



PROGRAMACION AVANZADA

ING. INDUSTRIAL

Esp. Ing.Prof.: Cristina Delia Cruz



FUNCIONES EN C++



Unidad 2





DIVIDE Y VENCERAS

Divide y Vencerás es una estrategia de diseño de algoritmos y resolución de problemas que **consiste en dividir un problema grande en varios subproblemas más pequeños, resolver cada uno de estos subproblemas, y luego combinar las soluciones de estos subproblemas para obtener la solución del problema original.**

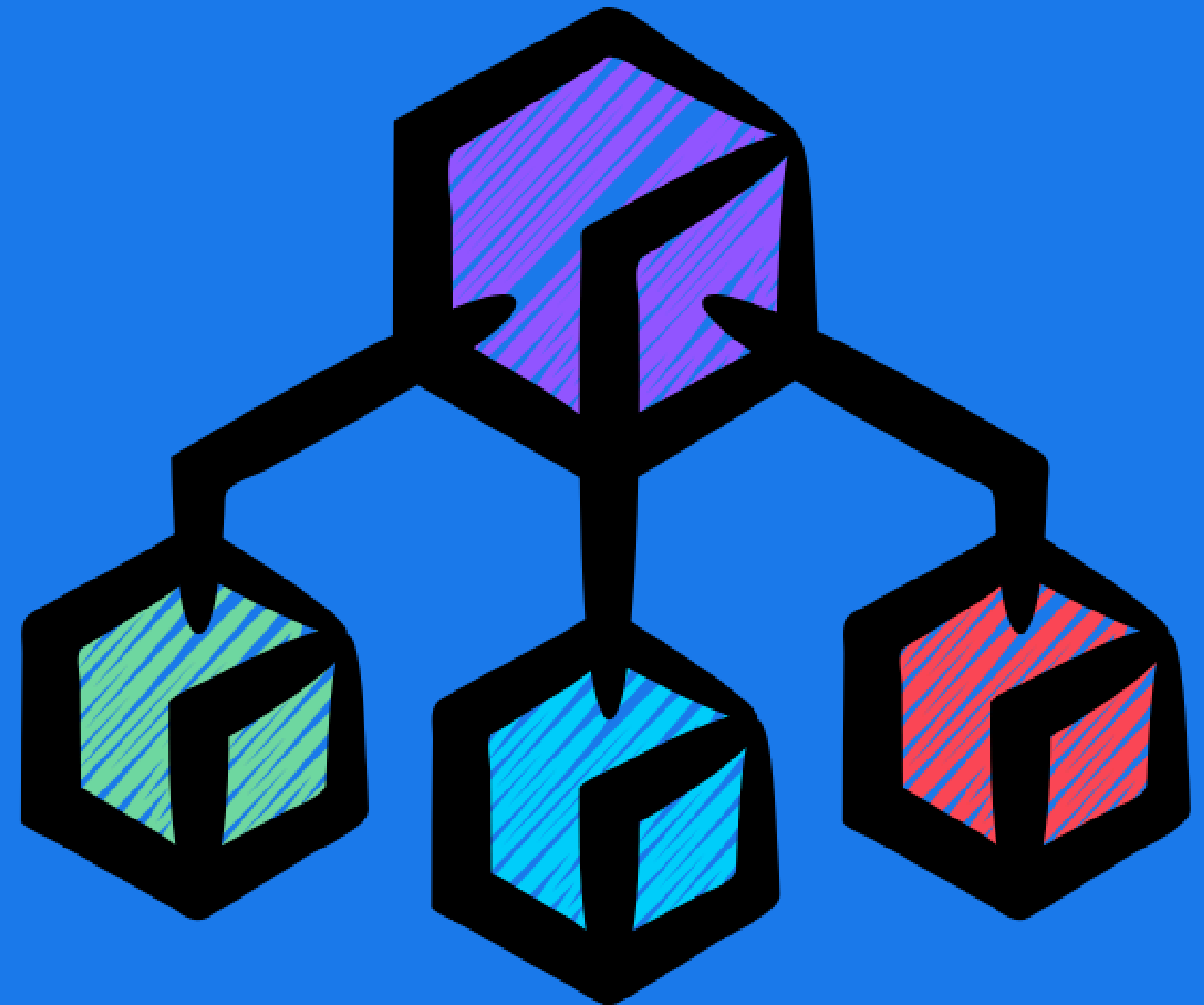




DIVIDE Y VENCERAS

Proceso de "Divide y Vencerás":

1. **Dividir:** El problema original se divide en dos o más subproblemas más pequeños, que son instancias más simples del problema inicial.
2. **Resolver (o vencer):** Cada uno de estos subproblemas se resuelve de manera recursiva (es decir, utilizando el mismo enfoque para resolver problemas más pequeños). Si el subproblema es lo suficientemente pequeño, se resuelve directamente (es el caso base).
3. **Combinar:** Las soluciones de los subproblemas se combinan para obtener la solución del problema original.



