Programación II

Programando en C

Lic. Federico Severino Guimpel
LCC - FCEIA - UNR

Switch

En C también contamos con la estructura switch.

El uso de la misma es bastante acotado ya que se compara el valor de una variable, que debe tener tipo entero o caracter, contra una serie de constantes. Después de encontrar una coincidencia se ejecuta las sentencias asociadas al valor.

Switch

La sintaxis es:

```
switch (variable) {
     case constante1:
          sentencias1
          break;
     case constante2:
          sentencias2
          break;
     case constante3:
          secuencias3
          break;
     ...
     default:
          sentenciasDefault
}
```

Switch

Se ejecuta la sentencia **default** si no coincide ninguna constante con la variable, esta última es opcional.
Cuando se encuentra una coincidencia, se ejecutan las sentencias asociadas con el case hasta encontrar la sentencia **break** con lo que sale de la estructura switch.

Switch

Aquí tenemos un ejemplo donde en caso de que la variable `i` tenga los valores 2, 3 o 5 ejecuta las mismas sentencias.

```
switch (i) {
    case 0:
        printf("El numero es 0");
        break;
    case 2:
    case 3:
    case 5:
        printf("El numero es primo");
        break;
    default:
        printf("El numero no es primo");
        break;
}
return 0;
```

Arrays

En C no existe el tipo String. La única forma, por el momento, que tenemos de definir una variable que almacene un String es declarando un `array` de `char`.

¿Qué es un `array`, o `arreglo`, en español? Es similar al concepto de las listas de Python: una estructura de datos secuencial que se accede por un índice. Sin embargo, tiene algunas (muchas) limitaciones si las comparamos entre sí.

Arrays

Lo primero que debemos saber es que al declarar un `array` en C se debe indicar el tamaño. Dicho tamaño, en general, no puede modificarse.
Otra característica importante es que todos los elementos de un `array` son del mismo tipo.
Veamos el siguiente código:

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

    int a[10];
    int b[] = {0, 1, 2};

    char c[10] = "hola";


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

    c[0] = 's';

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

    return 0;
}
```

Arrays

Las primeras líneas corresponden a declaraciones de `arrays` y podemos notar lo siguiente:

- La primera y segunda declaran `arrays` de `ints` mientras que la tercera es un `array` de `chars`.
- Tanto `a` como `c` son `arrays` que están siendo inicializados con un tamaño de 10.
- La notación para definir `b` indica, implícitamente, el tamaño que va a tener el array. `b` se inicializa como un `array` de `ints` de tamaño 3 y con valores iniciales 0, 1 y 2 en sus respectivas posiciones.

```
int a[10];
int b[] = {0, 1, 2};

char c[10] = "hola";
```

La salida, como nos podemos imaginar es:

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

ya que se está modificando el primer caracter de la cadena que se almacenó en el array.

Strings

Una característica importante que tienen los `arrays` en C es que no podemos recuperar el tamaño del mismo. Es decir, no existe una función que nos retorne el tamaño del `array`.

Sólo en el caso de `arrays` de `chars` se puede saber la longitud del String almacenado. Esto no es lo mismo que la capacidad definida en la declaración `array`. Veamos un ejemplo que nos permita entender cómo se hace esto.

Strings

Analicemos el siguiente código

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

    char c[8] = "hola";

    printf("la longitud de la cadena es: %ld\n", strlen(c));
    return 0;
}
```

Strings

Podemos notar que:

- Se usa una librería `string.h` la cual tiene las funciones propias de los Strings.
- strlen es una función que toma un `arrays` de `chars` y retorna la longitud. El tipo de esto es un entero sin signo.

