



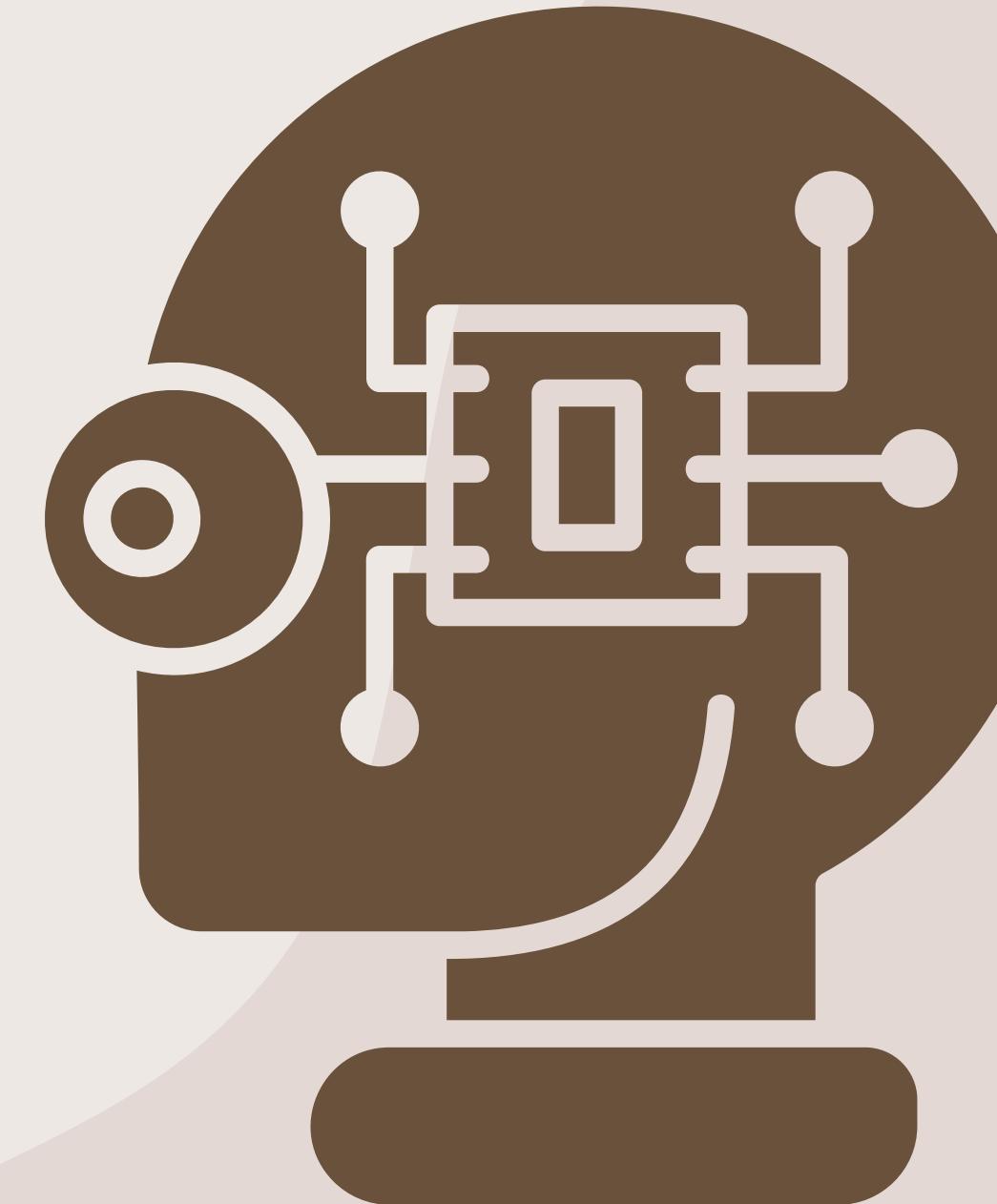
Escuela Superior
de Cómputo

LISTAS SIMPLEMENTE LIGADAS **FORMA CIRCULAR**

Realizado por:
Hector Josue de la Riva Martinez

TABLA DE CONTENIDO

• Introducción	03
• Definición	04
• Importancia	05
• Diferencia entre LSL y LSLC	06
• Ventajas y Desventajas	07
• Inserción de nodos	08
• Modificación de nodos	10
• Mostrar nodos	11
• Buscar nodos	12
• Eliminación de nodos	13
• Aplicaciones	14
• Conclusión	15



