#github

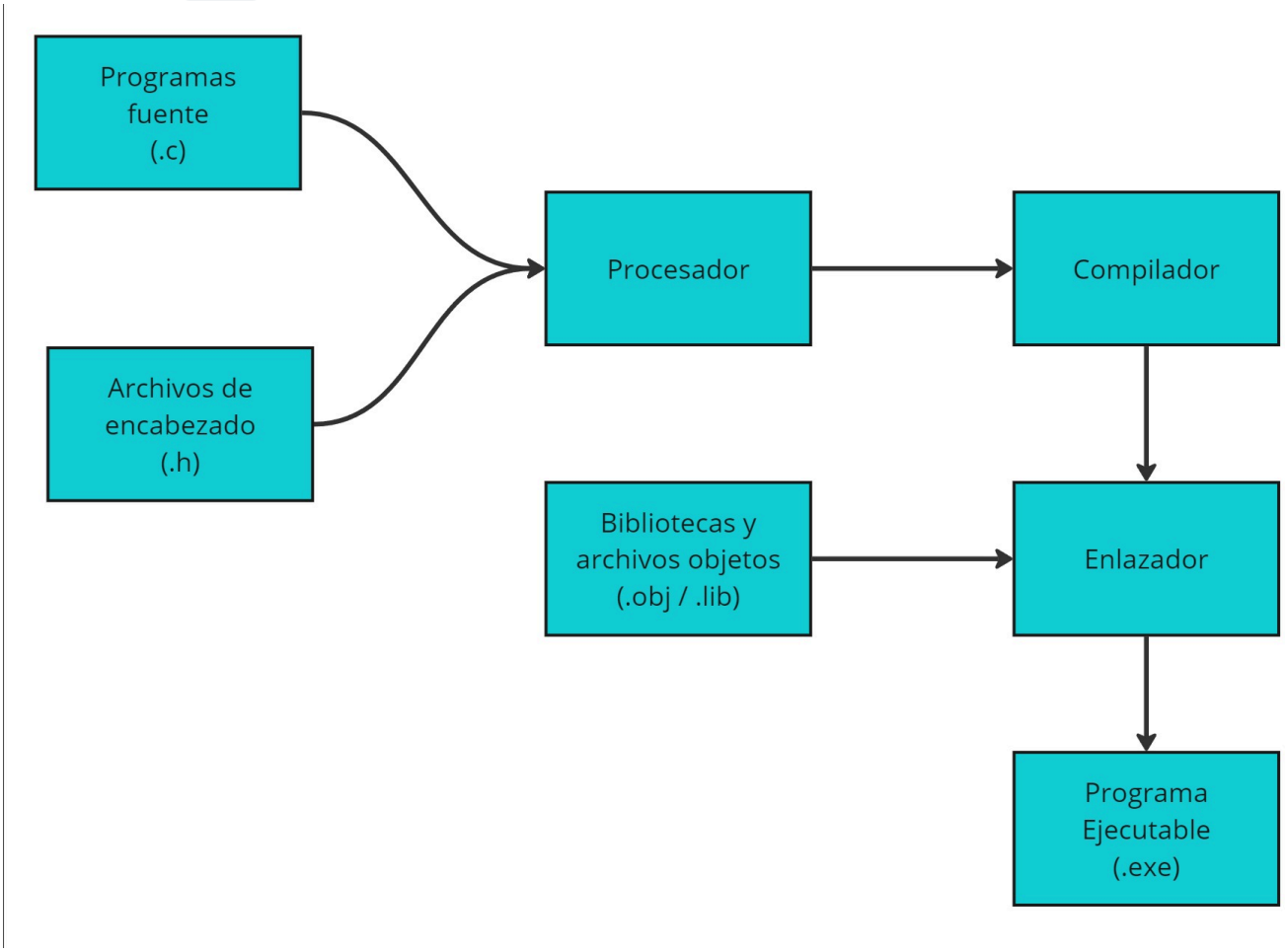
C es, actualmente, un lenguaje de propósito general, pero en sus comienzos, alrededor de los 70' nació como la idea de un lenguaje **sistémico**, un lenguaje que se usara para desarrollar **sistemas operativos y herramientas de bajo nivel.** Creado por **Dennis Ritchie** en los laboratorios de Bell. Se creó originalmente para sistema operativos como UNIX, hasta que se terminó extendiendo su uso.
(Las **herramientas de bajo nivel** son las que interactúan directamente con el hardware o sistema operativo de la computadora casi directamente. Herramientas que operan con la memoria, el procesador, etc.).
C logró que en muchos casos los sistemas operativos de esa época se dejaran se de desarrollar con lenguajes que se los denomina **ensamblador**.
(El **lenguaje ensamblador** es un tipo de lenguaje de programación de bajo nivel que está muy relacionado con el **código máquina**, es decir, de las instrucciones que el procesador de un dispositivo ejecuta directamente.
Los problemas que venían relacionados con los sistemas desarrollados en lenguajes ensamblador eran que, el ensamblador era específico para cada tipo de procesador que existía. No existía una compatibilidad directa entre el código que se escribiera para un hardware a otro. Además, también su mantenimiento era tan complejo como el desarrollo del código).
C apareció como una solución a muchos de los problemas que tenía el lenguaje ensamblador. Permitía que los sistemas escritos en C puedan acceder directamente al hardware que los componía, podían tener un control total del sistema y de sus recursos (como la asignación de memoria). Además, uno de los puntos más fuertes era su **portabilidad** también, que hacía sencillo poder hacer funcionar un sistema operativo escrito en C en diferentes arquitecturas de hardware.
Finalmente, con el tiempo, se C se convirtió en uno de los lenguajes más importantes de la historia de la programación. Lenguajes como **C++, Java, C#, Perl y PHP** son ejemplos de lenguajes que están directamente influenciados por C.