DISEÑO DESCENDETE

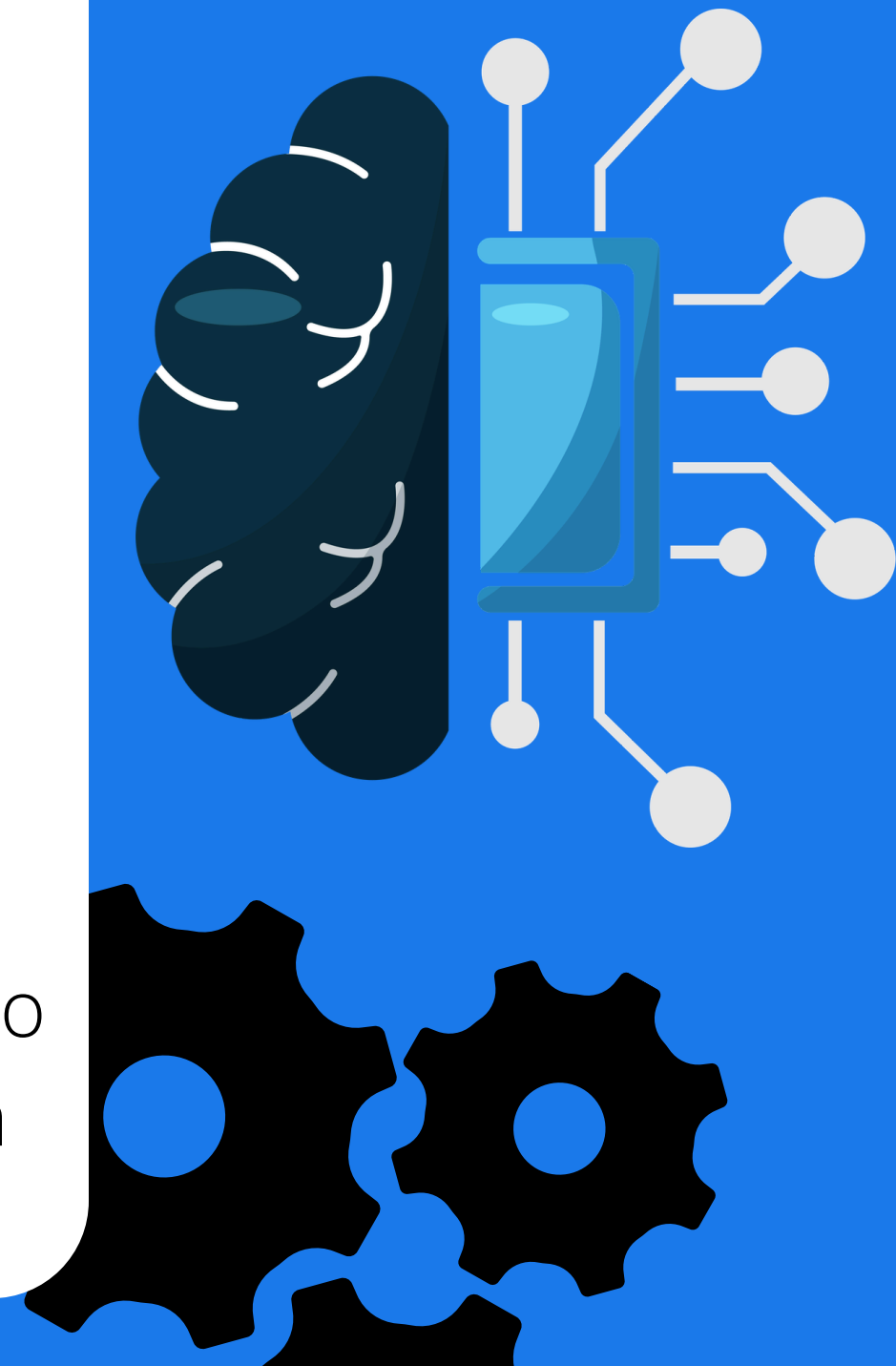
El diseño descendente (también conocido como "Top-Down Design") es una metodología de desarrollo de sistemas y software que consiste en descomponer un problema o sistema complejo en partes más pequeñas y manejables. La idea principal es comenzar desde una vista global o abstracta del problema y luego descomponerlo en componentes o subproblemas más específicos y detallados hasta que cada componente pueda ser fácilmente implementado o resuelto.



DISEÑO DESCENDETE

Características del diseño descendente:

1. **Partir del nivel más alto:** Se comienza con una descripción general del sistema o problema en su totalidad, sin entrar en detalles específicos.
2. **Descomposición en módulos:** El problema se descompone en varios subproblemas o módulos que son más sencillos de resolver. Esto continúa en varios niveles hasta que cada submódulo sea lo suficientemente simple para ser resuelto directamente.
3. **Implementación paso a paso:** Se comienza implementando los niveles más bajos de abstracción. Luego, los módulos pequeños se integran para formar los módulos más grandes, y finalmente el sistema completo.
4. **Recursividad en la descomposición:** Si uno de los módulos sigue siendo complejo, se repite el proceso de descomposición hasta que se llega a componentes manejables.

