UNSA
Universidad Nacional de San Agustín

FACULTAD DE INGENIERÍA DE PRODUCCIÓN Y SERVICIOS

ESCUELA PROFESIONAL DE INGENIERÍA DE SISTEMAS

CURSO:

TECNOLOGÍA DE OBJETOS

TEMA:

FUNCIONES EN C++

DOCENTE:

MARIBEL MOLINA BARRIGA

INTEGRANTES:

Coaquira Suyo Gabriela Dayana

Nina Calizaya Rafael Diego

Villafuerte Quispe Alexander

Venero Guevara Christian Henry

Quispe Saavedra Dennis Javier

AREQUIPA - PERÚ
2025

FUNCIONES EN C++

1. Ejercicio 1

Las funciones inline permiten que el compilador reemplace la llamada a una función por el contenido de la misma directamente en el código. Esto evita el salto de pila y mejora el rendimiento, especialmente en funciones pequeñas como el cálculo de áreas. En esta práctica creamos tres funciones inline para calcular el área de un cuadrado, un rectángulo y un círculo, lo que permite realizar operaciones matemáticas simples de forma rápida y eficiente. Además, implementamos un menú interactivo que permite al usuario elegir qué figura desea calcular.

C/C++

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

// Funciones inline

inline double areaRectangulo(double b, double h) {
    return b*h;
}

inline double areaCirculo(double r) {
    return 3.14*r*r;
}

inline double areaCuadrado(double l) {
    return l*l;
}

int main() {
    int opcion;
    double a, b;

    cout << "Seleccione la figura:\n";
    cout << "1. Cuadrado\n2. Rectangulo\n3. Circulo\n";
    cin >> opcion;

    if (opcion == 1) {
        cout << "Ingrese lado: ";
        cin >> a;
        cout << "Area del cuadrado: " << areaCuadrado(a) << endl;
    } else if (opcion == 2) {
        cout << "Ingrese base y altura: ";
        cin >> a >> b;
        cout << "Area del rectangulo: " << areaRectangulo(a, b) << endl;
    } else if (opcion == 3) {
        cout << "Ingrese radio: ";
        cin >> a;
        cout << "Area del circulo: " << areaCirculo(a) << endl;
    } else {
```

```
        cout << "La opcion no es valida\n";
    }
    return 0;
}
```

2. Ejercicio 2

En este ejercicio se nos pide crear la Clase Caja con atributo privado peso e implementar una función friend llamada comparar que indica cual caja pesa más.

Reto adicional: Permitir comparar tres cajas

Solución:

En el código, la clase Caja declara el peso (peso) como un atributo privado. Por lo general, solo las funciones dentro de la clase (Caja) pueden manipular este dato.

```
C/C++
class Caja {
private:
    double peso; // Atributo requerido
    string nombre;
```

Para que la función comparar pueda ver el peso, se le declara como amiga dentro de la clase. Esta declaración es el "permiso especial".

```
C/C++
public:
    Caja() : peso(0.0), nombre("Caja_Vacia") {}
    Caja(double p, string n) : peso(p), nombre(n) {}
    // Declaración de la función 'comparar' como amiga de la clase Caja//
    friend void comparar(Caja c1, Caja c2, Caja c3);
    //-----------------------------------------------
};
```

Una vez declarada como amiga, la función comparar puede usar el operador punto (.) para acceder al atributo privado peso de cualquier objeto Caja que reciba.

```
C/C++
void comparar(const Caja& c1, const Caja& c2, const Caja& c3 = Caja()) {
// ... (El cuerpo de la función verifica si c3 está "vacía") }
```

3. Ejercicio 3: Funciones Constantes – Inmutabilidad

En este ejercicio se creó la clase Empleado y su método mostrarDatos() pero definido como const, esto debido a que esta es una función que no tiene como objetivo modificar los atributos o el comportamiento del objeto, por lo que para asegurar inmutabilidad se realiza esta práctica.

Reto adicional: Mostrar un mensaje si se intenta modificar los atributos dentro de la función const.

Esto no seria posible, ya que si se intenta modificar los atributos dentro de una función const, esto lanzará un error de compilación, evitando la ejecución del programa y por ende la posibilidad de mostrar un posible mensaje de advertencia, ya que el compilador maneja esto.

C/C++

```
#include <iostream>
#include <string>

using namespace std;

class Empleado {
private:
    string nombre;
    int edad;
    int salario;

public:
    Empleado(string nombre, int edad, int salario)
        : nombre(nombre), edad(edad), salario(salario) {}

