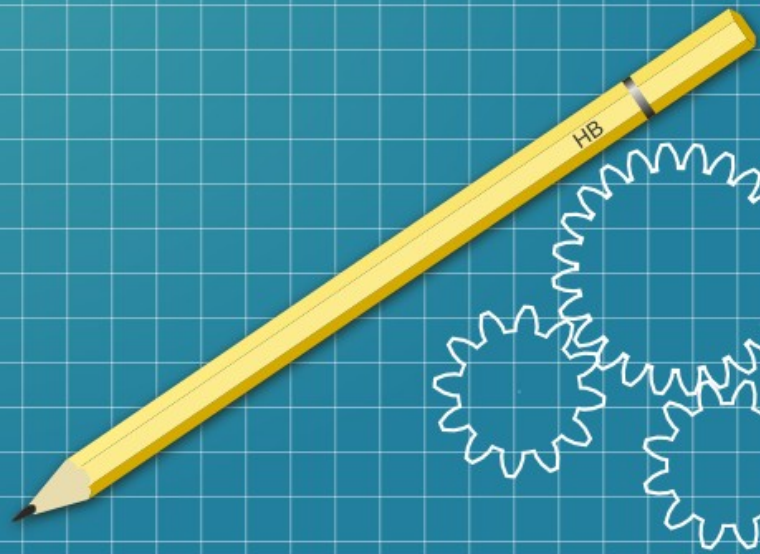
Programación orientada a objetos

```
#include<stdio.h>

int main(){
  int a = 10;
  int b;
  b ~= a;
  return 0;
}
```



Palomeque Nestor Levi

Programación orientada a objetos

La Programación Orientada a Objetos (POO) en C++ es un paradigma de programación que organiza el software en objetos que representan entidades del mundo real o conceptos abstractos. Estos objetos tienen atributos (datos) y comportamientos (funciones), que se agrupan en clases.

La POO permite reutilizar código a través de la herencia y bibliotecas de clases. Por ejemplo, la biblioteca estándar de C++ (STL) incluye clases reutilizables como std::vector, std::string, etc.

Nota: un **paradigma** en programación es un enfoque o estilo de resolver problemas y estructurar programas.

Analogía: receta y pastel

Una **clase** es una plantilla o modelo que define dos cosas principales: propiedades o atributos y comportamientos o métodos. Los primeros son los datos que describen el estado de los objetos creados a partir de la clase y los segundos son las funciones que determinan qué puede hacer un objeto de esa clase.

Un **objeto** es una instancia de una **clase**, es decir, es un elemento concreto creado a partir de esa plantilla (**clase**). Cada objeto puede tener sus propios valores para los atributos definidos por la **clase**.

Imagina que una **clase** es como una receta de cocina. La receta te dice qué ingredientes necesitas y los pasos para preparar un pastel. No es un pastel en sí mismo, solo es una guía para crear un pastel.

En la receta, los ingredientes son como los atributos de la **clase** (por ejemplo, harina, azúcar, huevos), que definen las características de los pasteles.

Los pasos de la receta son como los métodos de la **clase**, que indican cómo preparar el pastel (por ejemplo, mezclar, hornear, decorar).

Un **objeto** es el pastel real que haces siguiendo la receta. Cada vez que sigues la receta, obtienes un pastel diferente, aunque todos son creados a partir de la misma receta.

Puedes hacer varios pasteles (**objetos**) usando la misma receta (**clase**), pero cada pastel puede ser diferente (puedes usar diferentes colores de glaseado, agregar frutas distintas, etc.).

Separación entre la interfaz y la implementación

La interfaz es lo que los usuarios de la clase ven (definición de métodos y atributos públicos). La implementación es como se realizan internamente las operaciones de la clase (el código dentro de las funciones).

persona.h (Interface)

```
#ifndef PERSONA_H
#define PERSONA_H
#include <string>

class Persona {
public:
    std::string nombre;
    int edad;

    Persona(std::string nombre, int edad);
    void mostrarDatos() const;
};
#endif
```

persona.cpp (Implementación)

```
#include "persona.h"
#include <iostream>

//Constructor
Persona::Persona(std::string nombre, int edad) : nombre(nombre), edad(edad) {}

void Persona::mostrarDatos() const {
    std::cout << "Nombre: " << nombre << ", Edad: " << edad << std::endl;
}
```

Control de acceso a miembros

En C++, puedes especificar la visibilidad de los miembros (atributos y métodos) de una clase usando tres modificadores de acceso: public, private, y protected. Estos modificadores determinan qué partes del programa pueden acceder a los miembros de la clase.

