Identify real type

Descripción del ejercicio:

1. **Implementación de clases:**
   a. Crear una clase base llamada Base que contenga únicamente un destructor virtual público.
   b. Crear tres clases vacías denominadas A, B y C que hereden públicamente de Base.

   Nota: No es necesario que estas cuatro clases sigan la Forma Canónica Ortodoxa.

2. **Funciones a implementar:**
   a. Base *generate(void);
      Esta función instancia aleatoriamente objetos de las clases A, B o C y devuelve el objeto creado como un puntero a Base. Puedes utilizar cualquier método para implementar la elección aleatoria.
   b. void identify(Base* p);
      Esta función imprime el tipo real del objeto al que apunta p, mostrando "A", "B" o "C".
   c. void identify(Base& p);
      Esta función imprime el tipo real del objeto referenciado por p, mostrando "A", "B" o "C". Está prohibido utilizar punteros dentro de esta función.

   Nota: Está prohibido incluir el encabezado typeinfo.

3. **Programa de prueba:**
   Escribir un programa que verifique que todo funciona según lo esperado.

Explicación detallada de cómo abordar el ejercicio en C++98:

1. **Creación de la clase base y las clases derivadas:**
   a. **Clase Base:** Definir una clase Base con un destructor virtual público. Esto asegura que, al eliminar un objeto a través de un puntero a Base, se llame al destructor adecuado de la clase derivada, evitando posibles fugas de memoria.
   b. **Clases A, B y C:** Definir tres clases vacías que hereden públicamente de Base. Aunque están vacías, la herencia pública permite que un puntero de tipo Base* apunte a objetos de estas clases.
2. **Implementación de la función generate:**
   Esta función debe crear dinámicamente una instancia de A, B o C de manera aleatoria y devolver un puntero a Base. Para lograr la aleatoriedad en C++98, se puede utilizar la función rand() de la biblioteca <cstdlib>, junto con srand() para sembrar el generador de números aleatorios. Dependiendo del valor generado, se instancia una de las tres clases y se devuelve como un puntero a Base.
3. **Implementación de la función identify con puntero (Base* p):**
   Dado que no se permite el uso de std::typeinfo (que proporciona typeid para identificar el tipo en tiempo de ejecución), se puede emplear el *downcasting* seguro utilizando dynamic_cast. Este operador intenta convertir un puntero de la clase base a un puntero

de una clase derivada. Si la conversión es exitosa, el puntero resultante no será nulo, lo que indica que el objeto es de ese tipo específico. Se verifica secuencialmente contra cada clase derivada (A, B, C) y se imprime el tipo correspondiente.

4. **Implementación de la función identify con referencia (Base& p):**
   En esta función, al no poder utilizar punteros, se puede emplear el mismo enfoque de *downcasting* con dynamic_cast, pero esta vez con referencias. Si la conversión falla, dynamic_cast lanzará una excepción de tipo std::bad_cast. Utilizando bloques try-catch, se intenta convertir la referencia de Base a cada una de las clases derivadas. Al capturar la excepción, se determina si la conversión fue exitosa o no, y se imprime el tipo correspondiente.
5. **Programa de prueba:**
   Se debe escribir un programa que:
   i. Llame a la función generate para obtener un puntero a Base que apunta a una instancia de A, B o C.
   ii. Pase este puntero a la función identify(Base* p) y verifique que se imprime el tipo correcto.
   iii. Pase la referencia del objeto a la función identify(Base& p) y verifique nuevamente la salida.

Este programa ayudará a confirmar que las funciones identify pueden determinar correctamente el tipo real del objeto en tiempo de ejecución.

Recursos recomendados en español:

Para abordar con éxito este ejercicio, es fundamental comprender los conceptos de herencia, polimorfismo, *dynamic_cast* y manejo de excepciones en C++. A continuación, se presentan cinco videos en español que abordan estos temas:

1. **"Tutorial C++ - Herencia y Polimorfismo"**

   Este video ofrece una explicación detallada sobre la herencia y el polimorfismo en C++, conceptos fundamentales para la creación de clases base y derivadas, así como para la implementación de funciones virtuales.

   Ver video

2. **"181.- Curso de C++ Avanzado. Uso de dynamic_cast."**

   En este tutorial se aborda el uso de dynamic_cast en C++, una herramienta esencial para realizar conversiones seguras entre tipos en una jerarquía de herencia, lo cual es crucial para identificar el tipo real de un objeto en tiempo de ejecución.

   Ver video

3. **"Manejo de excepciones en C++"**

   Este video profundiza en el manejo de excepciones en C++, incluyendo la sintaxis y las mejores prácticas para utilizar try, catch y throw, elementos necesarios para gestionar errores y excepciones en tus programas.

Ver video

4. **"Curso de C++ Avanzado. Uso de excepciones dentro de funciones."**

   En este video se explora cómo las excepciones se propagan desde funciones y cómo se maneja el "desenredado de pila" en C++, proporcionando una comprensión más profunda del control de flujo en presencia de errores.

   Ver video

5. **"Tutorial de C++ - Excepciones"**

   Este tutorial ofrece una introducción al manejo de excepciones en C++, explicando cómo y cuándo utilizarlas para mejorar la robustez y fiabilidad de tus programas.

   Ver video

Estos recursos te proporcionarán una base sólida para comprender los conceptos de herencia, polimorfismo, conversión de tipos con dynamic_cast y manejo de excepciones en C++, facilitando así la implementación exitosa del ejercicio propuesto.

El código

Este código implementa un sistema de identificación de clases derivadas de una clase base utilizando *downcasting* con dynamic_cast, sin emplear std::typeinfo. Veamos qué hace cada parte y cómo funciona.