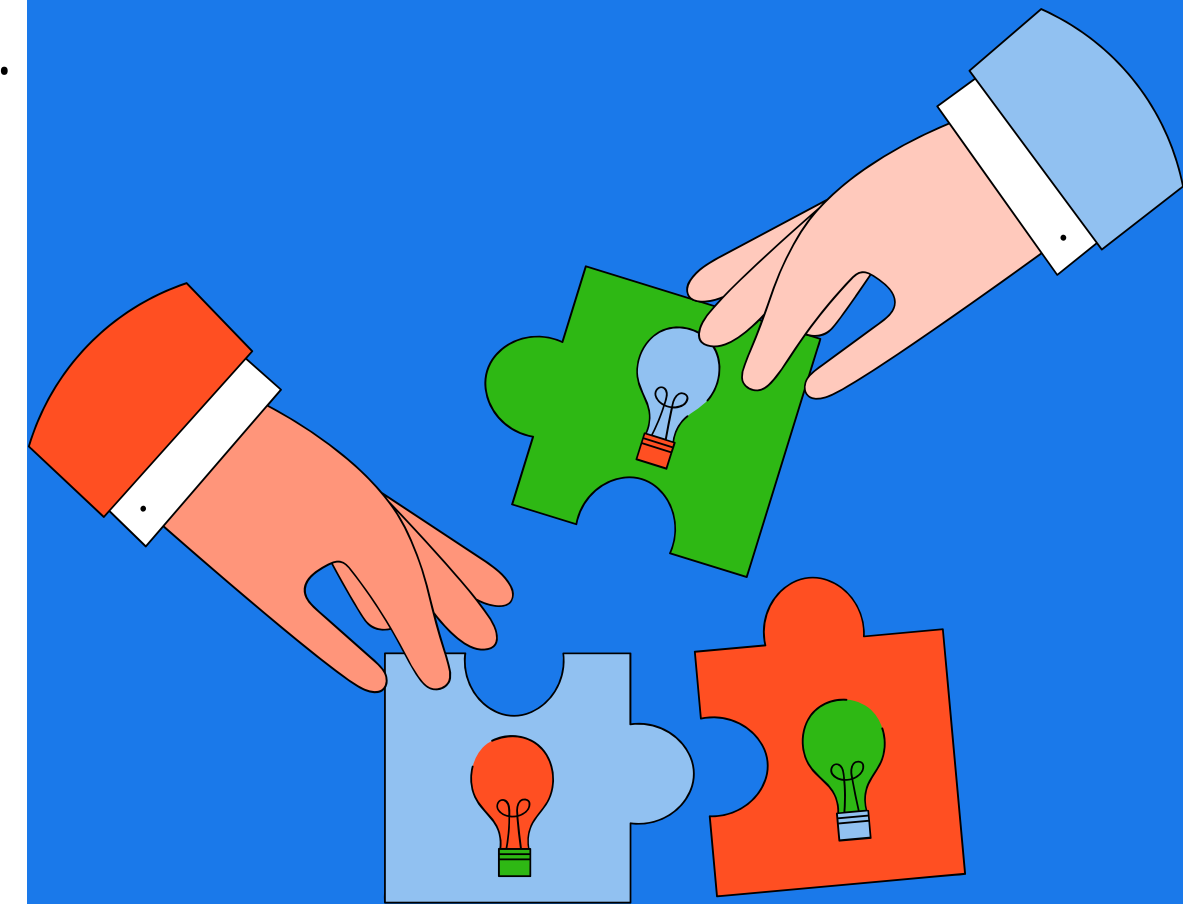




DISEÑO DESCENDETE

Ventajas del diseño descendente:

- **Organización clara:** Facilita una visión clara de todo el sistema y permite que cada parte se desarrolle de manera independiente.
- **Facilita la gestión:** Como se enfoca primero en los módulos más importantes, permite a los desarrolladores y gestores planificar el proyecto en etapas.
- **Modularidad:** Como se divide en módulos, facilita el mantenimiento, ya que cada módulo puede ser modificado sin afectar todo el sistema.
- **Facilidad en la documentación:** Es más fácil documentar el sistema, ya que se construye gradualmente desde los niveles más abstractos hasta los más detallados.



DISEÑO DESCENDETE

Desventajas:

- **Desconexión de los detalles:** El enfoque descendente puede a veces ignorar detalles importantes en las primeras etapas, lo que puede causar problemas cuando se llega a niveles inferiores.
- **Riesgo de errores en la descomposición:** Si el sistema no se descompone correctamente o los módulos no están bien definidos, el diseño puede ser ineficaz.
- **Tiempo inicial:** Al comenzar con una planificación general, puede requerir más tiempo al principio antes de comenzar la implementación detallada.





DISEÑO DESCENDETE

Ejemplo en C++:

Supongamos que queremos diseñar un sistema de facturación para un supermercado.

1. Definición del problema general:

- El sistema debe manejar productos, facturas y clientes.

2. División en módulos principales:

- Módulo de productos.
- Módulo de facturas.
- Módulo de clientes.

3. Descomposición de un módulo (facturas):

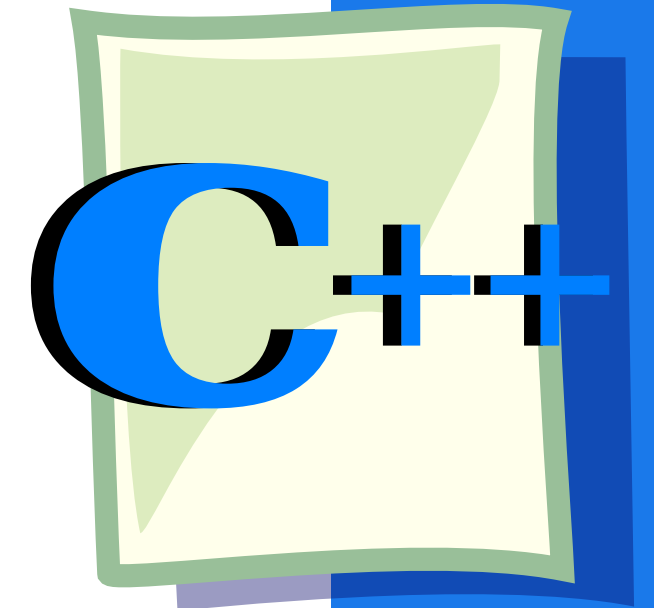
- El módulo de facturas puede tener submódulos para agregar productos a la factura, calcular el total, aplicar descuentos, etc.

4. Refinamiento:

- El submódulo de "calcular total" puede descomponerse aún más en "sumar precios" y "calcular impuestos".

5. Implementación:

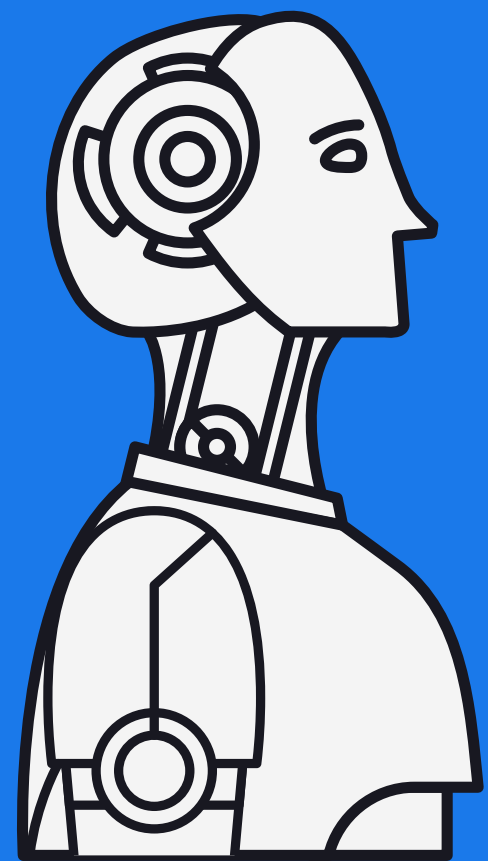
- Comenzamos implementando el cálculo de los precios de los productos, luego sumamos los impuestos y finalmente integramos el módulo de facturación completo.

