REGLAS GENERALES

Compilación

- Compila tu código con c++ y las banderas -Wall -Wextra -Werror
- Tu código aún debería compilar si agregas la bandera -std=c++98

Formato y convenciones de nomenclatura

- Los directorios de ejercicios se nombrarán de esta manera: ex00, ex01, ..., exn
- Nombra tus archivos, clases, funciones, funciones miembro y atributos según lo requerido en las pautas.
- Escribe los nombres de las clases en formato UpperCamelCase. Los archivos que contienen código de clase siempre se nombrarán de acuerdo con el nombre de la clase. Por ejemplo: ClassName.hpp/ClassName.h, ClassName.cpp o ClassName.tpp. Entonces, si tienes un archivo de encabezado que contiene la definición de una clase "BrickWall" que representa un muro de ladrillo, su nombre será BrickWall.hpp.

  Cuando trabajamos con clases en C++, es fundamental establecer una convención de nombres para los archivos que las contienen. Esto facilita la organización del código, la colaboración en equipos y la comprensión del proyecto a largo plazo.

  ¿Por qué es importante esta convención?

  - Claridad: Al relacionar directamente el nombre del archivo con el nombre de la clase, se hace evidente a simple vista dónde encontrar la definición y la implementación de una clase en particular.
  - Organización: Agrupar los archivos relacionados con una clase facilita la navegación y la gestión del proyecto.
  - Convención: Adoptar una convención común facilita la colaboración en equipos, ya que todos los miembros del equipo sabrán dónde buscar las definiciones de las clases.

  ¿Cuáles son las extensiones comunes?

  - .hpp o .h: Estos archivos suelen contener las declaraciones de la clase, es decir, los miembros (atributos y métodos) que la componen. Se les conoce como archivos de encabezado (header files) porque su contenido suele ser incluido en otros archivos mediante la directiva #include.
  - .cpp: Este archivo contiene la implementación de los métodos de la clase. Aquí es donde se escribe el código que define el comportamiento de los métodos.

  Imaginemos que tenemos una clase llamada Persona. Según la convención, los archivos relacionados con esta clase podrían ser:

  - Persona.hpp: Contiene la declaración de la clase Persona:

    ```
    class Persona {
    ```

```
public:
  std::string nombre;
  int edad;

  void saludar();
};
```

- Persona.cpp: Contiene la implementación del método saludar():

```
#include "Persona.hpp"

void Persona::saludar() {
  std::cout << "Hola, mi nombre es " << nombre << std::endl;
}
```

¿Qué significa .tpp?

La extensión .tpp se utiliza a veces para archivos que contienen la implementación de plantillas. Las plantillas son una característica avanzada de C++ que permite crear código genérico que puede trabajar con diferentes tipos de datos. Al separar la implementación de la plantilla en un archivo .tpp, se mejora la organización del código y se facilita la compilación.

Resumen:

.hpp/.h: Declaraciones de la clase.
.cpp: Implementaciones de los métodos de la clase.
.tpp: Implementaciones de plantillas.

¿Por qué es importante seguir esta convención?

Al seguir esta convención, estarás escribiendo código más limpio, organizado y fácil de mantener. Además, facilitarás la colaboración con otros programadores y te adaptarás a las prácticas comunes en el desarrollo de software en C++.

- A menos que se especifique lo contrario, todos los mensajes de salida deben terminar con un carácter de nueva línea y mostrarse en la salida estándar.
- ¡Adiós Norminette! No se aplica ningún estilo de codificación en los módulos de C++. Puedes seguir tu favorito. Pero ten en cuenta que un código que tus compañeros evaluadores no pueden entender es un código que no pueden calificar. Haz tu mejor esfuerzo para escribir un código limpio y legible.

Permitido/Prohibido

- Ya no estás programando en C. ¡Es hora de C++! Así que ¡: Puedes usar casi todo de la biblioteca estándar. Por lo tanto, en lugar de ceñirte a lo que ya conoces, sería inteligente usar lo más posible las versiones en C++ de las funciones C a las que estás acostumbrado.