INTRODUCCIÓN

Para hablar de listas simplemente ligadas en su forma circular se deberían de conocer los siguientes temas:

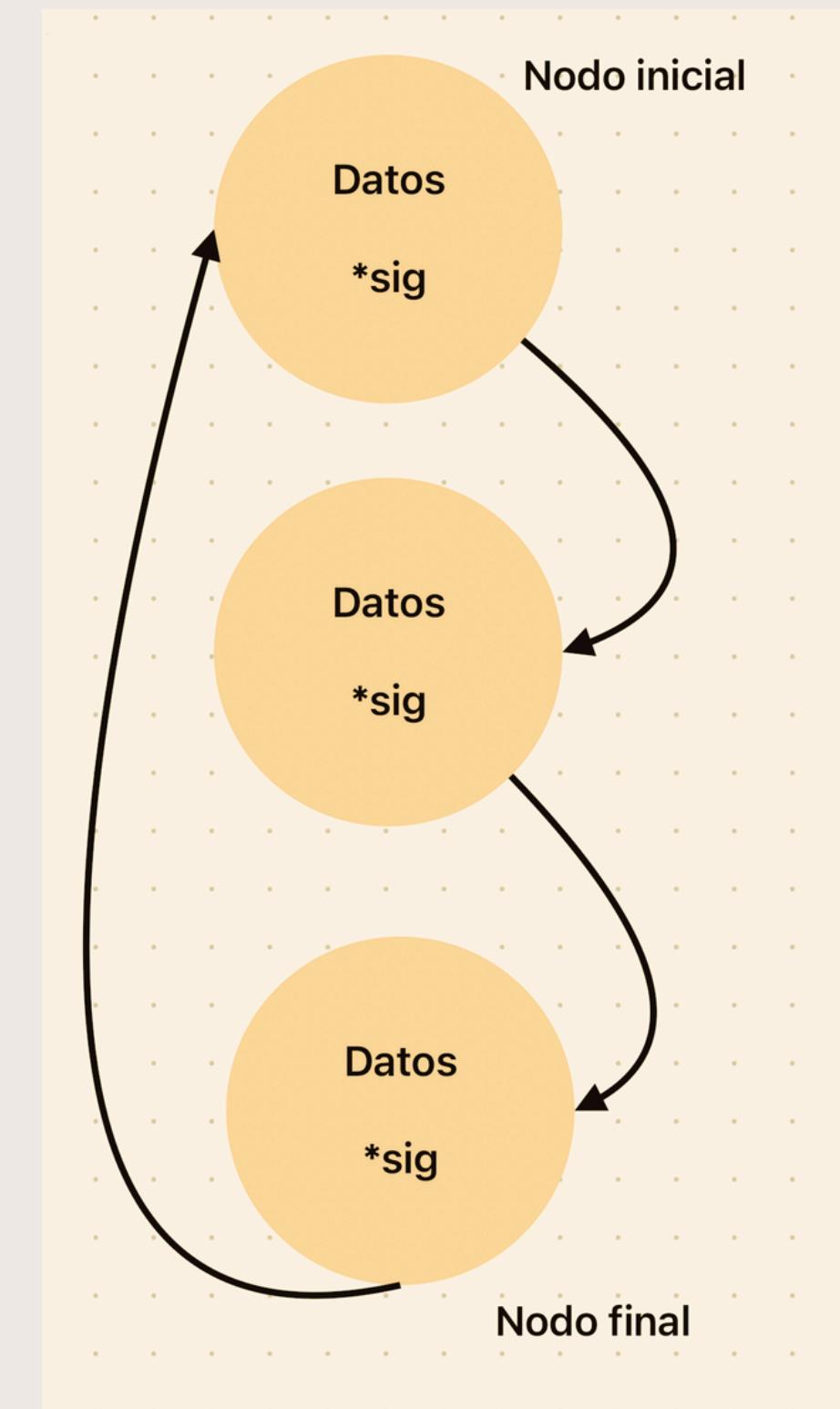
- Sintaxis y funciones básicas de C++.
- Uso de punteros.
- Estructuras y clases.
- Condicionales y ciclos básicos.
- Memoria dinámica.