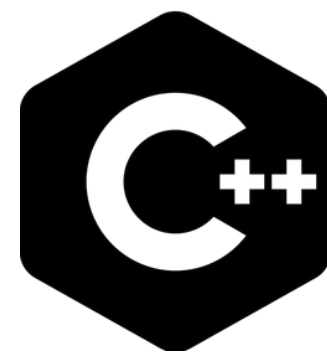


FUNCIONES



Las funciones son un concepto fundamental en la programación, ya que **permiten organizar el código en bloques reutilizables**, mejorando la claridad, estructura y mantenimiento del programa. Una **función es un bloque de código que realiza una tarea específica y que puede ser "llamada" o "invocada"** desde cualquier parte del programa para ejecutar esa tarea.



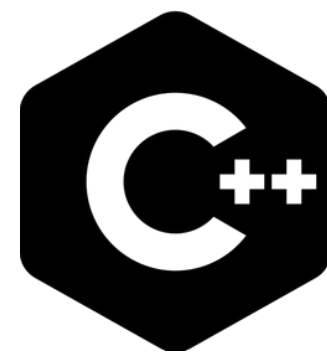


FUNCIONES

Características principales de una función:

1. **Reutilización del código:** Al definir una función, puedes reutilizar ese bloque de código cada vez que lo necesites sin tener que escribirlo de nuevo. Esto hace que el código sea más eficiente y fácil de mantener.
2. **Modularización:** Dividir un programa en varias funciones facilita la lectura, depuración y actualización del código, ya que cada función tiene un propósito específico.
3. **Parámetros y argumentos:** Las funciones pueden aceptar parámetros, que son valores o variables que se pasan a la función cuando se llama, para que pueda operar con ellos. Los argumentos son los valores que se envían a la función cuando es invocada.
4. **Retorno de valores:** Algunas funciones pueden devolver un resultado o un valor cuando finalizan. Este valor devuelto puede ser utilizado en otras partes del programa.



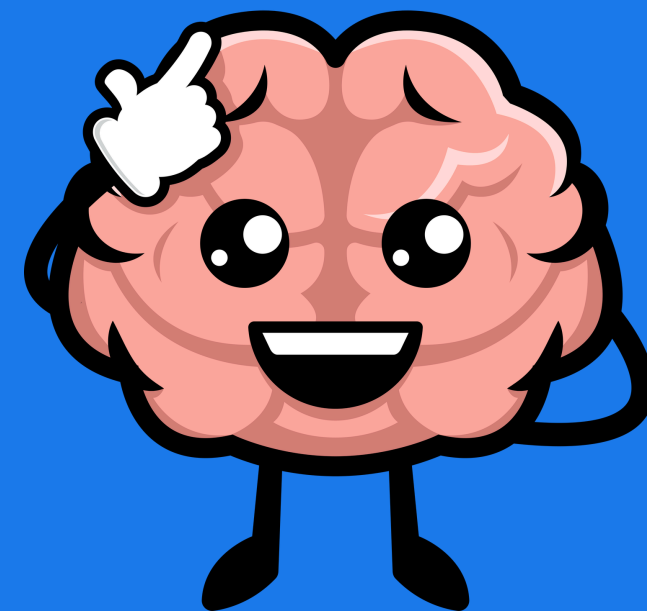


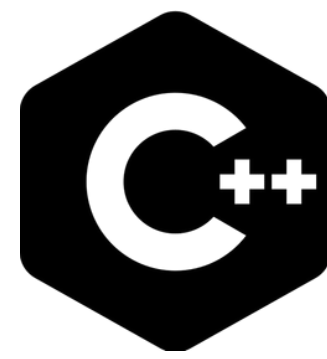
FUNCIONES

Estructura básica de una función en C++:

Una función en C++ tiene la siguiente estructura general:

```
tipo_de_retorno nombre_de_función(tipo_parametro1 nombre_parametro1,  
tipo_parametro2 nombre_parametro2, ...) {  
    // Bloque de código de la función  
    return valor; // opcional, si la función devuelve un valor  
}
```

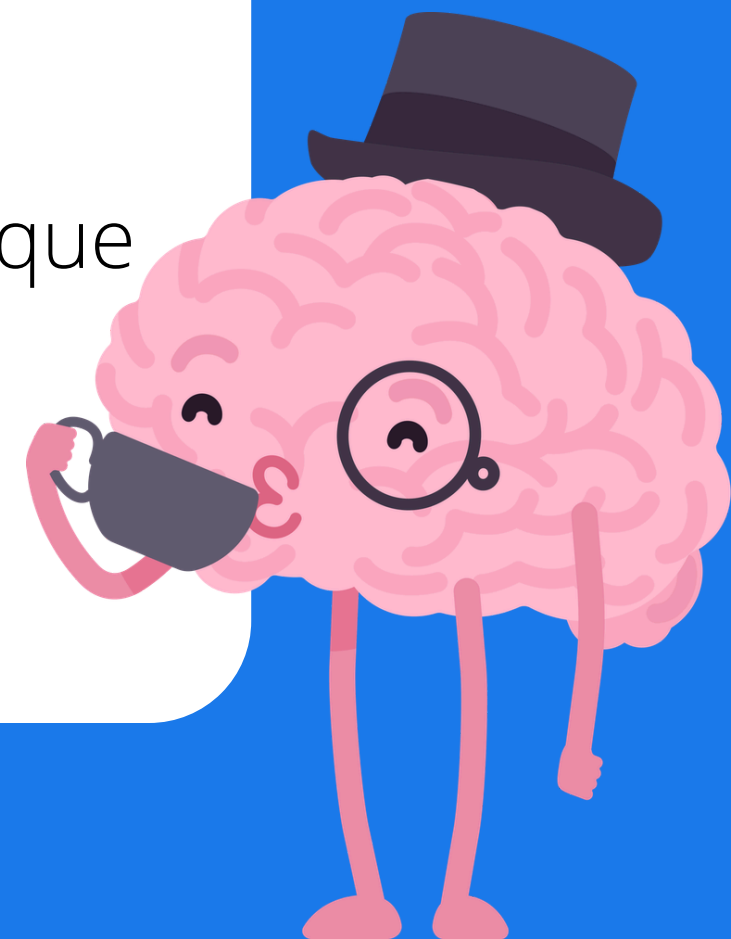
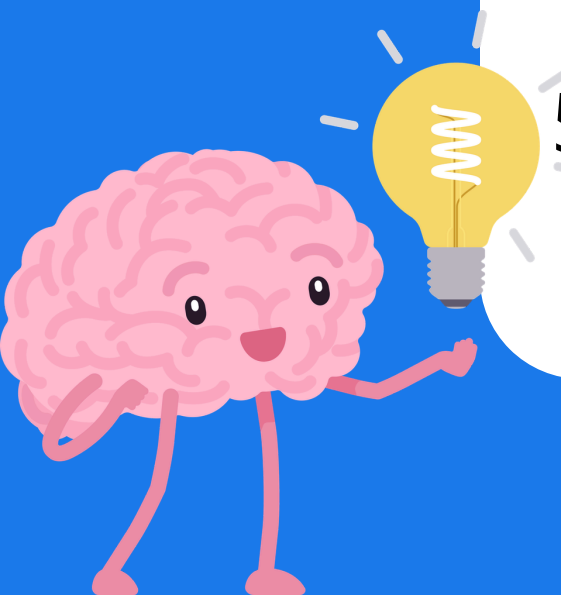