- Sin embargo, no puedes usar ninguna otra biblioteca externa. Esto significa que las bibliotecas C++11 (y formas derivadas) y Boost están prohibidas. Las siguientes funciones también están prohibidas: *printf(), *alloc() y free(). Si las usas, tu calificación será 0 y eso es todo.
- Ten en cuenta que a menos que se indique explícitamente lo contrario, las palabras clave using namespace y friend están prohibidas. De lo contrario, tu calificación será -42.

¿Por qué se prohíbe using namespace std?

La directiva using namespace std; importa todos los identificadores del espacio de nombres std (la biblioteca estándar de C++) al ámbito actual. Esto significa que puedes usar nombres como cout, cin, vector, etc., sin tener que precederlos con std::.

¿Por qué es problemático?

- Colisiones de nombres: A medida que tu código crece y utilizas más bibliotecas, es más probable que haya nombres duplicados. Esto puede llevar a comportamientos inesperados y difíciles de depurar.
- Pérdida de claridad: Al ocultar el origen de los identificadores, dificulta la lectura y comprensión del código.
- Dependencia de la biblioteca estándar: Acopla tu código a la biblioteca estándar, lo que puede limitar su portabilidad o dificultar la sustitución por otras bibliotecas en el futuro.

¿Cuál es la alternativa?

Calificar los nombres: Siempre que uses un identificador de la biblioteca estándar, especifica su espacio de nombres: std::cout, std::vector, etc. Esto hace el código más claro y evita ambigüedades.

¿Por qué se prohíbe la palabra clave friend?

La palabra clave friend otorga a una función o clase acceso a los miembros privados de otra clase. Esto puede romper la encapsulación y hacer que el código sea más difícil de entender y mantener.

¿Por qué es problemático?

- Rompe la encapsulación: La encapsulación es un principio fundamental de la POO que ayuda a ocultar los detalles de implementación y proteger la integridad de los objetos. Al declarar una función como amiga, se está violando este principio.
- Aumenta el acoplamiento: Las clases amigas se vuelven dependientes entre sí, lo que dificulta la modificación y reutilización de código.
- Hace el código menos mantenible: Al permitir el acceso a los miembros privados, se complica la tarea de realizar cambios en una clase sin afectar a otras.

¿Cuál es la alternativa?

- Métodos públicos: Proporciona métodos públicos para acceder a la información necesaria, en lugar de exponer los miembros privados.
- Interfaces: Define interfaces claras para la interacción entre clases.
- Patrones de diseño: Utiliza patrones de diseño como el patrón Observer o el patrón Strategy para gestionar las relaciones entre objetos.

Prohibir el uso indiscriminado de using namespace std y friend fomenta la escritura de código más claro, mantenible y escalable. Al evitar estas prácticas, se promueve una mejor comprensión de los principios de la POO y se reduce la probabilidad de errores.

- Solo puedes usar STL en los Módulos 08 y 09. Esto significa: no Contenedores (vector/lista/mapa/etc.) y no Algoritmos (cualquier cosa que requiera incluir el encabezado) hasta entonces. De lo contrario, tu calificación será -42.

¿Qué es la STL?

La STL, o Standard Template Library, es una biblioteca de C++ que proporciona una amplia gama de herramientas para la programación genérica. Estas herramientas incluyen:

- Contenedores: Estructuras de datos como vector, list, map, set, etc. que permiten almacenar y organizar colecciones de elementos de forma eficiente.
- Algoritmos: Funciones genéricas que realizan operaciones comunes sobre contenedores, como ordenar, buscar, copiar, etc.
- Iteradores: Objetos que permiten acceder y recorrer los elementos de un contenedor de forma secuencial.

¿Por qué la restricción de usar STL solo en los módulos 8 y 9?