DEFINICIÓN

La lista enlazada circular no es más que una lista enlazada en la que el último elemento de la lista está enlazado al primer elemento de la lista, formando un círculo cerrado.

Una lista circular es una lista lineal en la que el último nodo apunta al primero. Es decir no existen casos especiales, cada nodo siempre tiene uno anterior y uno siguiente.



IMPORTANCIA

Recorrido continuo y eficiente

Las listas simplemente ligadas circulares permiten recorrer los nodos sin un final definido, facilitando el acceso continuo a los datos sin necesidad de reiniciar el puntero.

Aplicaciones en sistemas y software

Se utilizan en buffers circulares, administración de procesos, listas de reproducción y sistemas de turnos, donde el acceso cíclico es clave para la eficiencia.

Optimización en operaciones

Permiten inserciones y eliminaciones rápidas cuando se mantiene un puntero al último nodo, reduciendo el tiempo de recorrido en comparación con listas lineales.

DIFERENCIAS ENTRE LSL Y LSLC

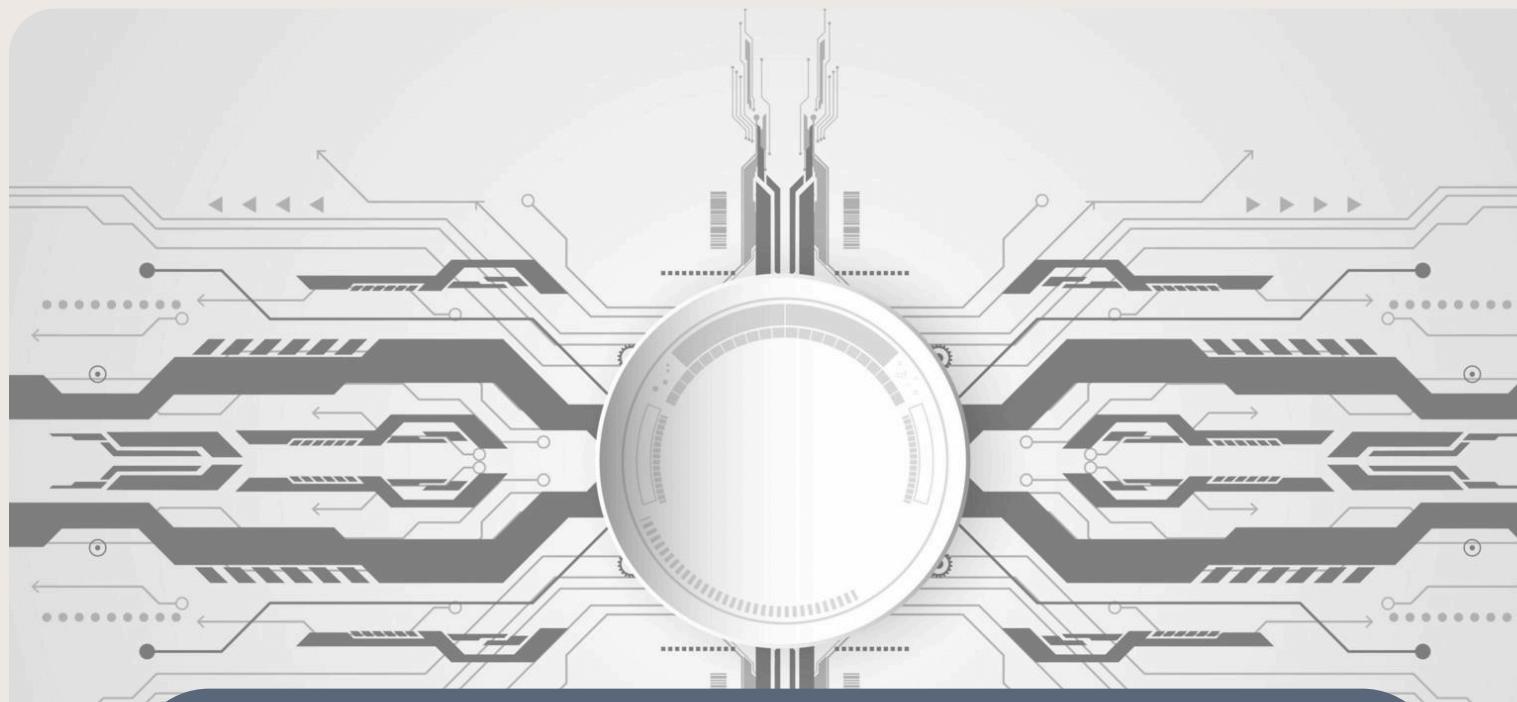
LSL

En una lista simplemente ligada lineal, cada nodo apunta al siguiente, y el último nodo tiene un puntero nulo (NULL), lo que indica el final de la lista y permite un recorrido con un límite claro.

LSLC

En una lista simplemente ligada circular, el último nodo apunta de regreso al primero, formando un ciclo sin un nodo final nulo, lo que permite recorridos continuos sin interrupción.

VENTAJAS Y DESVENTAJAS



Ventajas

- No hay nodos con punteros nulo.
- Facilitan los recorridos cíclicos.
- Aprovechamiento de memoria.
- Recorrido sin restricciones.



Desventajas

- Mayor complejidad de implementación.
- No hay un punto de finalización claro.
- La eliminación y búsqueda más difíciles.
- Menos intuitiva para usos simples.

INSERCIÓN DE NODOS

Agregar al inicio

```
void agregarIzquierda(apu_nodo &inicial, apu_nodo &actual, int &ban, int val) {  
    apu_nodo aux = new nodo;  
    aux->valor = val;  
  
    if (ban == 0) {  
        inicial = aux;  
        inicial->sig = inicial;  
        actual = inicial;  
        ban = 1;  
    } else {  
        aux->sig = inicial;  
        actual->sig = aux;  
        inicial = aux;  
    }  
    printf("Nodo agregado al inicio.\n");  
}
```

Agregar al final

```
void agregarDerecha(apu_nodo &inicial, apu_nodo &actual, int &ban, int val) {  