FUNCIONES

Componentes:

1. **Tipo de retorno:** Es el tipo de dato que la función devuelve, como int, double, char, void (si no devuelve nada), etc.
2. **Nombre de la función:** El nombre que se le da a la función para poder invocarla en otras partes del código.
3. **Parámetros (opcional):** Son los valores que la función recibe cuando es llamada. Pueden ser variables o constantes. Una función puede no recibir parámetros, en cuyo caso se deja vacío el paréntesis.
4. **Bloque de código:** Es el conjunto de instrucciones que definen lo que hace la función.
5. **Retorno (opcional):** Si la función devuelve un valor, se utiliza la instrucción return seguida del valor a devolver.





TIPOS DE FUNCIONES

Funciones sin retorno: Son funciones que realizan una acción pero no devuelven ningún valor. Su tipo de retorno es void.

```
void saludar() {  
    cout << "Hola, bienvenido!" << endl;  
}
```





TIPOS DE FUNCIONES

Funciones con retorno: Son funciones que realizan una acción y devuelven un valor. El tipo de retorno depende del tipo de dato que se devuelve.

```
int sumar(int a, int b) {  
    return a + b;  
}
```





TIPOS DE FUNCIONES

Funciones con o sin parámetros: Algunas funciones requieren parámetros para trabajar, mientras que otras no los necesitan.

```
double multiplicar(double x, double y) {  
    return x * y;  
}
```

Con Parámetros

Sin Parámetros

```
void mensaje() {  
    cout << "Este es un mensaje sin parámetros." << endl;  
}
```



PARÁMETROS

Por valor: Los parámetros se pasan a la función como copias. Si la función los modifica, esa modificación no afecta a las variables originales en el programa.

```
void incrementar(int x) {  
    x = x + 1;  
}
```

En este caso, si pasas una variable a incrementar, la variable original no se verá afectada.



PARÁMETROS

Por referencia: Los parámetros se pasan como referencia, lo que significa que cualquier cambio en los parámetros dentro de la función afectará a las variables originales.

```
void incrementar(int &x) {  
    x = x + 1;  
}
```





EJEMPLO 1: INICIALIZACIÓN DE TURNO



Escribir un programa en C++ que utilice una función sin retorno (void) encargada exclusivamente de mostrar por pantalla un mensaje de bienvenida institucional para los operadores que ingresan a su turno.

Requisitos específicos:

- 1. Crear una función llamada darBienvenida().**
- 2. La función debe imprimir por consola los siguientes mensajes de manera independiente:**
 - "Bienvenido al sistema de control de produccion."
 - "¡Que tenga un turno productivo!"
- 3. Desde la función principal (main()), invocar correctamente a la función darBienvenida() para ejecutar el saludo al arrancar el programa.**



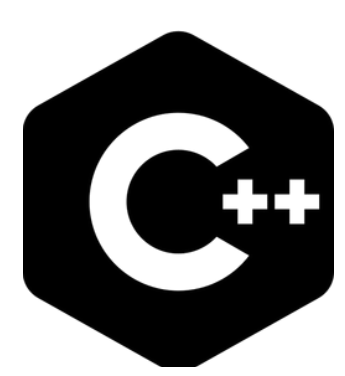
EJEMPLO 2: CÁLCULO DE COSTOS DE PRODUCCIÓN

Desarrollar un programa en C++ que utilice una función con retorno de tipo double para calcular el costo total de un lote de producción a partir del valor unitario y la cantidad de unidades fabricadas.

Requisitos específicos:

- 1. Diseñar una función llamada `calcularCostoTotal` que reciba como parámetros de entrada el costo por unidad (double) y la cantidad de unidades (int), y retorne el producto de ambos valores (double).**
- 2. En la función principal (main), solicitar al usuario que ingrese por teclado el costo por unidad y la cantidad de unidades producidas.**
- 3. Invocar a la función `calcularCostoTotal` pasándole los datos ingresados, almacenar el resultado e imprimir el costo total de producción por pantalla de forma clara.**





EJEMPLO 3: SIMULACIÓN DE RECEPCIÓN DE INVENTARIO

Desarrollar un programa en C++ que demuestre el concepto de paso de parámetros por valor utilizando una función de tipo void, evidenciando que los cambios realizados dentro de la función no afectan a la variable original del programa principal.