    // Creando una funcion const que no modificara los atributos de clase
    // Al ser una funcion que no debe alterar el estado del objeto, se
declara como const
    void mostrarDatos() const {
        cout << "Nombre: " << nombre << endl;
        cout << "Edad: " << edad << endl;
        cout << "Salario: " << salario << endl;
    }
};

int main() {
    Empleado emp1("Juan Perez", 30, 50000);
    Empleado emp2("Ana Gomez", 28, 60000);
    Empleado emp3("Luis Martinez", 35, 70000);

    Empleado empleados[] = {emp1, emp2, emp3};

    for(int i = 0; i < 3; i++) {
        cout << "EMPLEADO " << (i + 1) << ":" << endl;
```

```
        empleados[i].mostrarDatos();
        cout << endl;
    }

    return 0;
}
```

4. Ejercicio 4

En este ejercicio se nos pide primeramente crear un vector<int> con al menos 10 números, para esto se incluyó la cabecera <random>, para así poder crear ese vector de tamaño 10 con números aleatorios. Además se creó la función de imprimirVector para que se mostrara en consola los números guardados en el vector.

C/C++

```
#include <iostream>
#include <vector>
#include <numeric>
#include <algorithm>
#include <random>

using namespace std;

void imprimirVector(const vector<int>& vec, const string& titulo) {
    cout << titulo << ": ";
    for (int num : vec) {
        cout << num << " ";
    }
    cout << endl;
}

int main() {
    std::mt19937 generador(std::random_device{}());
    std::uniform_int_distribution<> distribucion(1, 100);

    std::vector<int> numeros;
    const int tamano = 10;
    for (int i = 0; i < tamano; ++i) {
        numeros.push_back(distribucion(generador));
    }
    imprimirVector(numeros, "Vector inicial");
```

Luego se nos pedía cumplir tres funciones principales usando funciones lambda. La primera es contar los números pares del vector, para esto se creó la lambda esPar, la cual toma un entero 'n' y retorna positivo si este es par, usando el operador módulo para comprobarlo. Luego se aplicó un std::count_if, el cual aplica la lambda a cada elemento, cuenta los que salen pares y guarda ese número en la variable cuentaPares. Después solo queda imprimirlo en la consola.

```C/C++
    auto esPar = [](int n) {
        return n % 2 == 0;
    };

    int cuentaPares = count_if(numeros.begin(), numeros.end(), esPar);

    cout << "1. Contador de numeros pares:" << endl;
    cout << " Resultado: " << cuentaPares << " numeros pares." << endl;
    cout << endl;
```

En segundo lugar, se nos pidió calcular la suma total de los números del vector. Para esto, se usó la lambda sumador, que pasa a std::accumulate. Esta recibe el valor acumulado hasta el momento y el elemento actual del vector. Luego simplemente va sumando el acumulador con cada elemento, calculando al final el total de todos los elementos. Después solo queda imprimirlo en la consola.

```C/C++
    auto sumaTotal = accumulate(numeros.begin(), numeros.end(), 0, [](int
acumulador, int elemento) {
            return acumulador + elemento;
        } );

    cout << "2. Calculo de la suma total:" << endl;
    cout << " Resultado: " << sumaTotal << endl;
    cout << endl;
```

En tercer lugar, se nos pidió mostrar los números del vector que son mayores a un valor dado. Para esto, se definió a 10 como el valorDado y luego se incluyó la lambda esMayor, la cual recibe un número n y solo lo imprime si este es estrictamente mayor que el valor dado, el std::for_each aplica esta impresión condicional a cada elemento del vector.

```C/C++
    int valorDado = 10;
    cout << "3. Numeros mayores que " << valorDado << ":" << endl;
    cout << " Resultado: ";

    auto esMayor = [valorDado](int n) {
```

```
        if (n > valorDado) {
            cout << n << " ";
        }
    };

    for_each(numeros.begin(), numeros.end(), esMayor);
    cout << endl;
    cout << endl;
```

Por último, en el reto adicional se nos pidió ordenar el vector con std::sort y una lambda personalizada. Para esto se creó la función lambda personalizada comparadorAscendente, esta recibe dos elementos a y b, y retorna si a es menor a b, indicando al std::sor' que ordene el vector de manera ascendente, de menor a mayor. Al final, solo queda imprimir el vector ya ordenado y todo culmina exitosamente.

C/C++
```
    auto comparadorAscendente = [](int a, int b) {
        return a < b;
    };

    sort(numeros.begin(), numeros.end(), comparadorAscendente);

    cout << "Reto Adicional: Vector Ordenado de menor a mayor:" << endl;
    imprimirVector(numeros, " Resultado");
    return 0;
}
```

Se realizó un par de pruebas para comprobar el funcionamiento del código:

prueba

```
21:18:49: Starting C:\Users\LENOVO\Documents\prueba\build\Desktop_Qt_6_10_0_MinGW_64_bit-Debug\prueba.exe...
Vector inicial: 14 71 96 90 64 58 17 99 57 54
1. Contador de numeros pares:
 Resultado: 6 numeros pares.