- **public**: Los miembros son accesibles desde fuera y dentro de la clase.
- **private**: Los miembros solo son accesibles dentro de la propia clase. Se usa para ocultar los detalles internos y proteger la integridad de los datos. Los métodos **private** a menudo son usados internamente por la clase para realizar tareas específicas. No permite el acceso desde clases derivadas.
- **protected**: Los miembros declarados como **protected** son accesibles desde dentro de la propia clase y desde clases derivadas (subclases), pero no desde fuera de la clase.

```
class Persona {
private:
    std::string nombre; // Atributo privado

protected:
    int edad; // Atributo protegido

public:
    Persona(std::string n = "Desconocido", int e = 0);    // Constructor PREDETERMINADO: no requiere argumentos explícitos

    void setNombre(std::string n);       // Método público para cambiar el nombre
    std::string getNombre() const;       // Método público para obtener el nombre

    // Método público para acceder a la edad
    int getEdad() const;
};
```

Constructor y destructor

Un **constructor** es una función especial que se llama automáticamente cuando se crea un objeto de una clase. Su propósito principal es inicializar los datos miembros del objeto. Posee las siguientes características:

- Nombre: Tiene el mismo nombre que la clase y no tiene tipo de retorno, ni siquiera void.
- Sobrecarga: Se pueden definir múltiples constructores con diferentes listas de parámetros (sobrecarga de constructores).
- Inicialización: Permite inicializar los atributos de la clase al momento de la creación del objeto.

Un **destructor** es una función especial que se llama automáticamente cuando un objeto de una clase es destruido. Su propósito principal es liberar recursos (cierra archivos, desconecta redes, etc) y realizar limpieza necesaria antes de que el objeto sea eliminado de la memoria. Posee las siguientes características:

- Nombre: Tiene el mismo nombre que la clase precedido por un tilde (~), y no tiene parámetros ni tipo de retorno.
- No Sobrecargado: No se puede sobrecargar. Cada clase tiene un único destructor.
- Limpieza: Se utiliza para liberar memoria dinámica, cerrar archivos, y otras tareas de limpieza.

```
class Archivo {
private:
    std::string nombreArchivo;
    // Aquí podrías tener un puntero a recursos que necesiten limpieza

public:
    Archivo(std::string nombre);    // Constructor

    ~Archivo();     // Destructor

    // ...
};
```

Si no defines un constructor en una clase, el compilador proporcionará un constructor por **defecto** automáticamente. Este constructor inicializa los datos miembros de manera predeterminada, en cero para int y float por ejemplo.

De manera similar, si no defines un destructor en una clase, el compilador proporcionará un destructor predeterminado que realiza una destrucción básica:

- Libera recursos asignados automáticamente por el compilador
- Si la clase tiene punteros o **recursos dinámicos**, el destructor predeterminado no liberará estos recursos, lo que puede llevar a fugas de memoria.

Constructor, lista de inicialización

La lista de inicialización es una característica de C++ que te permite inicializar los miembros de una clase directamente en el momento en que el constructor es llamado, antes de que el cuerpo del constructor sea ejecutado. La sintaxis es:

```cpp
#include <string>

class Persona {
private:
    std::string nombre;
    int edad;

public:
    // Constructor con lista de inicialización
    Persona(std::string n, int e);

    void mostrarDatos() const;
};
```

```cpp
#include "Persona.h"
#include <iostream>

// Constructor con lista de inicialización
Persona::Persona(std::string n, int e) : nombre(n), edad(e) {
    // No es necesario inicializar 'nombre' y 'edad' aquí,
    // ya que ya se inicializaron en la lista de inicialización.
}

void Persona::mostrarDatos() const {
    std::cout << "Nombre: " << nombre << ", Edad: " << edad << std::endl;
}
```

Los **miembros** que sean **const** y **referencias**, de una clase, deben ser inicializados usando una lista de inicialización, ya que no pueden ser asignados después de que el objeto ha sido creado.

