PROGRAMACION AVANZADA

ING. INDUSTRIAL

Esp. Ing.Prof.: Cristina Delia Cruz

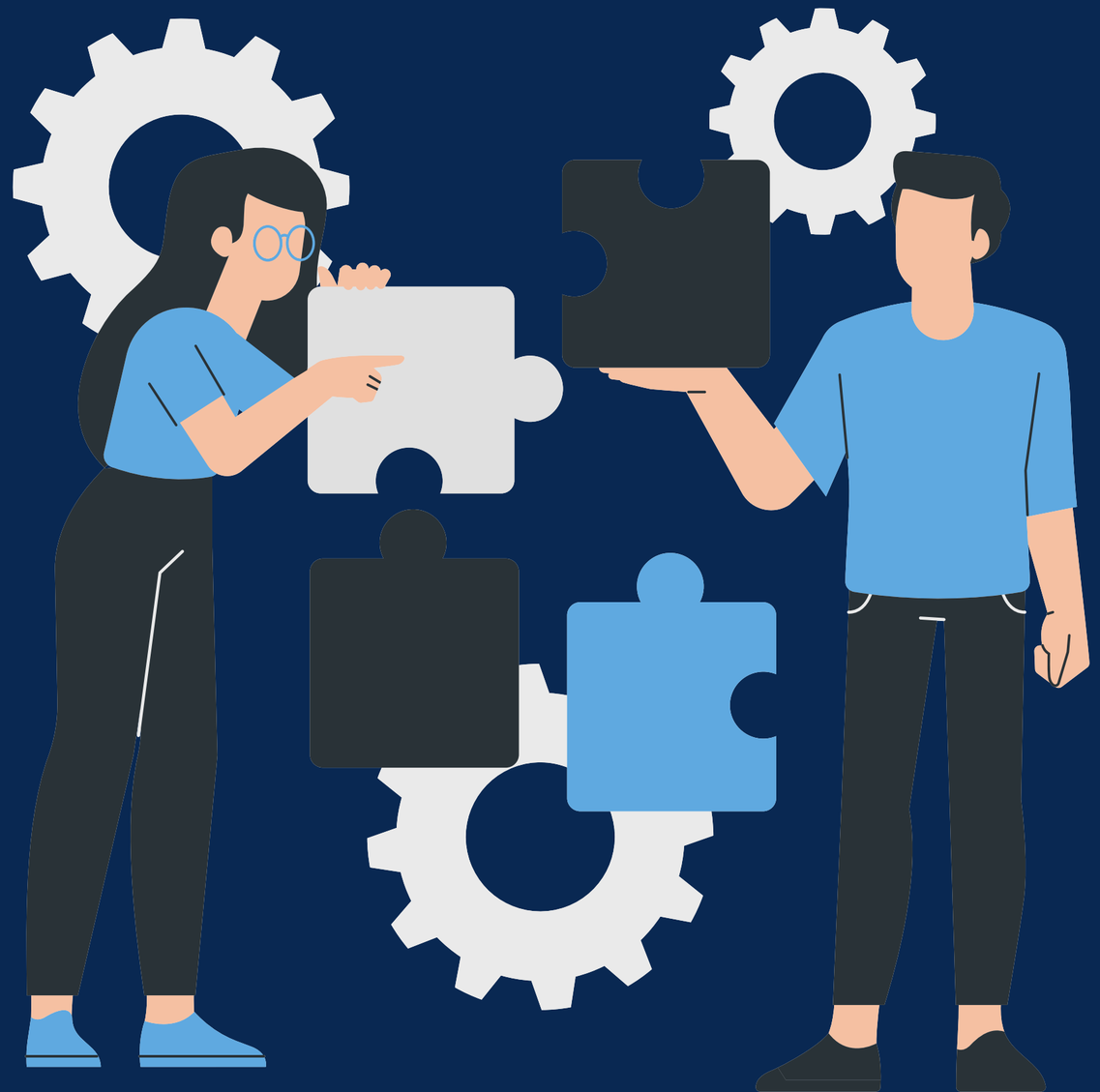
FACULTAD DE INGENIERIA
UNIVERSIDAD NACIONAL DE JUJUY