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

    char c[8] = "hola";

    printf("la longitud de la cadena es: %ld\n", strlen(c));
    return 0;
}
```

Al ejecutar, la salida es:

```
fede@DESKTOP-NUTBUG5:/mnt/d/Fede/Facultad/Programacion II/C/Ejemplos$ ./a.out
la longitud de la cadena es: 4
```

Esta función lo que hace no es otra cosa que buscar, desde el inicio del array, un terminador de cadena. Esto es, una marca de fin de cadena. En C esta marca de fin de cadena es el caracter '\0'.
Si miráramos la memoria para ver el contenido del array encontraríamos esto:

index	0	1	2	3	4	5	6	7
character	h	o	l	a	\0			

Una aclaración, de la posición 5 en adelante no sabemos qué dato hay. Como no fue inicializado hay basura.

Entrada estándar

Recordemos la forma en que se usa el `scanf`. Habíamos comentado que se debía poner un & delante de la variable para poder leerla.

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

Esto siempre es así en el scanf salvo cuando la variable es un `array` de `char`:

```
char s[10];
printf("Ingrese un string: ");
scanf("%s", s);
```

El motivo de esto se verá más adelante.

Sentencia for

Tal cual sucede en Python en C existe una sentencia `for` pero su comportamiento es distinto. Veamos por qué.

Recordemos que la sentencia `for`, en Python, sólo servía para iterar sobre alguna colección o estructura de datos.
En cambio en C la sentencia `for` tiene, al igual que el `while` en Python, una condición lógica. Mientras esta sea verdadera se ejecutan las sentencias. Sin embargo, esta no es la única diferencia.

Vamos a ver su sintaxis.

Sentencia for

La sintaxis del bloque `for` es:

```
for (inicialización; condición; incremento) {
    sentencias a ejecutar si condición es true
}
```

Como se puede notar hay 3 bloques (así se los denomina) en la definición del `for`:

- `inicialización`
- `condición`
- `incremento`

Sentencia for

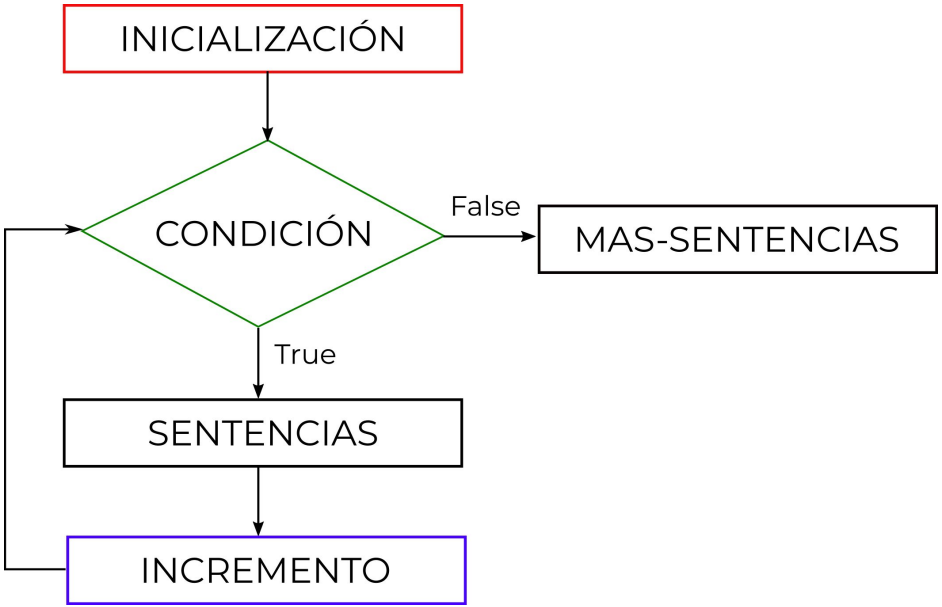
El nombre asociado a cada bloque tiene que ver con el significado de dicho bloque.

El bloque `inicialización` contiene sentencias que sólo se ejecutan una vez previo a verificar la `condición` del `for` por primera vez.

El bloque `incremento` se ejecuta cada iteración, luego de ejecutarse las `sentencias` y, previo a verificar la `condición`.

Sentencia for

Es decir que el diagrama de ejecución del `for` quedaría así:



Vamos a ilustrar esto con algunos ejemplos.

Sentencia for

Dado este programa que usa un bucle `while`.

```c
#include<stdio.h>

int main() {

    int i = 3;

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

    return 0;
}
```

¿Cómo se podría hacer lo mismo con un bucle `for`?

Sentencia for

Nuestra primera aproximación sería esta.