Requisitos específicos:

1. Definir una variable entera en la función principal (main) que represente el inventario actual (por ejemplo, con un valor inicial de 200).
2. Crear una función llamada `simularRecepcion` que reciba el inventario como parámetro (por valor), le sume una cantidad simulada (por ejemplo, 50 unidades) y muestre el resultado parcial dentro de la función.
3. Mostrar por pantalla el valor del inventario en el main antes y después de llamar a la función, comprobando mediante un mensaje o comentario que el valor original de la variable principal permanece intacto tras la ejecución.



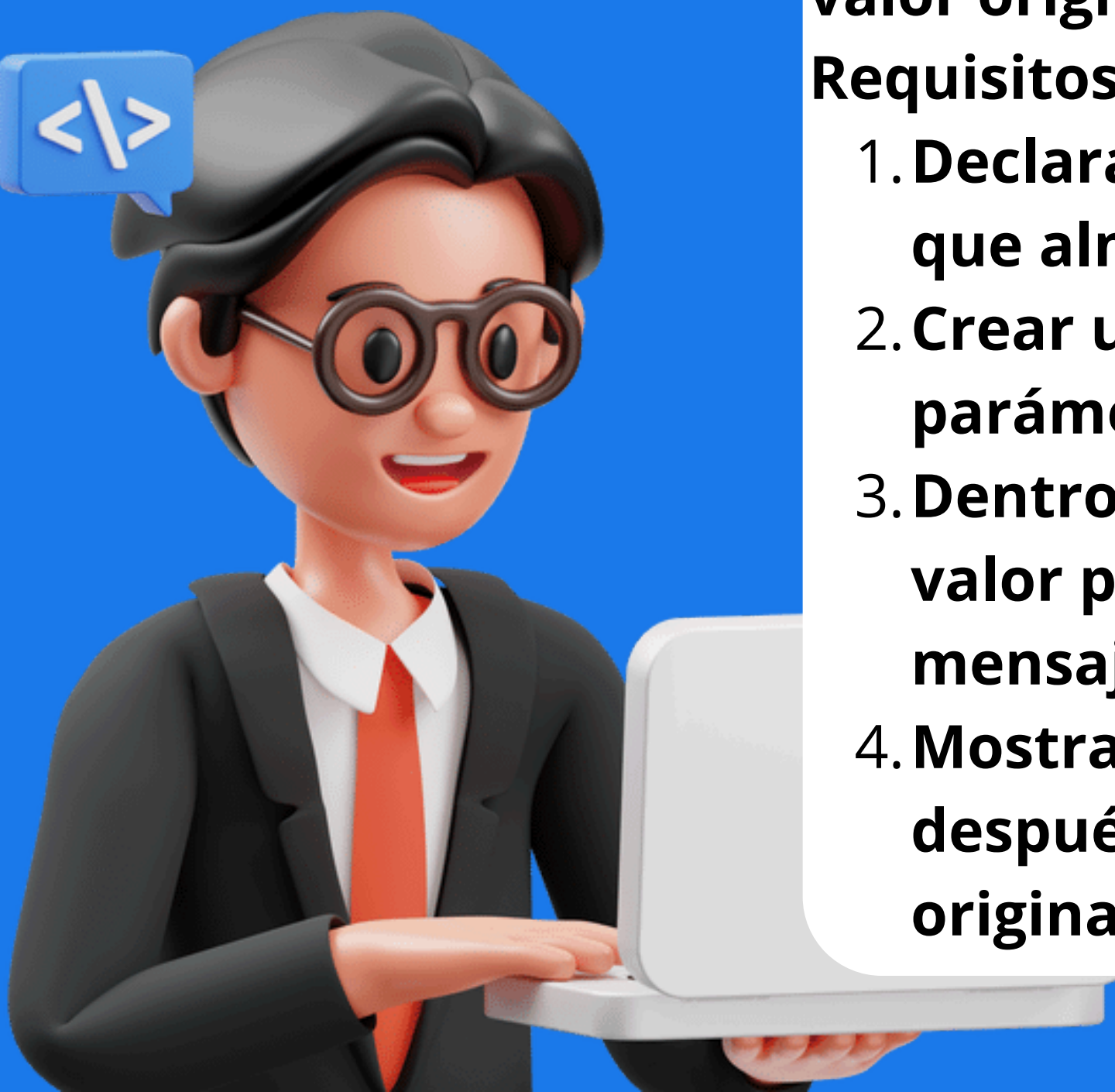


EJEMPLO 4: AJUSTE DE CALIBRACIÓN POR REFERENCIA

Desarrollar un programa en C++ que emplee el paso de parámetros por referencia mediante el uso del operador &, permitiendo modificar el valor original de una variable desde el interior de una función.

Requisitos específicos:

- 1. Declarar una variable de tipo double en la función principal (main) que almacene un valor de calibración inicial (por ejemplo, 15.5).**
- 2. Crear una función llamada ajustarCalibracion que reciba el parámetro por referencia (utilizando &calibracion).**
- 3. Dentro de dicha función, solicitar al usuario que ingrese un nuevo valor por teclado para actualizar el parámetro y mostrar un mensaje con la confirmación del cambio.**
- 4. Mostrar por pantalla el valor de la calibración en el main antes y después de la llamada a la función, evidenciando que el valor original fue modificado de manera directa.**