La razón detrás de esta restricción es que los primeros módulos buscan que adquieras una base sólida en los conceptos fundamentales de la programación orientada a objetos. Al prohibir el uso de la STL en un principio, se te obliga a:

- Implementar tus propias estructuras de datos: Esto te permite comprender a fondo cómo funcionan internamente las estructuras de datos y cómo elegir la más adecuada para cada situación.
- Escribir tus propios algoritmos: Al desarrollar tus propias funciones para ordenar, buscar, etc., adquieres un conocimiento más profundo de los algoritmos y su complejidad.
- Fortalecer tus habilidades de programación: Implementar tus propias soluciones te ayuda a desarrollar habilidades de diseño, resolución de problemas y pensamiento algorítmico.

¿Qué significa "no Contenedores" y "no Algoritmos"?

- No Contenedores: No puedes utilizar directamente los contenedores de la STL como vector, list, map, etc. Si necesitas almacenar una colección de elementos, deberás crear tu propia estructura de datos (por ejemplo, una clase Lista o Arbol).

- No Algoritmos: No puedes usar directamente los algoritmos de la STL como sort, find, copy, etc. Tendrás que implementar tus propias funciones para realizar estas operaciones.

¿Qué implica incluir el encabezado?

El encabezado algorithm contiene la declaración de muchos de los algoritmos de la STL. Al prohibir la inclusión de este encabezado, se refuerza la restricción de no utilizar los algoritmos de la STL.

A partir de los módulos 8 y 9, se te permitirá utilizar la STL. Esto significa que podrás aprovechar las ventajas de los contenedores y algoritmos predefinidos para resolver problemas de una manera más eficiente y con menos código.

Algunos requisitos de diseño

- Las fugas de memoria también ocurren en C++. Cuando asignes memoria (usando la palabra clave new), debes evitar las fugas de memoria.

Una fuga de memoria en C++ ocurre cuando se asigna memoria dinámicamente (utilizando el operador new) y luego se pierde la referencia a esa memoria. Es decir, el programa ya no puede acceder a esa porción de memoria, pero el sistema operativo sigue reservándola. Esto puede llevar a que el programa consuma cada vez más memoria hasta agotarla y provocar un fallo.

¿Por qué ocurren las fugas de memoria?

- Pérdida de punteros: El puntero que apuntaba a la memoria asignada se sobrescribe o se pierde, haciendo imposible liberar esa memoria.
- Ciclo de referencias: Cuando dos o más objetos se refieren circularmente entre sí, impidiendo que el recolector de basura (que C++ no tiene de forma automática) libere la memoria asociada.
- Errores en la gestión de la memoria: Utilizar incorrectamente las funciones new y delete, o no liberar la memoria en el momento adecuado, puede causar fugas.

Ejemplo de fuga de memoria:

```
#include <iostream>

int main() {
    int *ptr = new int;
    *ptr = 42;

    // Aquí debería liberar la memoria, pero no se hace
    // delete ptr;

    return 0;
}
```

En este ejemplo, se asigna memoria para un entero usando new, pero nunca se libera utilizando delete. Esto provoca una fuga de memoria, ya que el espacio reservado para el entero no se devuelve al sistema.

¿Cómo evitar las fugas de memoria?

- Emparejar new con delete: Por cada llamada a new, debe haber una llamada correspondiente a delete para liberar la memoria asignada.
- Gestión cuidadosa de los punteros: Asegúrate de que los punteros siempre apunten a una dirección de memoria válida y que se liberen cuando ya no sean necesarios.
- Utilizar contadores de referencias: En casos más complejos, puedes utilizar contadores de referencias para rastrear cuántas veces se hace referencia a un objeto y liberar la memoria solo cuando el contador llega a cero.
- Considerar Smart Pointers: La biblioteca estándar de C++ ofrece smart pointers (punteros inteligentes) como unique_ptr y shared_ptr que gestionan automáticamente la liberación de memoria, reduciendo el riesgo de fugas.

- Desde el Módulo 02 hasta el Módulo 09, tus clases deben diseñarse en la Forma Canónica Ortodoxa, excepto cuando se indique explícitamente lo contrario.

  La Forma Canónica Ortodoxa es una especie de plantilla o estructura recomendada para definir clases en C++. Esta estructura garantiza que tus clases estén bien definidas y se comporten de manera predecible en diversas situaciones. Se considera una buena práctica porque facilita la gestión de la memoria, la copia de objetos y la asignación, entre otras cosas.