```c
#include<stdio.h>

int main() {

    int i = 3;

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

    return 0;
}
```

Se puede ver que:

- el bloque de `inicialización` está vacío
- el bloque `condición` es igual al que tenía el `while`
- el bloque de `incremento` está vacío

Pero, ¿esto funciona? ¡Sí! Cualquiera de los tres bloques puede estar vacío. De hecho, estas estructura serían equivalentes:

```c
for( ;condición; ){
    sentencias
}
```

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

Sentencia for

Dijimos que cualquiera de los tres bloque puede estar vacío. ¿El bloque `condición` también? Sí, en general no se usa vacío. Un bloque `condición` vacío da lugar a un `for` infinito, ya que la condición se asume siempre true en ese caso.

Ahora bien, ¿si usamos los bloques del `for` para poner parte del código que contiene nuestro programa? Vamos a analizarlo.

Sentencia for

Estas dos propuestas de programa, ¿en qué cambian?

```c
#include<stdio.h>

int main() {

    int i;

    for(i = 3; i>0; ){
        printf("x = %d\n", i);
        i--;
    }

    return 0;
}
```

```c
#include<stdio.h>

int main() {

    for(int i = 3; i>0; ){
        printf("x = %d\n", i);
        i--;
    }

    return 0;
}
```

Como la variable `i` sólo se usa dentro del `for` se la puede definir e inicializar en él. Esto hace que la variable `i` no exista fuera del mismo.

Sentencia for

Este sería el típico ejemplo de un bloque `for`.

```c
#include<stdio.h>

int main() {

    for(int i = 3; i>0; i--){
        printf("x = %d\n", i);
    }

    return 0;
}
```

El bloque de `incremento` modifica una variable que afecta la `condición`.

Sentencia for

¿Se pueden poner varias sentencias en el bloque de `incremento` y en el de `inicialización`? Si es así, ¿cómo se hace? y, más importante aún, ¿hay sentencias que no se pueden poner?

Bueno, vamos a ir contestando por partes estas preguntas. La respuesta a la primera es que sí. Se pueden poner más de una sentencia en cualquiera de esos bloques. Veamos un par de ejemplos.

Sentencia for

Analicemos esta variante del programa anterior. El dato se ingresa por teclado en lugar de darle un valor explícito.

```c
int main() {

    int i;
    printf("ingrese un valor entero: ");
    scanf("%d", &i);

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

    return 0;
}
```

Antes de continuar quiero hacerles un comentario: esto es a modo de mostrar las posibilidades que la estructura `for` da en C. No es una recomendación de uso. Las variantes que están a continuación pueden ser más confusas que este programa.

Sentencia for

Una primera variante sería esta:

```c
#include<stdio.h>

int main() {

    int i;

    for(printf("ingrese un valor entero: "), scanf("%d", &i); i>0; ){
        printf("x = %d\n", i);
        i--;
    }

    return 0;
}
```

En el bloque de `inicialización` estamos poniendo dos sentencias que se ejecutaban previo al `for`, que es el comportamiento que este bloque tiene. Noten que se utiliza la `,` para separar los mismas debido a que el `;` tiene otro significado en el `for`.

Sentencia for

¿Podríamos poner esto?

```c
#include<stdio.h>

int main() {

    for(int i, printf("ingrese un valor entero: "), scanf("%d", &i); i>0; ){
        printf("x = %d\n", i);
        i--;
    }

    return 0;
}
```

Si compilamos se obtiene esto:

```
D:\Fede\Facultad\Programacion II\C\Ejemplos>gcc -Wall ejemplo1.c
ejemplo1.c: In function 'main':
ejemplo1.c:8:20: error: expected declaration specifiers or '...' before string constant
  for(int i, printf("ingrese un valor entero: "), scanf("%d", &i); i>0; ){
                    ^~~~~~~~~~~~~~~~~~~~~~~~~~~
```

Sentencia for

¿Por qué sucede esto?

Esta es la limitación que se puede marcar sobre qué sentencias poner en estos bloques. La declaración de variables ya usa la `,` para permitir declarar variables de un mismo tipo. Esto hace que el compilador cuando encuentra la `,` espere otro identificador. En su lugar, en este caso, hay una sentencia `printf`.

