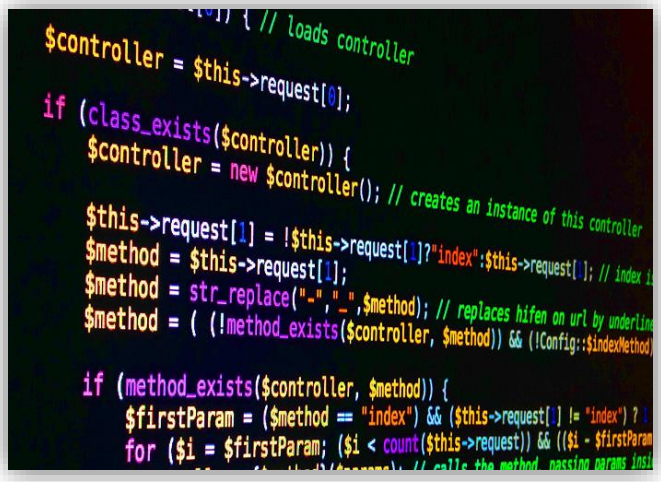


DESARROLLO DE SOFTWARE I

Tecnicatura en Sistemas IT

Docente: Ing. Ezequiel Escobar

CLASE 4

Ciclo exacto, punteros y más

Temario

1. Ciclo exacto FOR
2. Contadores
3. Acumuladores
4. Punteros
5. Vectores (ARRAYS)

1. Ciclo exacto: FOR

Es bastante obvio, pero repasemos el concepto de ciclo. Un ciclo es cierto *periodo temporal* que, una vez finalizado, vuelve a empezar. También se trata de la secuencia de etapas que atraviesa un suceso de características periódicas y del grupo de fenómenos que se reiteran en un cierto orden. Entonces bien: es un periodo. Un periodo se caracteriza por tener un momento inicial y un momento final, es decir, es finito.
Con esta pequeña introducción nos basta para entender el concepto de Ciclo exacto (FOR).
El ciclo FOR es un bucle muy flexible y a la vez muy potente ya que tiene varias formas interesantes de implementarlo, su forma más tradicional es la siguiente:

```
for (/* inicialización */; /* condición */; /* incremento */)
{
      /* código a ejecutar */
}
```

- *Inicialización*: en esta parte se inicia la variable que controla el ciclo y es la primera sentencia que se ejecuta. Sólo se ejecuta una vez ya que solo se necesita al principio del ciclo.

- *Expresión condicional*: esta expresión determina si el ciclo continuará ejecutándose o no. Deberá ser una condición, es decir que deberá utilizar mínimamente algún operador lógico. Además, deberá estar involucrada la variable de control (la variable que se utilizó en la sección de inicialización).

- *Incremento*: es una sentencia que ejecuta al final de cada iteración del ciclo. Por lo general, se utiliza para incrementar la variable con que se inició el ciclo. Luego de ejecutar el incremento, el bucle revisa nuevamente la condición: si es verdadera, tiene lugar una ejecución más del cuerpo del ciclo, si es falsa se termina el ciclo.

Veamos un pequeño ejemplo:

```
for (cantidadVueltas = 0; cantidadVueltas < 25 ;
cantidadVueltas = cantidadVueltas +1)
{
  std::cout << cantidadVueltas << endl;
}
```

Pero el ciclo anterior también lo podemos escribir:

```
for(cantidadVueltas = 0; cantidadVueltas < 25 ;
cantidadVueltas++)
```

```
{
   std::cout << cantidadVueltas << endl;
}
```

Nótese la diferencia en la sentencia de incremento: pasamos de "cantidadVueltas = cantidadVueltas + 1" a "cantidadVueltas++". ¡Claro! Reducen a lo mismo. Regla práctica:

> `variable = variable + 1` es equivalente a `variable++`

También vale:

> `variable = variable - 1` es equivalente a `variable--`

Otro ejemplo

Necesitamos un programa que muestre por pantalla los números pares del 50 al 100. Propongamos dos posibles soluciones.

Solución A

```
int unNumero = 50;

for (unNumero; unNumero <= 100; unNumero = unNumero + 2) {
   cout << unNumero << endl;
}
```

En este caso, incrementamos la variable "unNumero" de dos en dos, para que siempre nos toque los números pares.

Solución B

```
for (unNumero; unNumero <= 100; unNumero++) {
    if (unNumero % 2 == 0) {
            cout << unNumero << endl;
    }
}
```

En esta solución incrementamos en 1 a la variable "unNumero". Esto genera una desventaja: pasamos de iterar 25 veces a iterar 50 veces. Sumado a eso, estamos evaluando por cada iteración si el número es par o no. Con esto queda demostrado que:

No existen soluciones únicas, sino que existen varias, pero cada una tendrá algo mejor (o peor) que la otra.

2. Contadores

Un contador es una *variable numérica* cuyo valor se incrementa o decrementa en una cantidad constante cada vez que se produce un determinado suceso o acción.

Veamos un ejemplo de para qué nos sirve.

Necesitamos un programa que, dado un valor inicial y un valor final (ingresados por teclado por el usuario), cuente la cantidad de números que son divisibles por 5.

Bien, veamos el código:

```
int main()
{
  int numeroIncial;
  int numeroFinal;
  int contador = 0;

  cout << "Ingrese el número inicial" << endl;
  cin >> numeroIncial;
  cout << "Ingrese el número final" << endl;
  cin >> numeroFinal;

  for (int unNumero = numeroIncial; unNumero <= numeroFinal;
unNumero ++) {
      if (unNumero % 5 == 0) {
          contador++;
      }
  }

  cout << "La cantidad de números divisibles por 5 fueron: "
<< contador << endl;
}
```

Si prestamos atención al código, surge la siguiente regla:

> Un contador siempre debe estar inicializado.