  Elementos clave de la OCF:

  - Constructor por defecto: Un constructor que no toma ningún argumento y se encarga de inicializar los miembros de la clase a sus valores por defecto.
  - Constructor de copia: Un constructor que crea un nuevo objeto a partir de otro existente, copiando sus miembros.
  - Operador de asignación: Un operador que asigna el valor de un objeto a otro.
  - Destructor: Un método que se llama automáticamente cuando un objeto se destruye, liberando los recursos que haya asignado.

  ¿Por qué es importante la OCF?

  - Prevención de errores: Al seguir la OCF, reduces la posibilidad de errores comunes como fugas de memoria o copias superficiales.
  - Código más robusto: Un código que sigue la OCF es más fácil de entender, mantener y depurar.
  - Compatibilidad con la STL: Muchas de las clases de la Standard Template Library (STL) están diseñadas siguiendo la OCF, por lo que tus clases podrán interactuar con ellas de manera más natural.

  En este ejemplo, MiClase sigue la OCF al tener definidos los cuatro elementos clave:

```
class MiClase {
public:
  // Constructor por defecto
  MiClase() : miembro1(0), miembro2("") {}

  // Constructor de copia
  MiClase(const MiClase& otra) : miembro1(otra.miembro1),
miembro2(otra.miembro2) {}

  // Operador de asignación
  MiClase& operator=(const MiClase& otra) {
    // ... implementación ...
    return *this;
  }

  // Destructor
  ~MiClase() {}

private:
  int miembro1;
  std::string miembro2;
};
```

- Cualquier implementación de función colocada en un archivo de encabezado (excepto para plantillas de función) significa 0 para el ejercicio.

  Imagina un archivo de encabezado como un contrato: declara qué funciones, clases y variables están disponibles para usar, pero no proporciona los detalles de cómo funcionan. El archivo de implementación (generalmente con extensión .cpp) es donde se escribe el código real de esas funciones.

  Razones principales:

  - Doble inclusión: Si incluyes un archivo de encabezado en múltiples archivos de implementación, el compilador leerá la definición de la función varias veces, lo que puede generar errores de redefinición. Para evitar esto, se utilizan las guardas de inclusión. Sin embargo, poner la implementación en el encabezado aumenta la probabilidad de estos problemas.
  - Tiempo de compilación: Al incluir un archivo de encabezado, todo su contenido se copia en el archivo que lo incluye. Si una función está completamente definida en el encabezado, este se hará más grande y el tiempo de compilación aumentará, ya que el compilador tendrá que procesar más código.
  - Principio de separación de interfaces e implementación: Mantener la interfaz (declaración de la función) separada de la implementación promueve una mejor organización del código y facilita su mantenimiento. Si la interfaz cambia, solo es necesario modificar el archivo de encabezado, mientras que la implementación puede permanecer igual.

- Encapsulación: Al ocultar la implementación de una función en un archivo .cpp, se protege el código interno de la función y se evita que otras partes del programa dependan de detalles específicos de su implementación.

Excepción: Plantillas de función

Las plantillas de función son una excepción a esta regla porque su implementación depende de los tipos de datos con los que se utilicen. Al definir una plantilla de función en un archivo de encabezado, se permite al compilador generar código específico para cada instancia de la plantilla en el punto donde se utiliza.

En resumen:

La regla de no colocar la implementación de una función en un archivo de encabezado (excepto para plantillas) busca:

- Mejorar la eficiencia de la compilación.
- Facilitar el mantenimiento del código.
- Promover una mejor organización del código.
- Proteger la implementación de cambios externos.

Ejemplo:

```
// MiClase.h (archivo de encabezado)
#ifndef MICLASE_H
#define MICLASE_H

class MiClase {
public:
  void miFuncion(int x); // Declaración de la función
};

#endif

// MiClase.cpp (archivo de implementación)
#include "MiClase.h"

void MiClase::miFuncion(int x) {
  // Implementación de la función
  // ...
}
```

En este ejemplo, la función miFuncion se declara en el archivo de encabezado MiClase.h y se implementa en el archivo de implementación MiClase.cpp. Esto sigue las buenas prácticas de la programación en C++.