```
D:\Fede\Facultad\Programacion II\C\Ejemplos>gcc -Wall ejemplo1.c
ejemplo1.c: In function 'main':
ejemplo1.c:8:20: error: expected declaration specifiers or '...' before string constant
  for(int i, printf("ingrese un valor entero: "), scanf("%d", &i); i>0; ){
                    ^~~~~~~~~~~~~~~~~~~~~~~~~~~
```

Sentencia for

Continuando, otra variante que podría tener sería:

```c
#include<stdio.h>

int main() {

    int i;

    for(printf("ingrese un valor entero: "), scanf("%d", &i); i>0; i--){
        printf("x = %d\n", i);
    }

    return 0;
}
```

Donde, como mostramos antes, se pone la sentencia de decremento de la variable.

Pero, si pusimos esta sentencia, ¿no se podría poner también el `printf`?

Sentencia for

La respuesta es que sí.

```c
#include<stdio.h>

int main() {

    int i;

    for(printf("ingrese un valor entero: "), scanf("%d", &i); i>0; printf("x = %d\n", i), i--){
    }

    return 0;
}
```

El orden de estas sentencias es el mismo orden en el que estaban dentro del `for`.

Sentencia for

La respuesta es que sí.

```c
#include<stdio.h>

int main() {

    int i;

    for(printf("ingrese un valor entero: "), scanf("%d", &i); i>0; printf("x = %d\n", i), i--){
    }

    return 0;
}
```

El orden de estas sentencias es el mismo orden en el que estaban dentro del `for`.

Sentencia for

Dado que el `for` no tiene sentencias, en el cuerpo, se puede poner un `;` al final y obviar las llaves.

```c
#include<stdio.h>

int main() {

    int i;

    for(printf("ingrese un valor entero: "), scanf("%d", &i); i>0; printf("x = %d\n", i), i--);

    return 0;
}
```

Sentencia for

Para finalizar había quedado pendiente la explicación de por qué el operador ++ y -- cambia su precedencia según si se pone a izquierda o a derecha de la variable. Este ejemplo nos lo va a ilustrar:

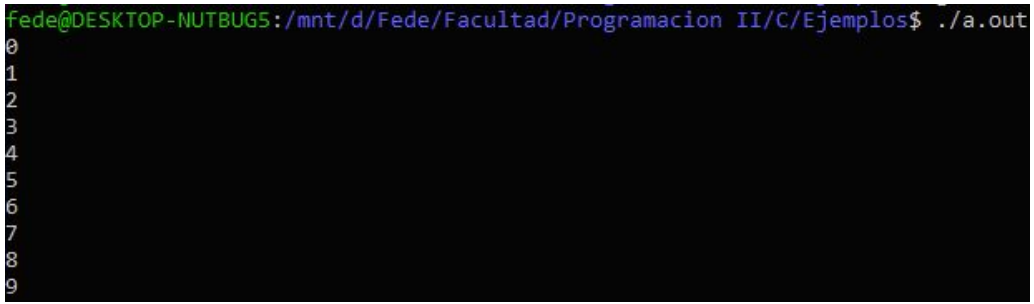
```c
#include <stdio.h>

int main(){
    for(int i=0; i<10;printf("%d\n", i++));
}
```

En la misma sentencia que estamos haciendo uso de la variable se la está incrementando. En este caso usando el operador ++ a derecha.

Sentencia for

El resultado de esta ejecución es:



En el caso de ponerlo a derecha el operador tiene menos precedencia que cualquier otro. Por lo tanto, se incrementa la variable luego de usarla.

Sentencia for

Si en cambio ponemos el operador a izquierda:

```c
#include <stdio.h>

int main(){
    for(int i=0; i<10;printf("%d\n", ++i));
}
```

el operador tiene la mayor precedencia. Es decir, se realiza la operación antes de cualquier otra. Entonces la salida es:

```
fede@DESKTOP-NUTBUG5:/mnt/d/Fede/Facultad/Programacion II/C/Ejemplos$ ./a.out
1
2
3
4
5
6
7
8
9
10
```