3. Acumuladores

Un acumulador es una variable numérica que suma sobre sí misma un conjunto de valores para, de esta manera, tener la suma de todos ellos.

Ejemplo

Supongamos ahora que, dado un valor inicial y un valor final (establecido por el usuario), necesitamos obtener la suma de todos aquellos números que sean divisibles por 5.

```
int main()
{
  int numeroIncial;
  int numeroFinal;
  int acumulador = 0;

  cout << "Ingrese el número inicial" << endl;
  cin >> numeroIncial;
  cout << "Ingrese el número final" << endl;
  cin >> numeroFinal;

  for (int unNumero = numeroIncial; unNumero <= numeroFinal;
unNumero ++) {
       if (unNumero % 5 == 0) {
            acumulador = acumulador + unNumero;
       }
  }

  cout << "La suma de los números divisibles por 5 fue: " <<
acumulador << endl;
}
```

4. Punteros

Los punteros son variables, pero en vez de contener un valor específico de un tipo de dato, contienen las direcciones de memorias de las variables a las que apuntan.
Para obtener o modificar el valor de la variable a la que apuntan se utiliza el operador de indirección (enseguida lo veremos). Los punteros, al ser variables, deben ser declarados como punteros antes de ser utilizados.

Declaración

Los punteros, como toda variable, deben ser declarados antes de ser utilizados. La forma genérica de declarar un puntero es la siguiente:

```
tipoDeDato *nombreVariable;
```

Por ejemplo, si queremos un puntero a un *int*, entonces su declaración tendrá la forma:

int *punteroAnumero;

Operadores

Existen dos operadores a tener en cuenta cuando trabajamos con punteros. El *operador de dirección (&)* que devuelve la dirección de memoria de su operando y el *operador de indirección (*)* que devuelve el valor que contiene el espacio de memoria al cual se está apuntando.
Viéndolo desde otro enfoque, un puntero es únicamente una dirección de memoria (un número) y el asterisco es el que hace la magia de obtener el valor referenciado por dicha dirección.

Ejemplo:

```
int *punteroAnumero = NULL;
int unNumero = 10;
punteroAnumero = &unNumero;
```

Si miramos bien, inicializamos el puntero con un valor nulo. Esto es importante no olvidarlo, ya que, si no inicializamos el puntero con null, estaríamos apuntando a una dirección de memoria "basura".
Sigamos con el ejemplo. Supongamos que la siguiente línea fuera la siguiente:

```
*punteroAnumero = *punteroAnumero + 3;
```

¿Qué pasa con el valor de la variable "unNumero"? ¡También cambia! Ahora "unNumero" pasa a valer 13, como así también lo vale "*punteroAnumero".
Sigamos dos líneas más:

```
punteroAnumero = NULL;
*punteroAnumero = 25;
```

¿Qué sucede ahora? ¡Explota todo! ¿Por qué? Porque ahora punteroAnumero no está apuntando a ninguna dirección, sino que está apuntando a NULL; y, por lo tanto, no puedo guardar un valor.

Vamos con otro ejemplo sencillo:

```
char *apuntador = NULL;
char letra;
unPuntero = &letra;
*unPuntero = 'x';
cout << letra;
```

¿Qué muestra? ¡X!

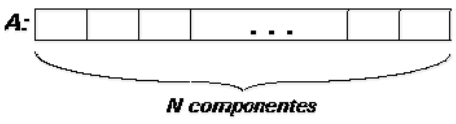
Algunos tips

- ✓ El tipo de dato del puntero debe coincidir con el de la variable cuya posición de memoria apunta.
- ✓ Siempre que queremos usar el valor que contiene un puntero debemos anteponer el asterisco (*) para indicar que usaremos el valor en la posición de memoria apuntada.
- ✓ De no usar el asterisco para el caso anterior, el comportamiento sería impredecible. En este caso, estaríamos haciendo uso de la dirección de memoria en vez del valor almacenado en ésta.
- ✓ Un puntero puede ser de cualquier tipo de dato, inclusive los podemos usar con tipos complejos (los veremos más delante).

5. Vectores – Arrays

Los arrays son ***variables estructuradas***, donde cada elemento se almacena de forma consecutiva en memoria. Los arrays pueden tener múltiples dimensiones.
Decimos entonces que ***un vector es un array unidimensional*** y que está formado por un número "n" de variables simples del mismo tipo que son denominadas "elementos del array". El número de componentes "n" es, entonces, la dimensión del array.

Gráficamente:



Sintaxis

```
tipoDeDato nombreVariable [n];
```

donde: n >= 1

¿Cómo acceder a un elemento?

```
nombreVariable[i];
```

donde: 0 <= i < n

Ejemplo

```
int unVector[4];
```

En la anterior sentencia declaramos un vector de dimensión 4, lo que significa que, como máximo, vamos a poder tener 4 elementos.
Pero… ¿qué pasa si en la siguiente sentencia queremos acceder al primer elemento del vector? ¿Qué hay allí? ¡BASURA!

Es buena práctica que, antes de utilizar un vector, y justo después de su declaración, lo inicialicemos. ¿Cómo hacemos esto?

```
for (i = 0; i< 4; i++){
  unVector[i] = 0;
}
```

¡Claro! Inicializar un vector no es más que "limpiarlo" de basura que podría llegar a haber en ese espacio de memoria. En este caso, como es un vector de ints, le asignamos a cada posición un cero (0).

Otra forma de declarar e inicializar un vector es:

int unVector[4] = {10, 15, 20, 25};

Recordemos: ***si no se inicializa explícitamente el array no se puede estar seguro del valor que tienen los elementos del mismo***.