Resumen:
C es un lenguaje orientado a la programación de sistemas, es altamente transportable, muy flexible y genera código muy eficiente.

1.1 ¿Cómo son las fases del desarrollo de un programa en C?

El copilador que se esté usando traduce cada archivo fuente (que tengan formato `.c`) a código objeto **(binarios `.obj`)** y después el **LINKER (enlazador)** combina los archivos objetivos y las

bibliotecas (`.lib`) y genera un ejecutable.



1.2 Sintaxis de C:

Los programas escrito en C llevan una estructura definida que facilita el orden y organización del código en general de forma disciplinada.
Un programa escrito en C suele comenzar con las siguientes líneas:

```
#include <stdio.h>

int main(){

        printf("Hola Mundo!"); // Ejemplo de código.

        return 0;
}
```

Desglosando el código:
La línea `#include <stdio.h>` es una **directriz del procesador de C.**
C tiene por defecto una serie de procesos ya desarrollados y almacenados, que nos permiten hacer una variedad de operaciones, tanto matemáticas, como trabajar con datos de entrada y de salida, mostrar cosas en pantalla, etc. Todos estos procesos se encuentran almacenados en la **librerías** que trae por defecto C en su interior.

La línea `#include` le indica al compilador que tiene que incluir dentro del ejecutable final el contenido de ese archivo de cabecera, que en este caso se llama `<stdio.h>` antes de compilarlo.
`<stdio.h>` es un archivo de cabecera estándar en C que tiene declarada las funciones que trabajan con las **entradas/salidas** de nuestro programa que estemos haciendo. `<stdio.h>` significa **Standard Input/Output**. Algunas de las funciones que mas se usan de este archivo de cabecera son:

- `printf()` : Nos deja mostrar texto en la pantalla.
- `scanf()` : Nos permite leer datos introducidos por teclado.
- `foopen()` : Nos permite abrir archivos.
- `fclose()` : Permite cerrar los archivos.
  La línea `int main(){}` es el corazón del programa en C. Los **paréntesis** `()` que están d después del `main` indican que todo ese bloque de código es una **función**. Todos los programas escritos en C comienzan ejecutándose dentro de la función `main()`. Lo que le sigue, las llaves `{}` declaran el comienzo y fin del cuerpo de la función. Todo lo que esté dentro de las llaves es un **bloque**.
  La línea `printf("Hola Mundo!");` muestra por pantalla el texto "Hola Mundo". La función `printf()` es la encargada de mostrar texto por pantalla. Dentro de los `()` van los **argumentos** de la función. En este caso, el tipo de argumento que espera es un **texto** que pueda mostrar en pantalla. Todas las instrucciones tienen que terminar en un `;` para hacerle entender a C que ahí termina la declaración de esa operación.

1.3 Variables en C:

Las variables son un espacio que ocupan en la memoria que tienen una dirección asignada donde puede almacenar un dato. En C **antes de poder usar una variable en un programa hay que declararla, indicar de que tipo es (para que cuando se compile, se sepa de antemano cuanta memoria hay que reservarle a esa variable en particular) y su nombre.** Lo más recomendable es que se declaren todas las variables con las que se vayan a trabajar al comienzo de la función `main()`.
Los nombres de las variables puede ser cadenas de letras y dígitos que vayan de 1 a 32. Es importante recordar que en C las **minúsculas** son distintas de las **mayúsculas**. Normalmente, todas la variables se escriben en minúscula salvo que se trate de una **constante** y otros casos.

1.3.1 Declaración de una variable en C:

Generalmente, para declarar una variable en C, se hace de la siguiente forma:
Se indica el `tipo_de_variable` y el `nombre_variable`, o sea `tipo nombre;`

TIPO	DESCRIPCIÓN	PALABRA CLAVE	FORMATO Y TAMAÑO
Entero	Número sin decimales entre -32768 y 32767	int	%d 2 byte

TIPO	DESCRIPCIÓN	PALABRA CLAVE	FORMATO Y TAMAÑO
Entero largo	Número sin decimales entre ±2.147.483.647	Long	%ld 4 byte
Punto flotante	Número con decimales en 3.4E - 38 decimales (6).	Float	%f 6 byte
Doble punto flotante	Número con decimales entre 1.7E - 308 decimales (10).	Double	%lf 8 byte
Carácter	Un carácter.	Char	%c 1 byte
String	Cadena de caracteres	char variable `[longitud]`	%s tantos bytes como caracteres.

Por ejemplo:

```
float mi_decimal;
int mi_entero;
char a;
char mi_nombre[15];
```

1.4 Operadores aritméticos en C:

Operador	Acción
-	Resta
+	Suma
`*`	Multiplica
/	Divide
%	Módulo
--	Decrementa
++	Incrementa

1.5 Operadores relacionales y lógicos en C:

Los **operadores relacionales** son **símbolos** que se usan para poder comparar dos valores. Estos operadores pueden devolver dos respuestas únicamente: Si el resultado de la comparación es correcta, la expresión va a ser considerada **verdadera**, en caso contrario es **falsa**.

Operador	Significado
>	Mayor que.

Operador	Significado
>=	Mayor o igual que.
<	Menor que.
<=	Menor o igual que.
==	Igual
!=	No igual

Los **operadores lógicos** retornan un **valor lógico** basados en las tablas de verdad:

Operador	Significado
&&	And (y)
||	Or (o)
!	Not

1.6 Operadores de Asignación:

En C, el operador que se usa para asignar valores se hace con el signo `=`.
Una **Asignación** significa que la variable que se encuentre a la izquierda del `=` guarda el valor que este declarado del lado derecho del `=`. Ahora, si a la derecha hay un operando o más, primero se resuelve la expresión y después se asigna el valor. Si a la derecha hay un operando más, pasa lo mismo.
EJ:
`variable1 = variable2;` Se le está asignando el valor de `variable2` a la variable `variable1`.
EJ2:
`variable1 = variable2*10`; Se le está asignando a la variable `variable1` el valor de la `variable2` pero multiplicado por 10.