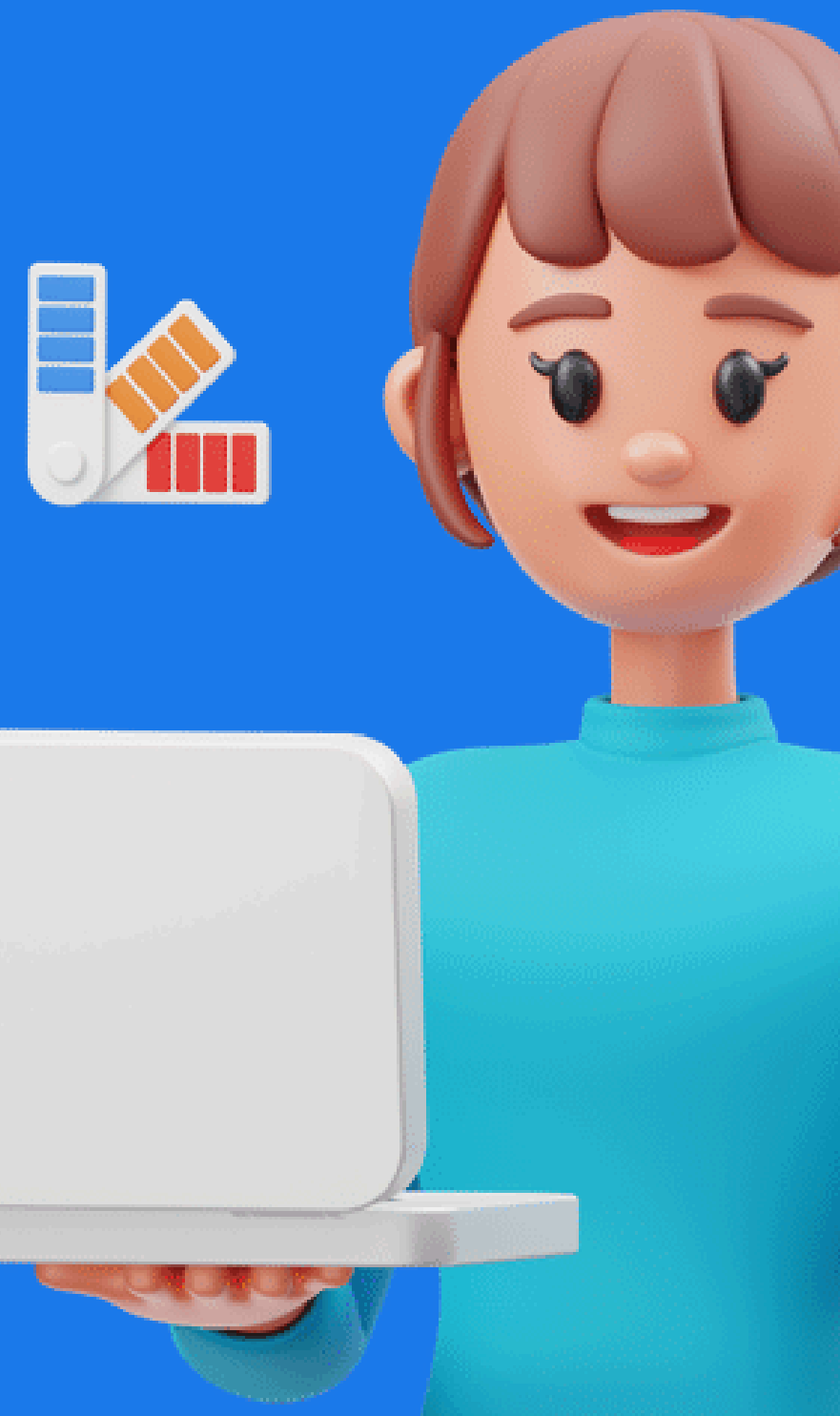


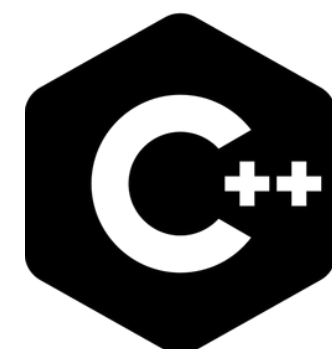
EJEMPLO 5: CÁLCULO DE ÍNDICE DE EFICIENCIA CON VALIDACIÓN

Desarrollar un programa en C++ que utilice una función con retorno de tipo double para calcular un índice de eficiencia, incorporando una estructura de selección para validar los datos de entrada y evitar fallos por división por cero.

Requisitos específicos:

- 1. Diseñar una función llamada `calcularEficiencia` que reciba como parámetros la cantidad de productos terminados (int) y la materia prima utilizada (int).**
- 2. Incluir una validación dentro de la función que verifique si la materia prima es igual a 0. En caso afirmativo, debe mostrar un mensaje de error por pantalla y retornar 0.**
- 3. Si el valor es válido, realizar la operación utilizando el operador `static_cast<double>` para asegurar una división decimal exacta.**
- 4. En la función principal (main), solicitar al usuario los datos por teclado, invocar a la función e imprimir el índice de eficiencia resultante.**





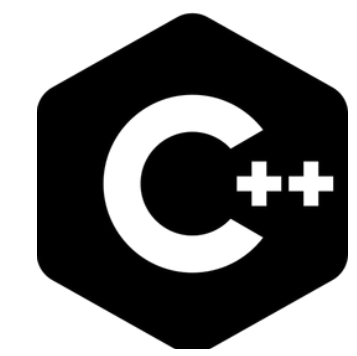
EJEMPLO 6:

Desarrollar un programa en C++ que utilice una función con retorno de tipo double y una estructura repetitiva para calcular el costo total de mantenimiento sumando iterativamente el costo por hora según la cantidad de horas trabajadas.

Requisitos específicos:

- 1. Diseñar una función llamada costoMantenimiento que reciba las horas de mantenimiento (int) y el costo por hora (double).**
- 2. Implementar dentro de la función una estructura repetitiva (bucle for) que acumule de forma secuencial el costo total sumando el valor por hora tantas veces como horas de mantenimiento se hayan indicado.**
- 3. Retornar el acumulador resultante (double) desde la función.**
- 4. En la función principal (main), solicitar al usuario que ingrese por teclado las horas de mantenimiento y el costo por hora, invocar a la función e imprimir el costo total final por pantalla.**





EJEMPLO 7:

Desarrollar un programa en C++ que utilice una función de tipo void junto con una estructura repetitiva condicional (while) para validar de forma iterativa el ingreso de una temperatura en un rango seguro de operación.

