Cristina Gómez
19-10266
Lenguajes de Programación I

Examen 1

El Lenguaje escogido para el examen es C++.

Pregunta a

(a) Diga qué tipo de alcances y asociaciones posee, argumentando las ventajas y desventajas de la decisión tomada por los diseñadores del lenguaje, en el contexto de sus usuarios objetivos.

Alcances y Asociaciones en C++

- **Alcance (Scope)**

  Alcance local: El identificador es visible solo dentro del bloque (delimitado por llaves {}) donde está declarado, incluyendo bloques anidados.

  Alcance global: Las variables declaradas fuera de cualquier función son globales y están accesibles en todo el programa.
  C++ usa principalmente los alcances local y global, pero también existe:

  Alcance de clase: Los nombres de los miembros (datos o funciones) de una clase son visibles dentro de la definición de la propia clase, y para el resto del programa solo a través de un objeto, una referencia, o con el operador de resolución de alcance.

  Alcance de Función: Un tipo de alcance que se aplica a las variables que no están en un bloque específico, como las variables declaradas a nivel de función, pero no en un bloque.

- **Asociación (Binding)**

  Asociación temprana (Early Binding): Cuando compilas el programa, el compilador decide exactamente qué función ejecutar. Es rápido y predecible.

  Asociación tardía (Late Binding): Como decidir "sobre la marcha". El programa decide en tiempo de ejecución qué función llamar, dependiendo del objeto real. Es flexible pero un poco más lento.

Ventajas y Desventajas

Los diseñadores de C++, crearon estas reglas pensando en su público objetivo: programadores de sistemas y aplicaciones que necesitan un control preciso sobre los recursos, un alto rendimiento y la capacidad de gestionar proyectos de gran tamaño.

- **Ventajas**

  Modularidad y Encapsulación: El sistema de alcance y asociación es la base del modularidad. Los espacios de nombres y las clases permiten agrupar entidades relacionadas y evitar colisiones de nombres, algo fundamental en proyectos grandes donde múltiples equipos desarrollan bibliotecas diferentes que podrían usar los mismos nombres para sus funciones o variables.

  Prevención de Errores: El alcance local es un pilar de la programación estructurada. Al limitar la vida y visibilidad de las variables a donde son estrictamente necesarias, se reduce la probabilidad de modificaciones accidentales y se facilita el razonamiento sobre el comportamiento del código.

Rendimiento máximo: La asociación temprana y los alcances estrictos permiten que el compilador optimice el código al máximo. C++ fue creado para sistemas donde cada nanosegundo cuenta.

Control absoluto: El programador decide exactamente qué es global, qué es local y cuándo usar polimorfismo.

- **Desventajas**

Complejidad y Curva de Aprendizaje: Para los programadores principiantes, la interacción entre static, extern, const, los espacios de nombres y las diferentes reglas de alcance puede ser confusa. Errores comunes, como definir una variable global en un fichero de cabecera (.h) sin inline o const, pueden llevar a errores de enlazado difíciles de diagnosticar.

Peligro de las Variables Globales: Aunque C++ ofrece herramientas para controlar su visibilidad, permite y facilita el uso de variables globales con asociación externa. El uso indiscriminado de estas variables es una fuente conocida de errores difíciles de depurar, ya que pueden ser modificadas desde cualquier parte del programa, rompiendo la encapsulación y haciendo que el estado del sistema sea impredecible.

Gestión manual de memoria: el programador es el responsable de crear y destruir objetos. Es fácil olvidar liberar memoria, lo que implica fugas de memoria.

(b) Diga qué tipo de módulos ofrece (de tenerlos) y las diferentes formas de importar y exportar nombres.

En C++, no existen "tipos" de módulos en el sentido de categorías diferentes, sino más bien diferentes tipos de ficheros o unidades que componen el sistema de módulos:

Unidad de Interfaz de Módulo: Es el fichero principal que define lo que un módulo exporta. Se reconoce por la declaración export module <nombre_modulo>; al principio. Contiene las declaraciones de funciones, clases, plantillas, etc., que serán visibles para otros módulos que lo importen.

Unidad de Implementación de Módulo: Contiene la implementación de las declaraciones exportadas en la unidad de interfaz, así como código privado del módulo. Se declara con module <nombre_modulo>;. El código aquí no es visible para los importadores, lo que permite ocultar por completo los detalles de implementación.

Particiones de Módulo: Permiten dividir un módulo grande en varias partes más manejables para mejorar la organización del código.

Formas de Exportar Nombres

- **Exportar Declaraciones Individuales:** Puedes exportar funciones, variables, clases o cualquier otra declaración anteponiendo export.
- **Exportar un Bloque de Declaraciones:** Puedes agrupar varias declaraciones en un bloque export { ... }.
- **Re-exportar Nombres de Otro Módulo:** Un módulo puede exportar nombres que ha importado de otro módulo.