    apu_nodo aux = new nodo;  
    aux->valor = val;  
  
    if (ban == 0) {  
        inicial = aux;  
        inicial->sig = inicial;  
        actual = inicial;  
        ban = 1;  
    } else {  
        aux->sig = inicial;  
        actual->sig = aux;  
        actual = aux;  
    }  
    printf("Nodo agregado al final.\n");  
}
```

Puntos importantes:

- ban es una bandera para saber si ya existen nodos.
- En una lista simplemente ligada circular, actual->sig siempre apunta al siguiente nodo después del nodo actual, esa línea asegura que el último nodo (el apuntado por actual) apunte al nuevo nodo aux, cerrando el ciclo circular

INSERCIÓN DE NODOS

Agregar en una posición indicada

```
void agregarEnPosicion(apu_nodo &inicial, apu_nodo &actual, int &ban, int val, int pos) {
    if (ban == 0 || pos == 1) {
        agregarIzquierda(inicial, actual, ban, val);
        return;
    }

    apu_nodo aux = new nodo;
    aux->valor = val;
    apu_nodo temp = inicial;
    int contador = 1;

    while (contador < pos - 1 && temp->sig != inicial) {
        temp = temp->sig;
        contador++;
    }

    aux->sig = temp->sig;
    temp->sig = aux;
    if (temp == actual) actual = aux;

    printf("Nuevo nodo agregado en la posición %d.\n", pos);
}
```

- if(temp == actual) actual = aux; Si temp es el último nodo, actualiza el puntero actual para que apunte al nuevo nodo (aux), ya que el nuevo nodo se convierte en el último nodo de la lista.

Puntos importantes:

- ban es una bandera para saber si ya existen nodos.
- pos es un variable que utilizamos para saber la posición de los nodos.
- Usamos la función de agregar a la izquierda porque se agrega en la posición 1 o cuando no existen nodos.
- Iniciamos nuestro contador en 1 para facilitar la lógica del usuario.
- En el while es importante especificar temp->sig != inicial para evitar que el ciclo se repita infinitamente ya que es circular.
- Si el usuario ingresa la posición 10 pero solo tenemos 4 nodos agregados con el mismo while nos aseguramos de que se agregara en la ultima posición al final de la lista.
- Al hacer aux->sig = temp->sig;, estamos diciendo que el nuevo nodo aux debe apuntar al nodo que temp apuntaba
- temp->sig se actualiza para que apunte al nuevo nodo aux
Esto enlaza el nodo anterior (temp) con el nuevo nodo aux (aux), insertándolo en la lista.