Constructor copia

Un constructor de copia es una función especial que se utiliza para crear un nuevo objeto como una copia de un objeto ya existente. Este constructor toma una referencia al objeto que se va a copiar y usa su estado para inicializar el nuevo objeto.

El constructor de copia es útil cuando:

- Necesitas crear una nueva instancia de un objeto a partir de una existente.
- La clase maneja recursos dinámicos (memoria, archivos, etc.), y necesitas asegurarte de que cada objeto tenga su propia copia de esos recursos, no solo una referencia compartida.

El compilador genera un constructor copia automáticamente el cuál realiza una copia superficial de los miembros de datos. Esto significa que simplemente copia los valores de los miembros del objeto original al nuevo objeto. Para tipos de datos simples (como int, float, std::string), esto suele estar bien, ya que se copia el valor. Si la clase maneja recursos dinámicos (como memoria asignada con new, punteros a archivos, etc.), el constructor de copia automático puede causar problemas. En estos casos, necesitarías un constructor de copia personalizado, que reciba una referencia constante a un objeto de la misma clase, para manejar la copia profunda o gestionar adecuadamente los recursos.

Ejemplo

Crearemos una clase **Persona** que posea nombre y edad. Luego implementaremos un método que imprima dichos datos.

```cpp
#ifndef PERSONA_H
#define PERSONA_H

#include <string>

class Persona {
private:
    std::string nombre;
    int edad;

public:

    Persona();      // Constructor predeterminado
    Persona(std::string n, int e);     // Constructor con parámetros
    ~Persona();    // Destructor

    void mostrarDatos() const;     // Método para mostrar datos
};
#endif // PERSONA_H
```

```cpp
#include "persona.h"
#include <iostream>

// Constructor predeterminado (usa lista de inicialización)
Persona::Persona() : nombre("Desconocido"), edad(0) {}

// Constructor con parámetros
Persona::Persona(std::string n, int e) {
    nombre = n;
    edad = e;
}

Persona::~Persona() { // Destructor
    // Si se gestionan recursos dinámicos, se liberarían aquí
}

// Método para mostrar datos
void Persona::mostrarDatos() const {
    std::cout << "Nombre: " << nombre << ", Edad: " << edad << std::endl;
}
```

Ejemplo, continuación

Veamos el main:

```
#include <iostream>
#include "persona.h"

int main() {
    Persona pers1;        //Uso el constructor predeterminado
    Persona pers2("Nestor", 38);   //Uso el constructor
    Persona pers3(pers2);       //Uso el constructor copia

    pers1.mostrarDatos();
    pers2.mostrarDatos();
    pers3.mostrarDatos();

    return 0;
}
```

El puntero **this**

El puntero **this** es un puntero implícito que está disponible en el contexto de todas las funciones miembro de una clase. Su propósito principal es proporcionar una referencia al objeto actual sobre el cual se está ejecutando el método.

El puntero **this** se utiliza principalmente para:

- Puedes usar **this** para acceder a los miembros del objeto dentro de una función miembro. Esto es especialmente útil cuando los nombres de los parámetros de la función miembro son iguales a los nombres de los miembros de la clase.
- Puedes usar **this** para devolver una referencia al objeto actual desde un método, lo que es útil para implementar el encadenamiento de métodos.

```
#include "persona.h"
#include <iostream>

Persona::Persona(const std::string& nombre, int edad) {
    this->nombre = nombre;
    this->edad = edad;
}

void Persona::establecerNombre(const std::string& nombre) {
    this->nombre = nombre;
}

std::string Persona::obtenerNombre() const {
    return nombre;
}

void Persona::mostrarInformacion() const {
    std::cout << "Nombre: " << nombre << ", Edad: " << edad << std::endl;
}
```