Formas de Importar Nombres

- **Importar un Módulo Completo:** Esta es la forma más común. Importa todos los nombres exportados por el módulo especificado, haciéndolos disponibles en el fichero actual.
- **Importar Particiones de Módulo:** Aunque se trabaja con particiones al escribir un módulo, al consumirlo, generalmente solo se importa el módulo principal. El sistema se encarga de ensamblar las partes. No es común importar una partición directamente desde fuera del módulo.

(c) Diga si el lenguaje ofrece la posibilidad de crear aliases, sobrecarga y polimorfismo. En caso afirmativo, dé algunos ejemplos.

Sí, C++ ofrece soporte completo para la creación de aliases, la sobrecarga de funciones y operadores, y el polimorfismo.

Aliases: C++ ofrece dos formas de hacerlo: typedef y using (introducida en C++11).

```cpp
#include <iostream>

// Usando typedef para crear el alias.
// Le decimos al compilador: "el tipo 'int' ahora también se puede llamar 'Edad'".
typedef int Edad;

// La función usa el alias para mayor claridad.
void imprimirEdad(Edad anios) {
    std::cout << "Esta persona tiene " << anios << " años." << std::endl;
}

int main() {
    // Declaramos la variable usando el alias 'Edad'.
    Edad edadDeJuan = 30;

    imprimirEdad(edadDeJuan); // Salida: Esta persona tiene 30 años.

    return 0;
}
```

Sobrecarga: La sobrecarga es la capacidad de definir múltiples funciones, constructores u operadores con el mismo nombre, pero con diferentes parámetros en el mismo ámbito.

```cpp
#include <iostream>

// Función para sumar dos enteros
int add(int a, int b) {
    return a + b;
}

// Sobrecarga para sumar dos números decimales
double add(double a, double b) {
    return a + b;
}

int main() {
    std::cout << "Suma de enteros: " << add(5, 10) << std::endl;      // Llama a la versión int
    std::cout << "Suma de doubles: " << add(3.5, 2.7) << std::endl; // Llama a la versión double
    return 0;
}
```

Polimorfismo: Permite que objetos de diferentes clases respondan al mismo mensaje de maneras distintas. En C++, esto se logra principalmente a través de funciones virtuales y herencia.

```cpp
#include <iostream>

// clase base
class Personaje {
public:
    // Hacemos que la función 'atacar' sea virtual.
    // Esto permite que cada tipo de personaje la implemente a su manera.
    virtual void atacar() {
        std::cout << "El personaje ataca..." << std::endl;
    }
    // destructor virtual
    virtual ~Personaje() {}
};

// clases derivadas - los personajes específicos
class Guerrero : public Personaje {
public:
    //guerrero redefine el ataque.
    void atacar() override {
        std::cout << "El guerrero ataca" << std::endl;
    }
};

class Mago : public Personaje {
public:
    //mago también redefine el ataque.
    void atacar() override {
        std::cout << "El mago ataca" << std::endl;
    }
};

// Esta función solo sabe que va a recibir un 'Personaje'.
// No le importa si es un Guerrero, un Mago o cualquier otro.
void ejecutarTurno(Personaje& jugador) {
    std::cout << "Es el turno del jugador:" << std::endl;
    jugador.atacar(); // Aquí ocurre el polimorfismo
}

int main() {
    Guerrero Kratos;
    Mago Gandalf;

    // La misma función 'ejecutarTurno' se comporta distinto
    // dependiendo del objeto que le pases.
    ejecutarTurno(Kratos);
    ejecutarTurno(Gandalf);

    return 0;
}
```

(d) Diga qué herramientas ofrece a potenciales desarrolladores, como: compiladores, intérpretes, debuggers, profilers, frameworks, etc.

- **Compiladores**

  GCC (GNU Compiler Collection)
  Clang
  MSVC (Microsoft Visual C++)
  Intel C++ Compiler (ICL)

- **Intérpretes**
  C++ no es un lenguaje interpretado. Su diseño se centra en la compilación anticipada para lograr el máximo rendimiento posible.

- **Debuggers**

  GDB (GNU Debugger)
  LLDB
  Visual Studio Debugger

- **Profilers**

    gprof (GNU Profiler)
    Perf
    Visual Studio Profiling Tools
    Intel VTune Profiler

- **Frameworks**

    Qt
    Unreal Engine
    Crow y Pistache
    JUCE

Las siguientes preguntas se encuentran en el repositorio de github: https://github.com/CristyGomez/CI3641-CristinaGomez

El repositorio contiene una carpeta por cada pregunta.

La pregunta 2 son 4 archivos .txt.

Las preguntas que contienen código, contienen un Makefile para ser ejecutadas

    En el caso de la pregunta 3 y la Pregunta 5 existe: “make main” para ejecutar el programa y “make test” para ejecutar las pruebas unitarias de cada programa.

    En el caso de la pregunta 4 solo se necesita hacer “make”.

    También en los tres casos existe “make limpiar” para eliminar los archivos creados.