Requisitos específicos:

- 1. Crear una función llamada monitorearTemperatura que no reciba parámetros ni retorne valores (void).**
- 2. Solicitar al usuario que ingrese la temperatura actual del proceso.**
- 3. Implementar un bucle while que se mantenga activo mientras la temperatura ingresada sea igual o mayor a 200. Dentro del bucle, mostrar una alerta de advertencia y volver a solicitar el ingreso de una nueva lectura por teclado.**
- 4. Una vez que el valor sea inferior al límite seguro, salir del bucle y mostrar un mensaje de confirmación indicando que la temperatura se encuentra en rango.**
- 5. Invocar correctamente a la función monitorearTemperatura desde la función principal (main).**



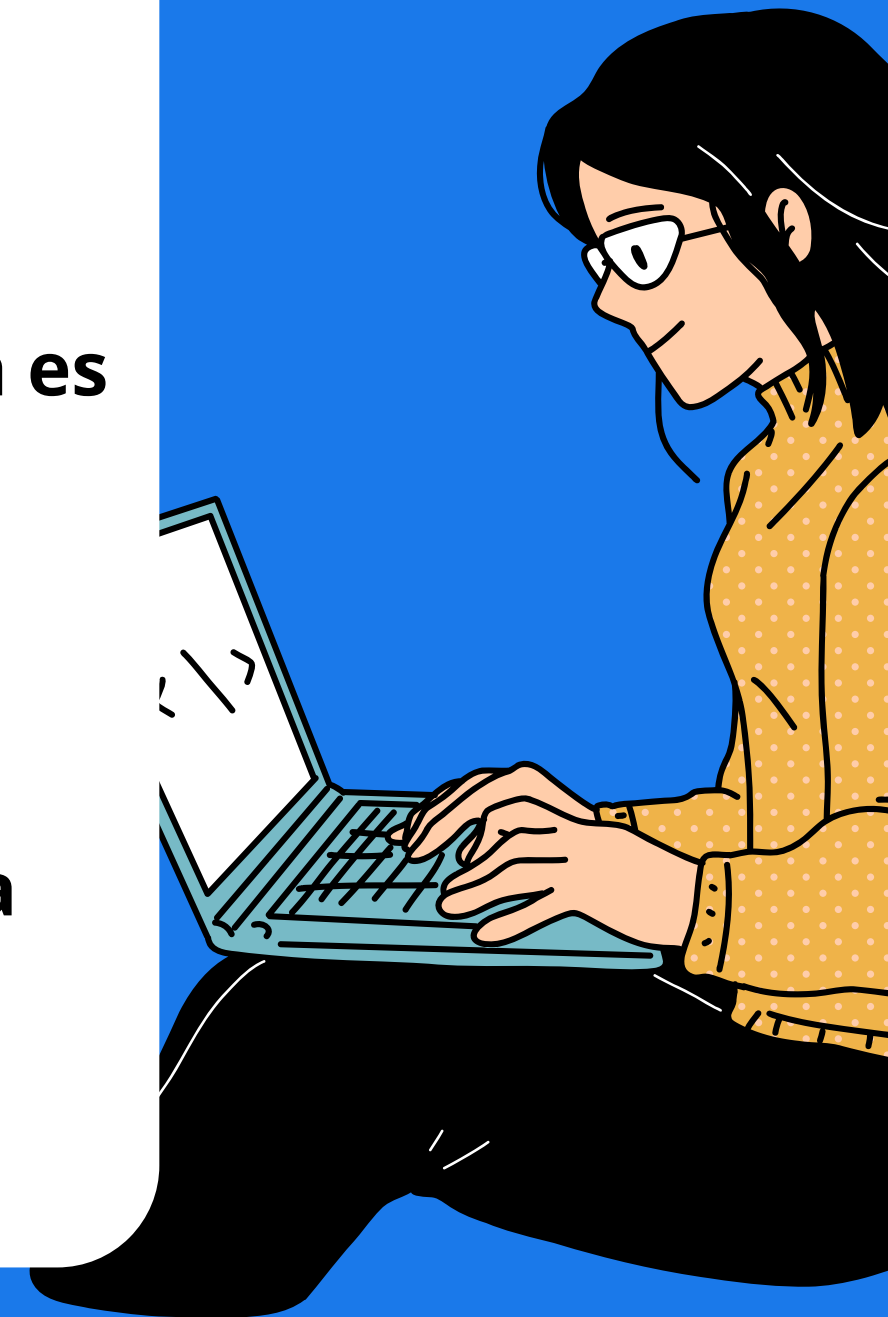


EJEMPLO 8: CÁLCULO DE RENDIMIENTO PORCENTUAL DE PROCESO

Desarrollar un programa en C++ que utilice una función con retorno de tipo double para calcular el rendimiento porcentual de un proceso productivo, incorporando una validación previa para evitar divisiones por cero o valores negativos de materia prima.

Requisitos específicos:

- 1. Diseñar una función llamada calcularRendimiento que reciba como parámetros la materia prima y el producto final (ambos de tipo double).**
- 2. Incluir una condición dentro de la función que verifique si la materia prima es menor o igual a 0. En caso afirmativo, debe retornar 0.0 para evitar errores matemáticos.**
- 3. Si el valor es correcto, calcular y retornar el porcentaje de rendimiento utilizando la fórmula correspondiente multiplicada por 100.0.**
- 4. En la función principal (main), solicitar al usuario que ingrese por teclado la cantidad de materia prima (en kg) y de producto final obtenido (en kg).**
- 5. Invocar a la función, almacenar el resultado e imprimir el valor del rendimiento porcentual por pantalla con su respectiva unidad.**



¿PREGUNTAS?

```
    }  
    render() {  
      return (  
        <React.Fragment>  
          <div className="py-5">  
            <div className="container">  
              <Title name="our" title="product">  
                <div className="row">  
                  <ProductConsumer>  
                    {(value) => {  
                      console.log(value)  
                    }}  
                  </ProductConsumer>  
                </div>  
              </div>  
            </div>  
          </React.Fragment>  
        )  
      )  
    }  
  }  
}
```


MUCHAS
GRACIAS!!!

Nos Vemos en la Siguiente Clase