Miembros **const**

Los datos miembros const de una clase son atributos cuyo valor no puede cambiar una vez que se inicializan. Esto garantiza que el estado de un objeto permanezca inmutable después de la inicialización. Es necesario utilizar aquí la lista de inicialización para el miembro **const**.

```
#include <iostream>

class Constante {
public:
    const int valor;  // Miembro const

    Constante(int v) : valor(v) {}    // Constructor para inicializar el miembro const

    void mostrar() const {              // Método para mostrar el valor
        std::cout << "Valor: " << valor << std::endl;
    }
};
int main() {
    Constante c(42);  // Crear un objeto Constante
    c.mostrar();      // Imprimir el valor

    // c.valor = 100; // Error: valor es const y no puede ser modificado
    return 0;
}
```

Funciones miembros **const**

Una función miembro **const** es una función que no modifica el estado del objeto sobre el que se llama y solo pueden llamar a otras funciones miembro const. Se garantiza que dentro de esta función, los datos miembros no se modificarán. Observar la función mostrarDatos().

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

class Persona {
public:
    std::string nombre;
    int edad;
    Persona(const std::string& nombre, int edad) : nombre(nombre), edad(edad) {}          // Constructor

    void mostrarDatos() const {     // Función miembro const
        std::cout << "Nombre: " << nombre << ", Edad: " << edad << std::endl;
    }
    void cumplirAnios() {         // Función miembro que modifica el estado
        edad++; // Esta función puede modificar el estado del objeto
    }
};
int main() {
    Persona p("Alice", 30); // Crear un objeto Persona
    p.mostrarDatos();       // Mostrar los datos, no modifica el objeto

    p.cumplirAnios();       // Modificar el estado del objeto
    p.mostrarDatos();       // Mostrar los datos después de la modificación
    return 0;
}
```

Una función miembro **const** puede recibir parámetros, y puede modificar esos parámetros dentro de la función, siempre y cuando no modifique el estado del objeto sobre el cual se llama.

Objetos **const**

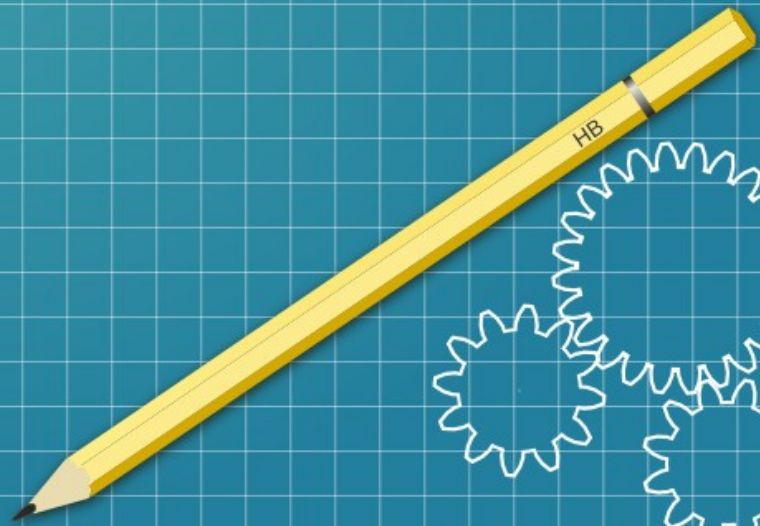
Un objeto **const** es un objeto que no puede ser modificado después de su creación. Esto significa que solo puede llamar a funciones miembro **const** y no se pueden cambiar los valores de los datos miembros **const**.

```
class Rectangulo {
private:
    int ancho;
    int alto;

public:
    Rectangulo(int a, int h) : ancho(a), alto(h) {}

    int obtenerArea() const {
        return ancho * alto;
    }
    void cambiarDimensiones(int nuevo_ancho, int nuevo_alto) {
        ancho = nuevo_ancho;
        alto = nuevo_alto;
    }
};
int main() {
    const Rectangulo rect(5, 10);
    std::cout << "Área: " << rect.obtenerArea() << std::endl;
    // rect.cambiarDimensiones(7, 12); // Error: cambiarDimensiones() no es const
    return 0;
}
```





HB