- Debes poder usar cada uno de tus encabezados independientemente de otros. Por lo tanto, deben incluir todas las dependencias que necesitan. Sin embargo, debes evitar el problema de la doble inclusión agregando protecciones de inclusión. De lo contrario, tu calificación será 0.

Al incluir un archivo de encabezado en otro, estás copiando todo su contenido en ese punto del código. Si incluyes el mismo archivo dos veces en el mismo archivo fuente, el compilador encontrará declaraciones duplicadas y generará un error. Esto se conoce como doble inclusión.

Para evitar este problema, se utilizan las protecciones de inclusión. Estas son macros que se definen al principio de un archivo de encabezado y que evitan que el contenido del archivo se incluya más de una vez en el mismo archivo fuente.

Ejemplo:

```
#ifndef MI_CLASE_H
#define MI_CLASE_H

// Declaración de la clase MiClase
class MiClase {
  // ...
};

#endif // MI_CLASE_H
```

Explicación:

- #ifndef MI_CLASE_H: Esta directiva verifica si la macro MI_CLASE_H no ha sido definida anteriormente. Si no ha sido definida, se ejecuta el código que sigue.
- #define MI_CLASE_H: Define la macro MI_CLASE_H.
- Contenido del archivo: Aquí va la declaración de la clase o cualquier otra declaración que necesites.
- #endif // MI_CLASE_H: Marca el final de la sección que se ejecutará si la macro no estaba definida.

¿Cómo funciona esto?

La primera vez que se incluye el archivo, la macro MI_CLASE_H no está definida, por lo que se ejecuta todo el código dentro de las directivas #ifndef y #endif. La segunda vez que se intenta incluir el mismo archivo, la macro ya está definida, por lo que se salta todo el contenido del archivo, evitando la doble inclusión.

¡Por Odín, por Thor! ¡Usa tu cerebro!!!

EL ESTANDAR C++98

El estándar C++98 representa una versión fundamental y ampliamente utilizada del lenguaje de programación C++. Publicado en 1998, este estándar estableció un conjunto de reglas y características que los compiladores de C++ debían cumplir para garantizar la portabilidad y compatibilidad del código.

¿Qué significa esto en la práctica?

Unificación: Antes de C++98, existían diferentes implementaciones de C++ con pequeñas variaciones, lo que dificultaba la portabilidad del código. El estándar C++98 ayudó a unificar estas implementaciones, asegurando que un programa escrito en C++98 se compilara y ejecutara de manera similar en diferentes compiladores y sistemas operativos.

Biblioteca Estándar (STL): Introdujo la Standard Template Library (STL), una colección de plantillas que proporcionan estructuras de datos (como vectores, listas, mapas) y algoritmos (como ordenar, buscar) listos para usar, lo que agiliza el desarrollo de software.

El estándar C++98 incluyó características clave como:

- Excepciones: Un mecanismo para manejar errores de forma estructurada.
- Espacios de nombres: Para organizar el código y evitar conflictos de nombres.
- Plantillas: Para crear código genérico que pueda trabajar con diferentes tipos de datos.
- Tipos booleanos: El tipo bool para representar valores verdaderos o falsos.

Si bien han surgido estándares posteriores (C++11, C++14, etc.) con nuevas características y mejoras, C++98 sigue siendo una base sólida sobre la que se construyen muchas aplicaciones.

La Biblioteca Estándar de C++98 (STL) es una colección de plantillas que proporcionan estructuras de datos y algoritmos predefinidos, lo que te permite realizar tareas comunes de programación de una manera eficiente y concisa.

¿Por qué utilizar la STL?

- Reutilización de código: Evita reinventar la rueda al proporcionar implementaciones robustas y optimizadas de estructuras de datos y algoritmos comunes.
- Aumento de la productividad: Reduce el tiempo de desarrollo al ofrecer herramientas listas para usar.
- Mejora de la calidad del código: El código escrito utilizando la STL suele ser más legible y mantenible.