1.7 `printf()` y `scanf()` (instrucciones de entrada/salida):

Para imprimir o mostrar texto en pantalla se usa la función `printf()` como la vista en el primer ejemplo. Hay **2 formas** de escribir esta función: con solo un argumento o con más de un argumento.

1. EJ: `printf("Hola Mundo!");` Hay un solo argumento dentro de los `()`, que es directamente el texto que se quiere imprimir en pantalla escrito entre las `""`.
2. EJ: `printf("Tengo %d años!", mi_edad);` dentro de las `""` donde se escribe el texto se pueden poner el formato del tipo de una variable cualquiera para que sea mostrado en pantalla. En este ejemplo, dentro de `"Tengo %d años"` esta el elemento `%d` que hace referencia a un tipo de dato **entero**. Después de la `,` se escriben orden las variables que se

quieran mostrar en pantalla. En este caso, hay solo una, `mi_edad` que podría haberla definido más arriba como `int mi_edad = 18;`, por lo que la salida por pantalla sería "Tengo 18 años".

Para **ingresar datos por teclado** se puede hacer uso de la función `scanf()`:
EJ: `scanf("Arg 1", &Arg 2);`: El primero argumento `Arg1` es la cadena de control de formato, ahí se especifica el tipo de dato que se tiene que ingresar por el usuario y va entre comillas dobles `""`. El segundo argumento `&arg2` que empieza con un **ampersand** `&`, o **operador de dirección**, seguido por el nombre de la variable cuyo formato se declaró en el primer argumento, indica que lo que se introduzca se guarde en la dirección de `&Arg 2`.

Ejemplo: *¿Cómo se podrían sumar dos números enteros en C?*

```
#include <stdio.h>

int main(){
        int num1, num2, sum; // Acá se declararon las variables.

        printf("Ingrese el primer número: \n");
        scanf("%d", &num1); Guardamos un '%d' (entero) a la dirección de 'num1';

        printf("Ingrese el segundo número: \n");
        scanf("%d%, num2);

        sum = num1 + num2;

        printf("El resultado de la operación es: %d", sum);

        return 0;
}
```

1.7.1 Para mejorar la salido de puntos flotantes:

Para mejorar el formato `%f` que se va a mostrar con 6 decimales aunque no sea real, por ejemplo en el caso de sueldo no queda bien mostrar: *$1245,500000*, entonces podemos recurrir a estas mejoras:
`% 6 1f` son 6 caracteres en total de los cuales 1 es decimal.
`%.2f` si adelante del `.` no hay dígito significa que la cantidad de caracteres totales no está restringida, pero el dígito que está atrás del `.` es la cantidad de decimales que se van a mostrar.

1.8 Estructuras de decisión o selección:

1.8.1 Sentencia `if`:

Nos permite tomar una decisión para ejecutar una acción u otra, basándose en un resultado booleano de una condición.

1.8.1.1 Sintaxis:

```
if (condición) {

        sentencia 1;

}
else {

        sentacia 2;

}
```

La salida por el **VERDADERO** sempre debe existir.
Es por eso que el `else` es opcional es posible que ante una decisión por la opción **V** se tenga que ejecutar sentencias, pero por el **F** no se ejecute ninguna sentencia. En caso no ser así, se tiene que poner la condición inversa.
Si tanto la salida de V o F se tienen que ejecutar más de 1 sentencia, las salidas se tienen que poner en `{}` para limitar el bloque.
Los **operadores de relación** que van a la condición son `<` , `>` , `<=` , `>=` , `==` , `!=` .
Cualquier regla convencional de sangrías que se elija tiene que aplicarse con cuidado en todo el programa. Un programa que no sigue las reglas de esparcimiento uniformes es difícil de leer.