FUNCIONESEN 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

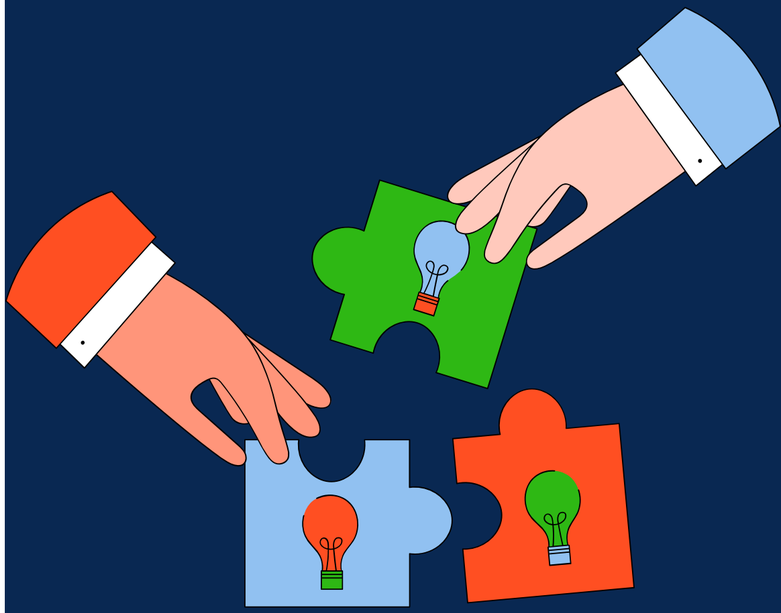
Características del diseño descendente:

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

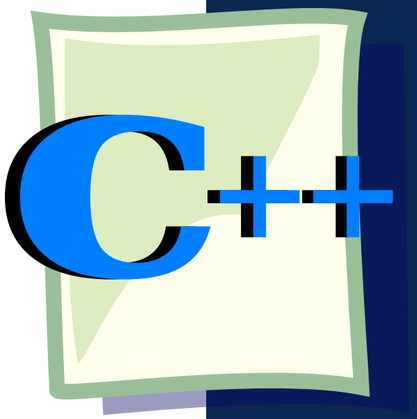
Desventajas:

- 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.

DISEÑO DESCENDETE

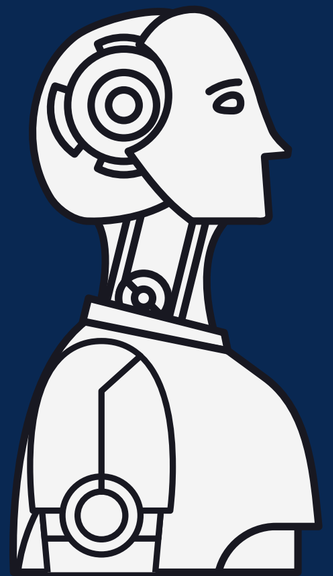
Ejemplo 1

```cpp
#include <iostream>
using namespace std;

// Submódulo para calcular el precio total
double calcularPrecio(double precio, int cantidad) {
    return precio * cantidad;
}

// Submódulo para calcular el impuesto
double calcularImpuesto(double subtotal, double tasaImpuesto) {
    return subtotal * tasaImpuesto;
}

// Módulo principal de facturación
double generarFactura(double precioProducto, int cantidad, double tasaImpuesto) {
    double subtotal = calcularPrecio(precioProducto, cantidad);
    double impuesto = calcularImpuesto(subtotal, tasaImpuesto);
    return subtotal + impuesto;
}
```

```cpp
int main() {
    double precioProducto = 100.0;
    int cantidad = 5;
    double tasaImpuesto = 0.21; // 21% de IVA

    double total = generarFactura(precioProducto, cantidad, tasaImpuesto);

    cout << "El total de la factura es: $" << total << endl;

    return 0;
}
```

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

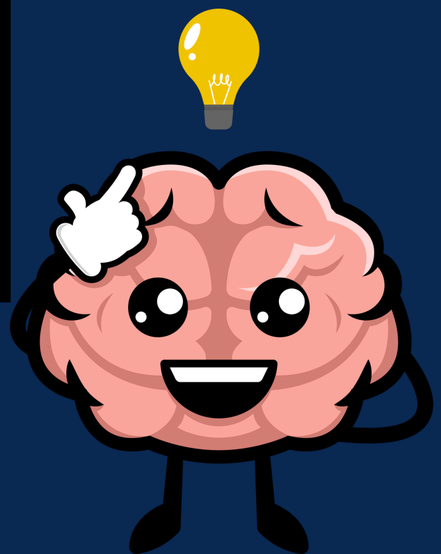
Características principales de una función:

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:

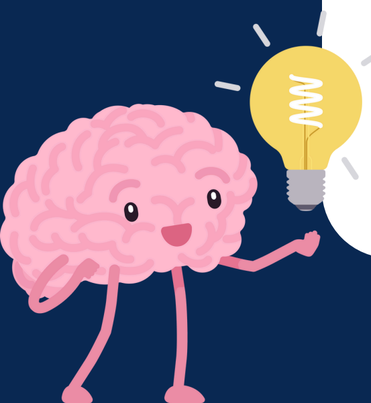
```
tipo_de_retorno nombre_de_función(tipo_parametro1
nombre_parametro1, tipo_parametro2
nombre_parametro2, ...) {
    // Bloque de código de la función
    return valor;  // opcional, si la función devuelve un
valor
}
```

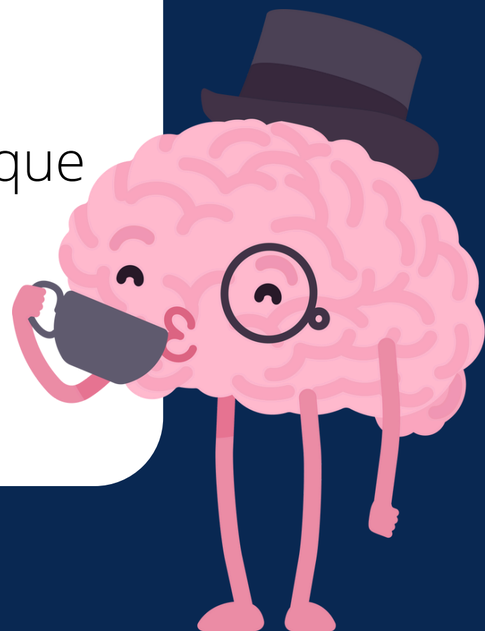


FUNCIONES

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.

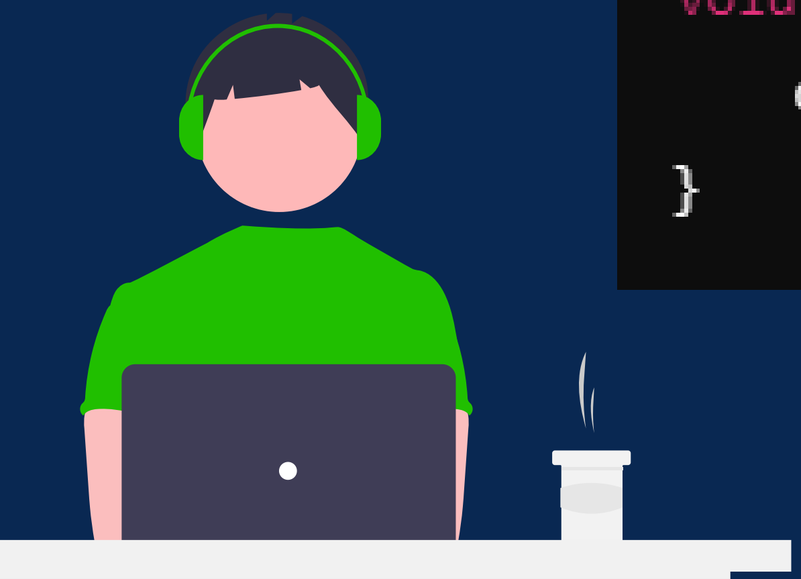




TIPOS DE FUNCIONES

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;
}
```

TIPOS DE FUNCIONES

Ejemplo 2

Hagamos un Programa Simple, que es la Suma de dos números, pero con funciones solamente, así vemos como funciona

FUNCIONES

Ejemplo 2

```cpp
#include <iostream>
using namespace std;

// Definición de una función que suma dos números
int sumar(int num1, int num2) {
    return num1 + num2;  // Retorna la suma de los dos números
}

int main() {
    int a = 5;
    int b = 3;

    // Llamamos a la función sumar y almacenamos el resultado
    int resultado = sumar(a, b);

    // Mostramos el resultado
    cout << "La suma de " << a << " y " << b << " es: " << resultado << endl;