2. Calculo de la suma total:
 Resultado: 620

3. Numeros mayores que 10:
 Resultado: 14 71 96 90 64 58 17 99 57 54

Reto Adicional: Vector Ordenado de menor a mayor:
 Resultado: 14 17 54 57 58 64 71 90 96 99
```

```
prueba

21:19:14: Starting C:\Users\LENOVO\Documents\prueba\build\Desktop_Qt_6_10_0_MinGW_64_bit-Debug\prueba.exe...
Vector inicial: 34 6 66 34 74 5 34 86 96 15
1. Contador de numeros pares:
 Resultado: 8 numeros pares.

2. Calculo de la suma total:
 Resultado: 450

3. Numeros mayores que 10:
 Resultado: 34 66 34 74 34 86 96 15

Reto Adicional: Vector Ordenado de menor a mayor:
 Resultado: 5 6 15 34 34 34 66 74 86 96
```

5. Ejercicio 5

La sobrecarga de operadores en C++ permite redefinir el comportamiento de operadores como +, ==, entre otros, para que funcionen con objetos de clases personalizadas. En esta práctica creamos una clase Vector2D con atributos x y y, y se sobrecarga el operador + para que pueda sumar dos vectores de forma intuitiva. Esto hace que el código sea más legible y natural, como si se estuviera trabajando con tipos de datos básicos.

También se implementamos la sobrecarga del operador == para comparar si dos vectores son iguales. Esta funcionalidad adicional permite verificar equivalencias entre objetos sin necesidad de escribir funciones externas.

C/C++

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

class Vector2D {
private:
    double x;
    double y;

public:
    Vector2D(double x_0, double y_0) : x(x_0), y(y_0) {}

    // operador +
    Vector2D operator+(const Vector2D& v) const {
        return Vector2D(x + v.x, y + v.y);
    }

    // operador ==
    bool operator==(const Vector2D& v) const {
        return x == v.x && y == v.y;
    }

    void mostrar() const {
        cout << "(" << x << ", " << y << ")\n";
    }
```

```
};

int main() {
    Vector2D v1(3.5, 2.0);
    Vector2D v2(1.5, 4.0);

    cout << "Vector 1: ";
    v1.mostrar();
    cout << "Vector 2: ";
    v2.mostrar();

    Vector2D v3 = v1 + v2;

    cout << "Vector 1 + Vector 2: ";
    v3.mostrar();

    if (v1 == v2)
        cout << "Los vectores son iguales\n";
    else
        cout << "Los vectores son diferentes\n";

    return 0;
}
```

6. Ejercicio 6

La práctica 06 nos pide crear la clase Fecha con atributos día, mes, año y luego sobrecargar el operador "<<" para mostrar la fecha con determinado formato.

Reto adicional: Validar que el día y mes sean válidos (1–31, 1–12).

Solución:

El operador << debe implementarse como una función global y debe devolver una referencia a ostream (el flujo de salida, típicamente cout).

```
C/C++
ostream& operator<<(ostream& os, const Fecha& f) {
    // ...
    return os;
    //Devuelve el flujo para permitir encadenar operaciones (cout << f1 <<
f2;)
}
```

Para garantizar que el día y el mes siempre tengan dos dígitos (ej. 05 en lugar de 5), el código utiliza herramientas de formato de la librería <iomanip>:

- std::setfill('0'): Indica que el relleno se haga con ceros.
- std::setw(2): Indica que el siguiente valor debe ocupar un ancho de 2 caracteres.

```C/C++
ostream& operator<<(ostream& os, const Fecha& f) {
    // Se usa el formato para imprimir 05 en lugar de 5
    os << setfill('0') << setw(2) << f.dia << "/"
       << setfill('0') << setw(2) << f.mes << "/"
       << f.anio;
    return os;
}
```

El reto consiste en asegurar que los datos de la fecha sean lógicamente válidos (ej. un mes no puede ser 13, ni abril tener 31 días).El código implementa una función interna (validarFecha) que se ejecuta en el constructor. Si el constructor detecta un error de lógica, establece una fecha por defecto (01/01) y notifica al usuario, evitando que se almacenen datos imposibles en el objeto.

```C/C++
Fecha(int d, int m, int a) {
    if (validarFecha(d, m)) {
        // ... asigna los valores si son correctos
    } else {
        // Si falla la validación, asigna un valor seguro (1/1)
        dia = 1;
        mes = 1;
        cout << "[AVISO: Fecha no es válida. Usando 01/01/...]" << endl;
    }
    anio = a;
}
```