1.8.1.2 Anidamiento de `if` :

1.8.1.3 Sintaxis:

```
if (condición 1) {

        if (condición 2) {

                sentencia 1;

        }

}
```

Y si sacamos las claves:

```
if (condición 1)
        if (condición 2)
                sentencia 1;
else:
                sentencia 2;
```

Los operador lógicos son: `AND` y `&&` , `OR` o `||` .

1.9 Estructuras de CICLOS:

Un ciclo le permite al programador que se repita una acción, en tanto cierta condición se mantenga VERDADERA:

1.9.1 Sentencia usando `WHILE` :

Sintaxis:

```
while (condición){

        sentencia 1;

}
```

Si se repite más de una sentencia:

```
while (condición) {

        sentencia 1;
        sentencia 2;

}
```

Si la condición es VERDADERA entra en el `while` y se ejecutan las sentencias dentro del `while` .
Si la condición es FALSA no entra y ejecuta la próxima instrucción ejecutable fuera del `while` .

IMPORTANTE:
Cuando la variable es un contador que se incrementa en +1 se puede escribir como: `variable++;`
Cuando la variable es un contador que se incrementa en -1 se puede escribir como: `variable--;`

1.9.2 Sentencia `DO/WHILE` :

Es similar al `while`. En el `while` la condición se encuentra al principio, antes de ejecutarse el cuerpo del mismo.
La estructura `DO/WHILE` prueba la condición de continuación del ciclo DESPUÉS de ejecutarse el cuerpo del ciclo, y por lo tant el cuerpo se **va a ejecutar por lo menos una vez.** Cuando la condición es FALSA se ejecuta la acción siguiente a la del `while`.

SINTAXIS:

```
do {

        sentencia 1;

        sentencia 2;

} while(condición){};
```

IMPORTANTE:
En el caso de repeticiones controladas por contadores se requiere que:

- El nombre de una variable de control (controlador del ciclo).
- El valor inicial de dicha variable.
- El incremento (o decremento) con el cual, cada vez que se termina un ciclo, la variable de control se modifica.
- La condición que compruebe la existencia del valor final de la variable de control.
  En caso de no ser variables del tipo controlador SIEMPRE se tiene que haber una **primera lectura** antes de la condición en el `while` (sino la condición se compararía contra "basura") y las **siguientes estructuras** antes de la llave del fin del `while`.

```
primera lectura;

while(condicion) {

        sentencias;

        siguiente lectura;

}
```

1.9.3 Sentencia `FOR`:

Se utiliza SÓLO cuando se sabe exactamente la cantidad de repeticiones que queremos hacer:
SINTAXIS:

```
for (variable1 = expresión1; condición; progesión de la condición) {

        sentencia1;
}
```

```
for (variable1 = expresion; progresion de la condicion){

        sentencia1;

        sentencia2;

}
```

En `variable1 = expresion1` se **INICIALIZA** la `variiable1` con el valor de la `expresion1`.

- La **CONDICIÓN** es la prueba de control de fin (nunca va un igual), mientras sea VERDADERA se repite el ciclo.
- La **PROGRESIÓN DE LA CONDICIÓN** es el incremento de avance de la `variable1`.
  El ciclo termina si la CONDICIÓN ES FALSA.
  En caso de TENER MÁS DE UNA SENTENCIA lo que se repite se encierra entre `{}`.

Ejemplos:
Imprimir los múltiplos de 7 que hay entre 7 y 112:

```
#include <stdio.h>

int main() {

        int x;

        for (x = 7; x <= 112; x += 7) {

                printf("%d ", x);

        }

}
```

EL **valor inicial** es 7, la condición es que sea <= 112 y el incremento o progresión es de 7, por pedir múltiplos de 7.

Imprimir los números del 9 al 1:

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

```
int main() {

        int k;

        for(k = 9; k >=1; k--) {

                printf("%d ", k);

        }

}
```

En este ejemplo el valor inicial es 9, pero el final es 1, valor menos al inicial. Es por eso que la condición tiene que ser con `>=1` y el incremento es de `-1` para que vaya disminuyendo la variable.