MODIFICACIÓN DE NODOS

```
// Modificar nodo

void modificarNodoPorPosicion(apu_nodo inicial, int ban, int pos, int nuevoValor) {
    if (ban == 0) {
        printf("La lista está vacía.\n");
        return;
    }

    apu_nodo temp = inicial;
    int contador = 1;

    while (contador < pos && temp->sig != inicial) {
        temp = temp->sig;
        contador++;
    }

    if (contador == pos) {
        temp->valor = nuevoValor;
        printf("Nodo en la posición %d modificado correctamente.\n", pos);
    } else {
        printf("Posición no encontrada.\n");
    }
}
```

Puntos importantes:

- Usamos < porque recordemos que posición es 1 mayor que nuestro contador para facilidad del usuario.
- Si ingresa una posición que no existe el while se frena en el nodo que apunta a inicial, y no se cumple la condición del if.

MOSTRAR NODOS

```
void mostrarLista(apu_nodo inicial, int ban) {
    if (ban == 0) {
        printf("No has agregado ningun nodo aun.\n");
        return;
    }

    apu_nodo temp = inicial;
    int i = 1;

    while (true) {
        printf("El valor del nodo %d es = %d\n", i, temp->valor);
        temp = temp->sig;
        i++;
        if (temp == inicial) break;
    }
}
```

Puntos importantes:

- El while va a imprimir los valores de los nodos hasta que se temp sea igual que inicial, ya que eso nos indicara que ya estamos comenzando a recorrer los nodos de nuevo;

BUSCAR NODOS

```
void buscarNodoPorPosicion(apu_nodo inicial, int ban, int pos) {
    if (ban == 0) {
        printf("La lista está vacía.\n");
        return;
    }

    apu_nodo temp = inicial;
    int contador = 1;

    while (contador < pos && temp->sig != inicial) {
        temp = temp->sig;
        contador++;
    }

    if (contador == pos) {
        printf("Nodo en la posición %d tiene el valor: %d\n", pos, temp->valor);
    } else {
        printf("Posición no encontrada.\n");
    }
}
```

Puntos importantes:

- Usamos < porque recordemos que posición es 1 mayor que nuestro contador para facilidad del usuario.
- Si ingresa una posición que no existe el while se frena en el nodo que apunta a inicial, y no se cumple la condición del if.

ELIMINACIÓN DE NODOS

```
void eliminarNodoPorPosicion(apu_nodo &inicial, apu_nodo &actual, int &ban, int pos) {  
    if (ban == 0) {  
        printf("La lista está vacía.\n");  
        return;  
    }  
  
    apu_nodo prev = actual, temp = inicial;  
    int contador = 1;  
  
    // Si el nodo a eliminar está en la primera posición  
    if (pos == 1) {  
        if (temp == inicial && temp == actual) {  
            delete temp;  
            ban = 0;  
        } else {  
            temp = inicial;  
            inicial = inicial->sig;  
            actual->sig = inicial;  
            delete temp;  
        }  
        printf("Nodo en la posición 1 eliminado correctamente.\n");  
        return;  
    }  
  
    // Buscar el nodo en la posición dada  
    while (contador < pos && temp->sig != inicial) {  
        prev = temp;  
        temp = temp->sig;  
        contador++;  
    }  
  
    // Si el nodo existe en la posición proporcionada  
    if (contador == pos) {  
        prev->sig = temp->sig;  
        if (temp == actual) actual = prev;  
        delete temp;  
        printf("Nodo en la posición %d eliminado correctamente.\n", pos);  
    } else {  
        printf("Posición no válida.\n");  
    }  
}
```

Puntos importantes:

- La función **delete** en C++ se utiliza para liberar la memoria que ha sido previamente asignada dinámicamente con **new**.
- El caso del if donde **ban = 0** se debe a que solo existiría un nodo, ya que **temp == actual** y **temp == inicial**.
- Es importante conservar un apuntador que apunte al nodo anterior al nodo que borraremos, en este caso es **apu_nodo prev**.
- Si la posición ingresada es correcta haremos que nuestro **prev** (apuntador que apunta al nodo previo del que queremos borrar) apunte al que esta apuntado el que queremos borrar.
- **if (temp == actual) actual = prev;** Esto nos dice que si el nodo eliminado es el último nodo, entonces actualizamos **actual** para que apunte al nodo anterior, ya que ese se convierte en el nuevo último nodo de la lista.
- Debido al while para buscar cuando se elimina un nodo (incluso si es el último), el puntero **prev->sig** se actualiza para apuntar al siguiente nodo de **temp**. Si **temp** era el último nodo, el siguiente nodo será **inicial**, lo que mantiene la circularidad, por eso también se iguala **actual** con **prev**.

APLICACIONES

Las listas simplemente ligadas en forma circular se utilizan en aplicaciones donde se necesita un recorrido continuo sin un punto final, como en sistemas operativos para la gestión de procesos en planificación circular, en estructuras de datos para implementar buffers circulares eficientes, y en juegos o simulaciones donde se requiere un ciclo repetitivo de elementos. Su diseño permite una navegación optimizada sin necesidad de reiniciar el puntero.



Planificación



Buffers



Juegos



Simulaciones



Navegación

CONCLUSIONES

1

Las listas simplemente ligadas circulares permiten un recorrido continuo, optimizando el acceso a los datos sin necesidad de reiniciar punteros.

2

Son útiles en aplicaciones como planificación de procesos, buffers circulares y estructuras de datos eficientes en memoria.

3

Su implementación en C++ requiere un manejo adecuado de punteros para evitar errores y garantizar su correcto funcionamiento.



CÓDIGO COMPLETO

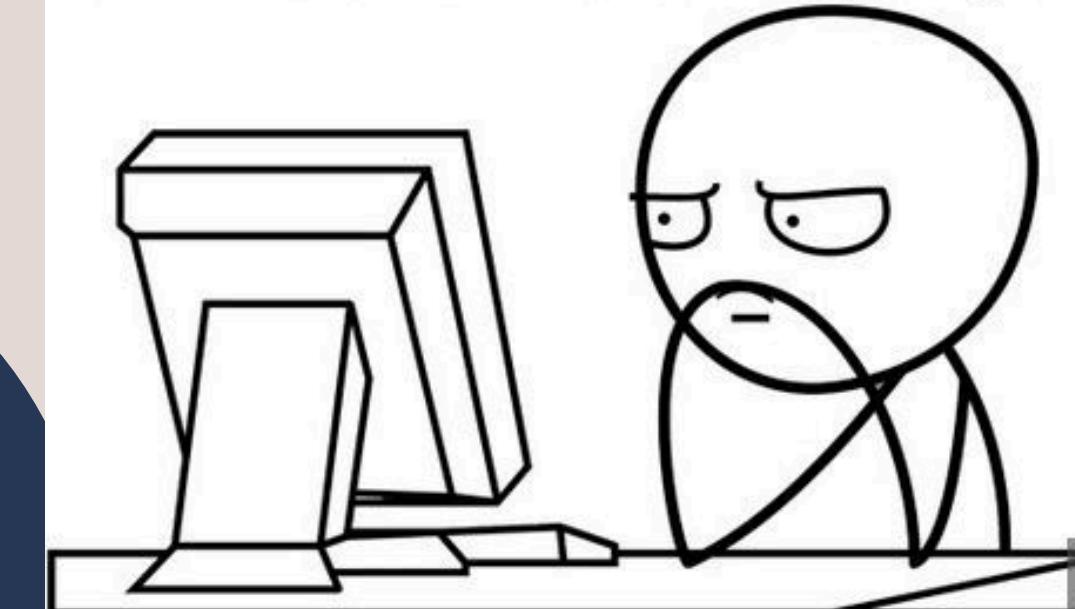




Escuela Superior
de Cómputo

**MUCHAS GRACIAS
POR SU ATENCIÓN**

NO COMPILA Y NO SÉ POR QUÉ



COMPIILA Y NO SÉ POR QUÉ