7. Ejercicio 7

a. Función Inline para Calcular Descuento

La palabra clave inline sugiere al compilador que inserte el código de la función directamente donde se llama, evitando la sobrecarga de una llamada a función tradicional. Esta función calcula el monto del descuento multiplicando el precio por el porcentaje dividido entre 100. Se utiliza en el método aplicarDescuento() de la clase Producto para determinar cuánto dinero se reduce del precio original.

```C/C++
inline double calcularDescuento(double precio, double porcentajeDescuento) {
    return precio * (porcentajeDescuento / 100.0);
}
```

b. Clase con Atributos Privados y Función Friend

Los atributos nombre, precio y cantidad están declarados como privados, lo que significa que solo pueden ser accedidos desde dentro de la clase. Sin embargo, la función compararPrecios() está declarada como friend, lo que le otorga permiso especial para acceder directamente a los atributos privados precio de ambos productos sin necesidad de usar getters. Esto permite comparar precios de forma eficiente manteniendo el encapsulamiento.

```C/C++
class Producto {
private:
    string nombre;
    double precio;
    int cantidad;

    friend bool compararPrecios(const Producto& p1, const Producto& p2);
};

bool compararPrecios(const Producto& p1, const Producto& p2) {
    return p1.precio < p2.precio;
}
```

c. Funciones Const para Mostrar Datos

La palabra clave const al final de estas funciones indica que no modifican el estado del objeto. Esto garantiza que métodos como getNombre(), getPrecio() y mostrarInfo() solo leen datos sin alterarlos. El compilador verifica esta restricción y genera un error si intentamos modificar cualquier atributo dentro de una función const, promoviendo así un código más seguro y predecible.

```C/C++
string getNombre() const { return nombre; }
double getPrecio() const { return precio; }
int getCantidad() const { return cantidad; }

double getPrecioTotal() const {
    return precio * cantidad;
}

void mostrarInfo() const {
    cout << "Producto: " << nombre << endl;
    cout << "Precio unitario: $" << precio << endl;
}
```

d. Lambda para Filtrar Productos

Se utilizan dos lambdas: la función externa filtrarPorPrecio y una lambda interna dentro de copy_if. La lambda interna [precioMax](const Producto& p) { return p.getPrecio() <= precioMax; } captura precioMax por valor y actúa como predicado para filtrar productos. Solo los productos cuyo precio sea menor o igual al máximo especificado son copiados al vector resultante, demostrando el uso de funciones anónimas para lógica de filtrado concisa.

```
C/C++
auto filtrarPorPrecio = [](const vector<Producto>& prods, double precioMax)
{
    vector<Producto> filtrados;
    copy_if(prods.begin(), prods.end(), back_inserter(filtrados),
            [precioMax](const Producto& p) { return p.getPrecio() <=
precioMax; });
    return filtrados;
};
```

e. Sobrecarga del Operador +

La sobrecarga del operador + permite sumar dos objetos Producto usando una sintaxis natural como producto1 + producto2. El operador crea un nuevo producto que combina los nombres concatenados, suma los precios unitarios y suma las cantidades de ambos productos. Este operador devuelve un nuevo objeto sin modificar los originales (por eso es const), permitiendo operaciones intuitivas entre productos.

```
C/C++
Producto operator+(const Producto& otro) const {
    string nombreCombinado = nombre + " + " + otro.nombre;
    double precioTotal = precio + otro.precio;
    int cantidadTotal = cantidad + otro.cantidad;
    return Producto(nombreCombinado, precioTotal, cantidadTotal);
}
```

f. Sobrecarga del Operador << para Salida

Esta sobrecarga del operador << permite imprimir un objeto Producto directamente usando cout << producto, similar a como se imprimen tipos básicos. Al ser declarada como friend, puede acceder a los atributos privados del producto. La función devuelve una referencia a ostream& para permitir encadenamiento de operadores (como cout << p1 << p2). Esto proporciona un formato de salida consistente y legible para todos los productos del sistema.

```
C/C++
friend ostream& operator<<(ostream& os, const Producto& p) {
    os << fixed << setprecision(2);
    os << "Nombre: " << p.nombre << endl;
    os << "Precio unitario: $" << p.precio << endl;
```

```
    os << "Cantidad: " << p.cantidad << endl;
    os << "Precio total: $" << p.getPrecioTotal() << endl;
    return os;
}
```