    return 0;
}
```

FUNCIONES

Ejemplo 3

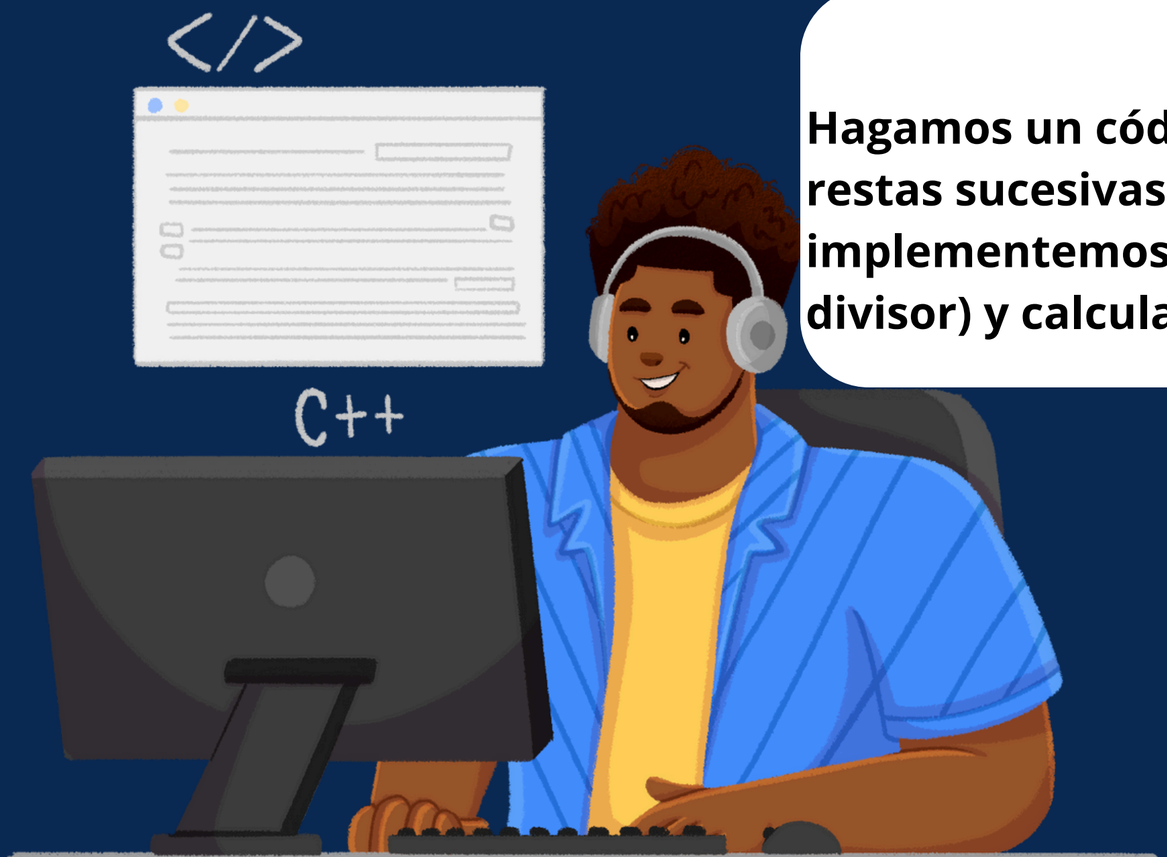
Hagamos un ejemplo sencillo de una función en C++ que realiza divisiones entre números reales (de tipo double). Esta función tomará dos números como parámetros y devolverá el resultado de la división, con un control para evitar la división por cero.

FUNCIONES

Ejemplo 3

```cpp
#include <iostream>
using namespace std;

double dividir(double num1, double num2) {
    if (num2 == 0) {
        cout << "Error: No se puede dividir por cero." << endl;
        return 0;
    } else {
        return num1 / num2;
    }
}

int main() {
    double a, b;

    cout << "Introduce el primer número (dividendo): ";
    cin >> a;
    cout << "Introduce el segundo número (divisor): ";
    cin >> b;

    double resultado = dividir(a, b);
    if (b != 0) {
        cout << "El resultado de la división es: " << resultado << endl;
    }

    return 0;
}
```

FUNCIONES



Ejemplo 4

Hagamos un código en C++ que realice una división utilizando restas sucesivas y bucles de repetición. Para este ejemplo, implementemos una función que recibe dos números (dividendo y divisor) y calcular el cociente mediante restas sucesivas

FUNCIONES

Resolución Ejemplo 4

```cpp
#include <iostream>
using namespace std;

void divisionPorRestas(int dividendo, int divisor, int &cociente, int &resto);

int main() {
    int dividendo, divisor, cociente, resto;

    cout << "Ingrese el dividendo: ";
    cin >> dividendo;

    cout << "Ingrese el divisor: ";
    cin >> divisor;

    if (divisor == 0) {
        cout << "Error: el divisor no puede ser cero." << endl;
        return 1;
    }

    divisionPorRestas(dividendo, divisor, cociente, resto);

    cout << "El cociente es: " << cociente << endl;
    cout << "El resto es: " << resto << endl;

    return 0;
}

void divisionPorRestas(int dividendo, int divisor, int &cociente, int &resto) {
    cociente = 0;

    while (dividendo >= divisor) {
        dividendo -= divisor;
        cociente++;
    }

    resto = dividendo;
}
```

FUNCIONES

Ejemplo 5

Hagamos un programa en C++ que calcule el factorial de un número usando sumas sucesivas para realizar las multiplicaciones.

FUNCIONES

Resolución Ejemplo 5