Componentes principales de la STL

- Contenedores: Almacenan elementos de datos de forma organizada. Algunos ejemplos son:
  - vector: Un arreglo dinámico que puede crecer y encoger.
  - list: Una lista doblemente enlazada que permite inserciones y eliminaciones eficientes en cualquier posición.
  - map: Un contenedor asociativo que almacena pares clave-valor ordenados.
  - set: Un contenedor asociativo que almacena elementos únicos ordenados.
- Iteradores: Permiten acceder a los elementos de un contenedor de forma secuencial.

- Algoritmos: Realizan operaciones sobre los elementos de un contenedor, como ordenar, buscar, copiar y transformar.

LA PROGRAMACIÓN ORIENTA A OBJETOS

La Programación Orientada a Objetos (POO) es un paradigma de programación que nos permite modelar el mundo real en nuestro código de una manera más intuitiva y organizada. En vez de pensar en líneas de código como instrucciones secuenciales, la POO nos invita a pensar en términos de "objetos", que son como pequeñas representaciones de cosas del mundo real.

¿Qué es un objeto?

Imagina que estás creando un programa para gestionar una biblioteca. En lugar de pensar en libros como simples registros de datos (título, autor, ISBN), en la POO podemos crear un objeto llamado "Libro". Este objeto tendrá atributos (propiedades) como el título, el autor, el ISBN y métodos (acciones) como prestar, devolver u obtener información sobre el libro.

```
class Libro {
public:
  std::string titulo;
  std::string autor;
  std::string isbn;

  void prestar() {
    // Lógica para prestar el libro
  }

  void devolver() {
    // Lógica para devolver el libro
  }
};
```

Conceptos clave de la POO:

- Clases: Son como planos o moldes para crear objetos. La clase "Libro" define las características y comportamientos que todos los libros tendrán.
- Objetos: Son instancias de una clase. Cada libro que creamos en nuestro programa será un objeto de la clase "Libro".
- Atributos: Son las características o propiedades de un objeto. Por ejemplo, el título, el autor y el ISBN son atributos del objeto "Libro".
- Métodos: Son las acciones que un objeto puede realizar. Los métodos "prestar" y "devolver" son ejemplos de métodos de la clase "Libro".
- Encapsulación: Consiste en ocultar los detalles internos de un objeto y exponer solo una interfaz para interactuar con él. Esto hace que el código sea más fácil de mantener y reutilizar.
- Herencia: Permite crear nuevas clases (subclases) a partir de clases existentes (superclases), heredando sus atributos y métodos. Esto fomenta la reutilización de código y la creación de jerarquías de clases.

- Polimorfismo: Permite que objetos de diferentes clases puedan ser tratados como si fueran de la misma clase, siempre y cuando tengan métodos con el mismo nombre. Esto aumenta la flexibilidad y la extensibilidad del código.

¿Para qué se usa la POO?

- Organización del código: La POO permite organizar el código de forma más modular y jerárquica, lo que facilita su comprensión y mantenimiento.
- Reutilización de código: Al crear clases, podemos reutilizarlas en diferentes partes de nuestro programa, evitando la duplicación de código.
- Abstracción: La POO nos permite modelar el mundo real de forma más abstracta, centrándonos en los aspectos relevantes y ocultando los detalles internos.
- Flexibilidad: La POO facilita la creación de software más flexible y adaptable a los cambios.

Ejemplo más complejo:

Imagina que además de libros, queremos gestionar usuarios en nuestra biblioteca. Podemos crear una clase "Usuario" con atributos como nombre, apellido y un método para solicitar un préstamo.

```
class Usuario {
public:
  std::string nombre;
  std::string apellido;

  void solicitarPrestamo(Libro* libro) {
    // Lógica para solicitar un préstamo
  }
};
```

Ahora podemos crear objetos de tipo "Libro" y "Usuario" y hacer que interactúen entre sí.

En resumen:

La POO es una forma de programar que nos permite modelar el mundo real de una manera más natural y organizada. Al utilizar clases, objetos, atributos y métodos, podemos crear programas más estructurados, reutilizables y fáciles de mantener.