicion 3) {
    // Bloque de instrucciones
}

if (!condicion_1) {
    // Bloque de instrucciones en caso la condicion 1 sea falsa
}
```

Se puede anidar un `if` luego de un `else` de tal forma que se tenga otras opciones de condicion y se tendrá un código más ordenado aun. Por ejemplo:

```
if (condicion 1 && condicion 2 && condicion 3) {
    // Bloque de instrucciones
} else if (!condicion_1) {
    // Bloque de instrucciones en caso la condicion 1 sea falsa
}
```

In [8]:

```c
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <time.h>

int main()
{
    int mes = 3;

    if (mes < 1 || mes > 12) {
        printf("El mes ingresado no es valido\n");
    } else {
        // El mes es valido
        if (mes <= 6) {
            printf("El mes %d esta en el primer semestre\n", mes);
        }

        if (mes % 2 == 0) {
            printf("El mes %d es un mes de numero par\n", mes);
        }

         /*
        if (mes <= 6) {
            if (mes % 2 == 0) {
                printf("El mes %d es un mes par y pertenece al primer semestre\n", me
s);
            } else {
                printf("El mes %d no es par y pertenece al primer semestre\n", mes);
            }
        } else {
            if (mes % 2 == 0) {
                printf("El mes %d es un mes par y pertenece al segundo semestre\n", me
s);
            } else {
                printf("El mes %d no es par y pertenece al segundo semestre\n", mes);
            }
        }
         */

        if ((mes <= 6) && (mes % 2 == 0)) {
            printf("El mes %d es un mes par y pertenece al primer semestre\n", mes);
        }

        if (!(mes <= 6) && (mes % 2 == 0)) {
            printf("El mes %d es un mes par y pertenece al segundo semestre\n", mes);
        }

        if ((mes <= 6) && !(mes % 2 == 0)) {
            printf("El mes %d es un mes impar y pertenece al primer semestre\n", mes);
        }

        if (!(mes <= 6) && !(mes % 2 == 0)) {
            printf("El mes %d es un mes impar y pertence al segundo semestre\n", mes);
        }
    }

    return 0;
}
```

```
El mes 3 esta en el primer semestre
El mes 3 es un mes impar y pertenece al primer semestre
```

Gracias el los bloques condicionales ya podemos confeccionar un programa que realice algo interesante. Por ejemplo, un script que calcule el Indice de Masa Corporal (IMC) de un paciente. La formula y la tabla de valores la puede encontrar en el articulo de wkipedia (https://es.wikipedia.org/wiki/%C3%8Dndice_de_masa_corporal).

In [ ]:

```c
#include <stdlib.h>
#include <string.h>
#include <time.h>

int main()
{
    // Declaracion de variables
    int peso = 0;
    float altura = 0.0, IMC = 0.0;
    char sexo = ' ';

    printf("Ingrese sexo [M/F]: ");
    scanf("%c", &sexo);
    printf("Ingrese su peso [kg]: ");
    scanf("%d", &peso);
    printf("Ingrese altura [m]: ");
    scanf("%f", &altura);

    IMC = peso / (altura * altura);

    printf("Paciente:\n\t* Sexo: %c\n\t* Peso: %d kg\n\t* Altura: %.2f\n\t* IMC: %.2f\n
",
           sexo, peso, altura, IMC);

    /* Rangos IMC:
        - Bajo peso: < 18.5
        - Normal: [18.5 - 24.99]
        - Sobrepeso: >= 25.00
        - Obesidad: >= 30.00
    */
    if (IMC < 18.5) {
        printf("\t* Estado: Bajo Peso\n");
    } else if(IMC < 25.00) {
        printf("\t* Estado: Normal\n");
    } else if (IMC < 30.00) {
        printf("\t* Estado: Sobrepeso\n");
    } else {
        printf("\t* Estado: Obesidad\n");
    }

    return 0;
}
```

switch case: cuando se evaluan casos exactos

En muchos casos las evalaciones del tipo:

```
if (num == 1) {
    // Hacemos algo...
} else if (num == 2) {
    // Hacemos otra cosa...
} else if (num == 3) {
    // Hacemos otra cosa diferente...
}
```

en donde se revisa la condicion de que la misma variable tome valores distintos, no suele ser muy práctico el utilizar la instrucción `if` . Para este caso es más cómodo utilizar un bloque `switch case` :

```
switch (num) {
    case 1:
        // Hacemos algo...
        break;
    case 2:
        // Hacemos otra cosa...
        break;
    case 3:
        // Hacemos otra cosa diferente...
        break;
    default:
        // Hacemos algo en otro caso cualquiera
 }
```

Lo importante es recordar la inclusión de `break` al terminar cada caso (a excepción del último o en `default`) ya que por la forma como esto funciona, todos los casos son evaluados desde arriba hacia abajo. La instrucción `break` termina la evaluación del bloque. Es importante recordar que los casos son valores y no condiciones.

Por ejemplo, en el siguiente script se tiene un programa que imprime el mes según el número del mes. Editando el bloque `switch case` para que cada caso ocupe una línea se obtiene un código mas ordenado.

In [ ]:

```c
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <time.h>

int main()
{
    int mes;

    printf("Ingrese el mes: [1-12]: "); scanf("%d", &mes);

    switch (mes) {
        case 1: printf("Enero\n"); break;
        case 2: printf("Febrero\n"); break;
        case 3: printf("Marzo\n"); break;
        case 4: printf("Abril\n"); break;
        case 5: printf("Mayo\n"); break;
        case 6: printf("Junio\n"); break;
        case 7: printf("Julio\n"); break;
        case 8: printf("Agosto\n"); break;
        case 9: printf("Setiembre\n"); break;
        case 10: printf("Octubre\n"); break;
        case 11: printf("Noviembre\n"); break;
        case 12: printf("Diciembre\n"); break;
    }

    return 0;
}
```

```
Ingrese el mes [1-12]: 2
Febrero
```

La conducta del `switch case` de evaluar todas las condiciones (y por lo tanto de requerir la instrucción `break`) resulta de utilidad cuando hay varios casos que estan asociados a los mismos resultados. Por ejemplo:

In [ ]:

```c
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <time.h>

int main()
{
    int mes;

    printf("Ingrese el mes: [1-12]: "); scanf("%d", &mes);

    switch (mes) {
        case 2:
            // Y si es bisiesto???
            printf("El mes tiene 28 dias\n");
            break;
        case 1:
        case 3:
        case 5:
        case 7:
        case 8:
        case 10:
        case 12:
            printf("El mes tiene 31 dias\n");
            break;
        case 4:
        case 6:
        case 9:
        case 11:
             printf("El mes tiene 30 dias\n");
             break;
        default:
            printf("El mes ingresado no es valido\n");
    }

    return 0;
}
```

```
Ingrese el mes: [1-12]: 8
El mes tiene 31 dias
```

while: Lazo de repetición controlado por una condición

Si debo de darle instrucciones a un robot trepador de escaleras para que suba por las escaleras hasta el siguiente piso, ¿cuál será la instucción que debo de darle? No será "sube hasta el siguiente piso" pues no conoce la definición de "piso", solo de la escalones.

Entonces...

Un caso será la de una instrucción repetitiva controlada por una condición, del tipo: "miesntras haya un escalón al frente, sube el escalón". Como observa en la instrucción solo esta presente la palabra "escalón" y la condición "mientras haya escalón". Esto se puede construír en un código con la instrucción `while` :

```
while (haya_escalon) {
    subir_escalon
}
```

De esto debemos sacar dos conclusiones:

- Debe de haber un escalón inicial al frente porque sino nunca subira (o sea, la condicion debe ser verdadera al inicio).
- Debe de cambiar esta condición en algún momento para poder detener el lazo o de lo contrario entraremos en un lazo infinito (es decir, una escalera con un número infinito de escalones).

In [10]:

```c
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <time.h>

int main()
{
    int num, cont = 1, n_valores = 10;

    srand(time(NULL));

    while (n_valores > 0) {
        printf("Dado %2d: %d\n", cont, 1 + rand() % 6);

        n_valores--;
        cont++;
    }

    return 0;
}
```

```
Dado  1: 2
Dado  2: 4
Dado  3: 2
Dado  4: 4
Dado  5: 1
Dado  6: 5
Dado  7: 1
Dado  8: 5
Dado  9: 4
Dado 10: 1
```

En el script anterior se lanzan 10 dados y se va mostrando el número de dado lanzado. Siga el flujo de la información para entender como funciona y en que momento el lazo de detiene.

for: Lazo de control controlado por el número de iteraciones

Otra forma de instruír a nuestro robot para que suba los escalones será "sube y cuenta los escalones que vas subiendo hasta que la cuenta llegue a 16". De esta forma tenemos un lazo en el que hemos especificado el número de repeticiones o *iteraciones*. Esto se contruye con un lazo `for` de la forma:

```
for (int escalones=0; i<16; i++) {
    subir escalon...
}
```

Observe el código: en este lazo de inicializa y declara una varialble entera `i` (¡esto solo se puede hacer en la versión del compilador C11!. En compiladores más antiguos, debe de inicializar la variable `int i;` fuera del bloque) que solo existirá dentro del bloque del lazo `for` (más sobre eso luego). El proceso sigue de la siguiente manera:

```
i = 0.
i que es igual a 0 es menor que 16? Si. Entonces subir escalones y al terminar i
ncrementamos el valor de i a 1
i que es igual a 1 es menor que 16? Si. Entonces subir escalones y al terminar i
ncrementamos el valor de i a 2
.
.
.
i que es igual a 15 es menor que 16? Si. Entonces subir escalones y al terminar
 incrementamos el valor de i a 16
i que es igual a 16 es menor que 16? No. Entonces hemos terminado.
```

Siga la cuenta y verá que han sucedido 16 iteraciones (la iteración 0, 1, 2, 3, .. 15) y por lo tanto ha subido 16 escalones. Siga esta regla que le ayudaraá mucho:

- Inicie la varaiable de control con 0
- Indique con i < n, donde n será el número de iteraciones que quiere realizar
- Utilice i++ o ++1. Es indistinto.

Se pueden construír lazos más complejos e incluso con varias variables de control de forma simultanea:

```
for (int i=0, int j=10; i < 10, j > 25; i++, j--) {
    // Varias operaciones...
}
```

In [11]:

```c
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <time.h>

int main()
{
    for (int i=0; i<10; i++) {   // int i : C11
        printf("Dado %2d: %d\n", i+1, 1 + rand() % 6);
    }

    return 0;
}
```

```
Dado  1: 2
Dado  2: 5
Dado  3: 4
Dado  4: 2
Dado  5: 6
Dado  6: 2
Dado  7: 5
Dado  8: 1
Dado  9: 4
Dado 10: 2
```

Igual que en el caso anterior, este script lanza 10 dados bajo en control de un lazo `for` .

while, for... ¿cuándo usar cuál?

Millones de estudiantes siempre se hacen esta pregunta... y la respuesta es que es a gusto de cada uno. Aunque se puede tener una regla que ayuda a tener alguna referencia.

- Cuando se sabe el número de iteraciones a realizar, for
- Cuando no se sabe el número de iteraciones a realizar, while

Esto ultimo puede resultar confuso, pero vamos con un ejemplo:

In [12]:

```c
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <time.h>

int main()
{
    int num = 1000;

    while (num > 1) {
        printf("%d / 2 = ", num);
        num /= 2;   // num = num2 / 2
        printf("%d\n", num);
    }

    return 0;
}
```

```
1000 / 2 = 500
500 / 2 = 250
250 / 2 = 125
125 / 2 = 62
62 / 2 = 31
31 / 2 = 15
15 / 2 = 7
7 / 2 = 3
3 / 2 = 1
```

El programa anterior va dividiendo un número y muestra los resultados de esta operación hasta que el resultado es 1. ¿Cuantas iteraciones se deben realiar? Esto dependerá de el número del que partimos al inicio. Por eso en este caso es prefereible utilizar un lazo `while` que in lazo `for` (¿se puede resolver con un lazo `for` ? Si, pero resultará en un código poco legible).

do while: cuando la condición viene despues

Hay una pequeña sutileza... la instrucción a nuestro robot puede ser "sube un escalón hasta que ya no haya un escalón". Comparela con "mientras haya un escalón al frente, sube el escalón". En el primer caso la condición esta *al final*, lo que quiere decir que mas le vale que haya un escalón al frente porque intentará subirlo antes de considerar si hay más escalones. Esto hace un lazo `do while` : es un lazo `while` donde la condición se verifica después de cada iteraciones y no antes:

```
do {
    subir escalon...
} while (haya escalon);
```

Esto puede ser de utilidad cuando se tomar una decisión luego de pedir un dato, ya que no se puede decír antes pues no se tiene el dato, forzando al usuario a volver a intentar el ingreso:

In [ ]:

```c
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <time.h>

int main()
{
    int num;

    do {
        printf("Ingrese un numero par positivo: ");
        scanf("%d", &num);
    } while (num % 2 != 0);

    printf("Gracias\n");

    return 0;
}
```

```
Ingrese un numero par positivo: 5
Ingrese un numero par positivo: 6
Gracias!
```

Observe que la petición del número con `scanf` esta dentro del lazo `do while` y una vez que se ha ingresado el valor se verifica si es par. Si es impar el lazo se sostendrá y por lo tanto se pedirá nuevamente que ingrese un número.

Se puede realizar lo mismo con un lazo `while` donde se inicializa la condición para que sea `true` inicialmente (inicializar la variable con `1` , por ejemplo). Es por eso que muchos programadores no suelen utilizar el lazo `do while` pues agrega una complicación innecesaria.

Lazos anidados

Se pueden anidar lazos y conseguir algunos códigos realmente complicados (innecesariamente complicados). Condidere el siguente ejemplo:

In [13]:

```c
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <time.h>

int main()
{
    int n_mul3 = 0, val;

    srand(time(NULL));

    // 10 valores aleatorios multiplos de 3
    while (n_mul3 < 10) {

        do {
            val = 1 + rand() % 100;
        } while (!(val % 3 == 0));

        printf("Val %2d: %d\n", n_mul3+1, val);
        n_mul3++;
    }

    return 0;
}
```

```
Val  1: 18
Val  2: 3
Val  3: 15
Val  4: 42
Val  5: 24
Val  6: 66
Val  7: 33
Val  8: 57
Val  9: 48
Val 10: 21
```

Hay un lazo `while` que encierra un lazo `do while` que genera valores y que se mantendrá generando valores hasta que este sea múltiplo de 3 (`while (!(val % 3 == 0))` , donde se romperá el lazo interno para imprimir el número generado y volver a "dar vueltas" por el lazo `while` externo luego de contabilizar el número de valores generados. El lazo `while` esta verificando la condición del número de valores aleatorios múltiplos de 3 generados y cuando llegan a 10 el lazo externo se rompe y el programa se detiene.

Uf....

Esto es un bonito ejercicio pero denota una falta de conocimiento tanto de aritmética como se programación. Primero, porque todo número múltiplo de 3 tiene la forma 3n, y porque se que quiere 10 números aleatorios por lo que puede utilizar un lazo `for` :

In [16]:

```c
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <time.h>

int main()
{
    for (int i=0; i<10; i++) {
        printf("Val %2d: %d\n", i+1, 3 * (1 + rand() % 100));
    }

    return 0;
}
```

```
Val  1: 252
Val  2: 261
Val  3: 234
Val  4: 48
Val  5: 282
Val  6: 108
Val  7: 261
Val  8: 279
Val  9: 150
Val 10: 66
```

break y continue: cuando se cambian las condiciones al vuelo

El caso anterior es una buena solución, a menos que se quiera tener múltiplos de 3 pequeños. Podemos modificar el código incluyendo dos instrucciones especiales: `continue` y `break` .

- `continue` detiene una iteración y continua con la siguiente.
- `break` detiene el lazo de control por completo.

Revise el siguiente código:

In [20]:

```c
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <time.h>

int main()
{
    int n_mul3 = 0, val;

    srand(time(NULL));

    // 10 valores aleatorios multiplos de 3
    while (1) {
        val = 1 + rand() % 100;

        if (val % 3 != 0) {
            continue;   // Regresa a la linea 13...
        } else {
            printf("Val %2d: %d\n", n_mul3+1, val);
            n_mul3++;
        }

        if (n_mul3 > 10) {
            break;
        }
    }

    return 0;
}
```

```
Val  1: 21
Val  2: 63
Val  3: 63
Val  4: 96
Val  5: 93
Val  6: 87
Val  7: 27
Val  8: 33
Val  9: 12
Val 10: 39
Val 11: 60
```

Aqui tenemos un lazo infinito `while (1)` donde se generá un número aleatorio. Luego, se consulta si el número generado no es múltiplo de 3 con la operación `if (val %3!== 0)` . Si esto es así, sucede la operación `continue` , lo que equivale a cancelar las demás instrucciones y regresar al principio del lazo (a evaluar la condición del lazo `while`). Si el número generado es múltplo de 3 se imprime y se incementa el valor de la variable de control `n_mult3` .

Al final de las instrucciones en el lazo `while` se pregunta si esta variable de control `n_mult3` ha superado el valor de 10. Si es asi sucede la operación `break` y eso rompe el lazo que contiene el bloque de instrucciones (esto es, el lazo `while (1)`) y el programa termina.

Aca suceden varias cosas:

- `continue` controla si se continua con algunas operaciones. Esto puede resultar muy útil.
- `break` rompe el lazo infinito y esto no es un buen código, pues se especifica una condición inicial y luego se niega esa condición. Esto nunca es una buena idea. En términos generales, debe de evitarse el uso de `break` a toda costa y tratar de buscar otra solución que mantenga el código legible.
- AL ejecutar el código se obtiene 11 resultados y no 10 como en el ejemplo anterior. Esto es un caso típico de "error por uno". Esto se debe a la condición `if (n_mult > 3)` en lugar de `if (n_mult >= 3)` . Este error es muy común en programación y hay que estar atento a estos casos.

Proyecto: Impresión del calendario de un mes/año

Queremos un programa que pida al usuario el mes y al año y nos imprima un calendario mensual de la forma:

```
 D   L   M   X   J   V   S
         1   2   3   4   5
 6   7   8   9  10  11  12
13  14  15  16  17  18  19
20  21  22  23  24  25  26
27  28  29  30
```

Para esto lo que necesitamos es saber:

- Cuantos días tiene el mes.
- Que dia empieza el mes (dia 1 del mes en mención)

El número de días del mes lo podemos obtener de una estructra *switch... case* (vamos a obviar el caso de los años bisiestos).

Sobre el 1 del mes, podemos utilizar un algoritmo llamado la Congruencia de Zeller (https://es.wikipedia.org/wiki/Congruencia_de_Zeller) que permite calcular el dia de una fecha. Para el calendario gregoriano tenemos la siguiente fórmula:

$$h = \left(q + \left\lfloor \frac{13(m+1)}{5} \right\rfloor + Y + \left\lfloor \frac{Y}{4} \right\rfloor - \left\lfloor \frac{Y}{100} \right\rfloor + \left\lfloor \frac{Y}{400} \right\rfloor \right) \mod 7$$

donde *m* es el mes del año, *Y* es el año y *q* es el dia (en nuestro caso, siempre será 1).

Hay algunos detalles a considerar: la cuenta de los meses inicia en marzo = 3, abril = 4, ..., enero = 13 y febrero = 14. Estos dos últimos modifican el valor del año pues se considera que pertenecen al año anterior.

El resultado *h* será un valor entre 0 y 6 (observe la operación *mod 7*). Esto se interpreta como sábado = 0, domingo = 1, ..., viernes = 6. Esto no esta de acuerdo con el ordenamiento de nuestro calendario que inicia en el dia domingo, por lo que habrá que hacer unos ajustes.

Respecto a los meses: pedimos al usuario que ingrese mes y año. Si el mes es enero o febrero, debemos hacer la correccion de 13 y 14 a 1 y 2, junto con decrementar el año:

```
if (mes == 1 || mes == 2) {
    mes += 12;
    anio--;
}
```

Con esto calculamos el valor de *h*, pero debemos hacer la siguiente corrección:

```
 D   L   M   X   J   V   S
[0] [1] [2] [3] [4] [5] [6]

0 (sábado) -> posición en el calendario [6]
1 (domingo)-> posicion en el calendario [0]
2 (lunes)  -> posición en el calendario [1]
.
.
.
6 (viernes) -> posición en el calendario [5]
```

¿Cuál es el patrón? El valor de *h* debe decrementarse para todos los días, con la excepción de 0 que debe de ser igual a 6:

```
if (prim_mes == 0) {
    prim_mes = 6;
} else {
    prim_mes--;
}
```

Esto se puede implementar en una *operacion ternaria*. Cada vez que se tiene un *if...else* se puede escribir la siguente instrucción:

```
if condicion ? caso_true: caso_false
```

Entonces nos queda:

```
if (prim_mes == 0) ? prim_mes = 6: prim_mes--;
```

In [ ]:

```c
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <time.h>

/*
Calendario mensual
*/

int main()
{
    int mes, anio, prim_mes, dias_mes;
    char barra;

    printf("Ingrese el mes y el anio [mm/aaaa]: ");
    scanf("%d%c%d", &mes, &barra, &anio);

    // Cuantos dias tiene el mes ingresado?
    switch (mes) {
    case 2: dias_mes = 28; break;
    case 1: case 3: case 5: case 7: case 8: case 10: case 12: dias_mes = 31; break;
    case 4: case 6: case 9: case 11: dias_mes = 30; break;
    }

    // Congruencia de Zeller (https://es.wikipedia.org/wiki/Congruencia_de_Zeller)
    // meses: 3=marzo, 4=abril, 5=mayo,... 13=enero, 14=febrero
    //   nota: los meses enero y febrero son del aÃ±o anterior
    //         dia de la semana: 0=sabado, 1=domingo, 2=lunes,... 6=viernes

    if ((mes == 1) || (mes == 2)) {
        mes += 12;
        anio--;
    }

    prim_mes = (1 + ((13 * (mes + 1)) / 5) + anio + (anio / 4) - (anio / 100) + (anio /
400)) % 7;

    // Ajustar el numero de dia al calendario [L, M, X, J, V, S]
    // De: SAB(0), DOM(1), LUN(2), 3, 4, 5, 6
    // A : SAB(6), DOM(0), LUN(1), 2, 3, 4, 5
    // Ternary if...
    prim_mes == 0 ? prim_mes = 6: prim_mes--;

    // Imprimir el encabezado del calendario
    printf("  D  L  M  X  J  V  S\n");

    // Se imprimen los espacios en blanco antes del dia 1
    for (int i=0; i < prim_mes; i++) {
        printf("   ");
    }
    // Se imprimen los dias del calendario desde el 1 del mes
    for (int i=1; i <= dias_mes; i++) {
        printf("%3d", i);
        // ...y se salta a una nueva linea al llegar al fin de semana
        if ((i + prim_mes) % 7 == 0) printf("\n");
    }

    return 0;
}
```

Esto no esta concluido. Algunas cosas que le puede agregar:

- Que imprima el nombre del mes en la cabecera
- Que valide el dato del mes y vuelva a pedir la información
- Que muestre el numero de dias correcto para febrero de un año bisiesto.

Aprenda de los errores. Esa es la guía para ser un mejor programador.