```cpp
#include <iostream>
using namespace std;

int productoPorSumas(int a, int b);
int factorial(int n);

int main() {
    int numero;

    cout << "Ingrese un número para calcular su factorial: ";
    cin >> numero;

    if (numero < 0) {
        cout << "El factorial no está definido para números negativos." << endl;
    } else {
        int resultado = factorial(numero);
        cout << "El factorial de " << numero << " es: " << resultado << endl;
    }

    return 0;
}
```

```cpp
int productoPorSumas(int a, int b) {
    int resultado = 0;
    for (int i = 0; i < b; i++) {
        resultado += a;
    }
    return resultado;
}

int factorial(int n) {
    int resultado = 1;
    for (int i = 2; i <= n; i++) {
        resultado = productoPorSumas(resultado, i);
    }
    return resultado;
}
```

FUNCIONES

Ejemplo 6

Hagamos un Programa en C++ que utiliza un bucle para realizar una operación matemática. Este programa suma los primeros N números naturales (por ejemplo, 1 + 2 + 3 + ... + N)

FUNCIONES

Resolución Ejemplo 6

```cpp
#include <iostream>
using namespace std;

int sumaNaturales(int n);

int main() {
    int numero;

    cout << "Ingrese un número para sumar todos los números naturales hasta él: ";
    cin >> numero;

    if (numero < 0) {
        cout << "Por favor, ingrese un número positivo." << endl;
    } else {
        int resultado = sumaNaturales(numero);
        cout << "La suma de los números naturales hasta " << numero << " es: " << resultado << endl;
    }

    return 0;
}

int sumaNaturales(int n) {
    int suma = 0;
    for (int i = 1; i <= n; i++) {
        suma += i;
    }
    return suma;
}
```

FUNCIONES

Ejemplo 7

Hagamos un programa en C++ que determina si un número es primo o no usando funciones. El programa incluye una función esPrimo que verifica si un número es primo y una función principal para manejar la entrada y salida.

FUNCIONES

Resolución Ejemplo 7

```cpp
#include <iostream>
using namespace std;

// Función para verificar si un número es primo
bool esPrimo(int numero) {
    if (numero <= 1) {
        return false;  // Los números menores o iguales a 1 no son primos
    }
    for (int i = 2; i <= numero / 2; ++i) {
        if (numero % i == 0) {
            return false;  // Si el número es divisible por otro número, no es primo
        }
    }
    return true;  // Si no es divisible por ningún número aparte de 1 y él mismo, es primo
}
```

```cpp
int main() {
    int numero;

    cout << "Ingrese un número para verificar si es primo: ";
    cin >> numero;

    if (esPrimo(numero)) {
        cout << numero << " es un número primo." << endl;
    } else {
        cout << numero << " no es un número primo." << endl;
    }

    return 0;
}
```

FUNCIONES

Ejemplo 8

Hagamos un ejemplo de un programa en C++ que realiza una multiplicación utilizando sumas sucesivas y emplea parámetros por referencia para obtener el resultado. La función multiplicar toma dos números enteros como parámetros y calcula el producto mediante sumas sucesivas, actualizando el resultado a través de un parámetro por referencia.

FUNCIONES

Resolución Ejemplo 8

```cpp
#include <iostream>
using namespace std;

// Función para realizar la multiplicación mediante sumas sucesivas
void multiplicar(int a, int b, int &resultado) {
    resultado = 0; // Inicializar resultado
    for (int i = 0; i < b; ++i) {
        resultado += a; // Sumar 'a' a resultado 'b' veces
    }
}
```

```cpp
int main() {
    int num1, num2, resultado;

    // Solicitar al usuario dos números
    cout << "Ingrese el primer número: ";
    cin >> num1;
    cout << "Ingrese el segundo número: ";
    cin >> num2;

    // Llamar a la función de multiplicación
    multiplicar(num1, num2, resultado);

    // Mostrar el resultado
    cout << "El producto de " << num1 << " y " << num2 << " es: " << resultado << endl;

    return 0;
}
```


¿PREGUNTAS?

MUCHAS

GRACIAS!!!

Nos Vemos en la Siguiente Clase