1 Definición de Clases (Base.hpp, A.hpp, B.hpp, C.hpp)

Base.hpp

```
#pragma once

class Base {
	public:
		virtual ~Base();  // Destructor virtual
};
```

- Base es una clase base abstracta con un **destructor virtual**.
- El destructor virtual es importante porque permite que los objetos derivados sean correctamente destruidos cuando se eliminan a través de un puntero a Base.
- No tiene atributos ni métodos adicionales.

A.hpp, B.hpp, C.hpp

```
#pragma once
#include "Base.hpp"

class A : public Base {};  // A hereda de Base
```

```
#pragma once
#include "Base.hpp"

class B : public Base {};  // B hereda de Base
```

```
 #pragma once
#include "Base.hpp"

class C : public Base {};  // C hereda de Base
```

- Estas tres clases (A, B, C) **heredan públicamente** de Base y están vacías.
- No contienen atributos ni métodos, solo existen para diferenciarse en tiempo de ejecución.

2 Implementación de funciones (Functions.cpp)

◇ Función generate()

```
Base *generate() {
	srand(time(NULL));			// Inicializa la semilla de números aleatorios
	char s = "ABC"[rand() % 3];		// Genera un carácter aleatorio 'A', 'B' o 'C'
	cout << "Type: " << s << ", aleatorily generated 🎰 " << endl;

	switch (s) {
		case 'A': return new A;		// Devuelve una nueva instancia de A
		case 'B': return new B;		// Devuelve una nueva instancia de B
		case 'C': return new C;		// Devuelve una nueva instancia de C
	}
	return NULL; // Si por alguna razón no coincide con 'A', 'B' o 'C' (no debería ocurrir)
}
```

📌 ¿Qué hace esta función?

- Usa rand() para seleccionar aleatoriamente una de las tres clases (A, B o C).
- Crea dinámicamente un objeto de la clase seleccionada y devuelve un **puntero a Base**.
- Esto permite tratar la instancia de A, B o C como un Base*.

◇ Función identify(Base* p)

```
void identify(Base* p) {
	bool casted;

	cout << "Identify with Base: *";

	casted = dynamic_cast<A *>(p);	// Intenta hacer un cast de p a A*
	if (casted) {
		cout << "A" << endl;		// Si el cast es exitoso, el objeto es de tipo A
		return;
	}
	casted = dynamic_cast<B *>(p);	// Intenta hacer un cast de p a B*
	if (casted) {
		cout << "B" << endl;		// Si el cast es exitoso, el objeto es de tipo B
		return;
	}
	casted = dynamic_cast<C *>(p);	// Intenta hacer un cast de p a C*
	if (casted) {
		cout << "C" << endl;		// Si el cast es exitoso, el objeto es de tipo C
		return;
	}
	cout << "Unknown" << endl;		// Si ningún cast es exitoso, el tipo es desconocido
}
```

📌 ¿Cómo funciona?

1. **Aplica dynamic_cast<A *>**: Si p es realmente un A*, la conversión será exitosa y casted será true, por lo que se imprime "A".
2. **Repite para B* y C***.
3. Si p no es de ninguno de esos tipos (lo cual no debería ocurrir), se imprime "Unknown".

🛠 Nota sobre dynamic_cast con punteros:

Si el dynamic_cast falla, devuelve NULL, lo que facilita la verificación.

◇ Función identify(Base& p)

```
void identify(Base& p) {
	cout << "Identify with Base: &";

	try {
		A &a = dynamic_cast<A &>(p);	// Intenta hacer un cast de p a A&
		(void)a;			// Evita advertencia de variable no utilizada
		cout << "A" << endl;		// Si el cast es exitoso, imprime A
```

```
	} catch (const std::exception &e) {
		try {
			B &b = dynamic_cast<B &>(p);	// Intenta hacer un cast de p a B&
			(void)b;
			cout << "B" << endl;
		} catch (const std::exception &e) {
			try {
				C &c = dynamic_cast<C &>(p);	// Intenta hacer un cast de p a C&
				(void)c;
				cout << "C" << endl;
			} catch (const std::exception &e) {
				cout << "Unknown" << endl;			// Si todo falla, imprime
Unknown
			}
		}
	}
}
```

📌 **¿Cómo funciona?**

1. **dynamic_cast<A&>(p)**: Si p es un A, la conversión es exitosa y se imprime "A".
2. Si falla, se lanza una excepción (std::bad_cast), que se captura en el catch y se intenta convertir a B&.
3. Si también falla, se intenta C&.
4. Si todas las conversiones fallan, se imprime "Unknown".

🛠 **Nota sobre dynamic_cast con referencias:**

Si el dynamic_cast falla con referencias, lanza una excepción (std::bad_cast), lo que obliga a usar try-catch.

3 Programa principal (main.cpp)

```
int	main() {
	Base	*ptr;	// Puntero a Base

	cout << "\n* * * Executing Test * * *\n" << endl;

	ptr = generate();	// Genera una instancia aleatoria

	cout << "\n";
	identify(ptr);		// Identifica el tipo usando un puntero
	identify(*ptr);		// Identifica el tipo usando una referencia

	delete ptr;			// Libera memoria
```

```
        cout << "\n* * *  Ending  Test  * * *\n" << endl;

        return 0;
}
```

📌 **¿Qué hace main()?**

1. Llama a generate() para obtener un Base* a una instancia de A, B o C.
2. Usa identify(ptr) para determinar el tipo mediante un puntero.
3. Usa identify(*ptr) para determinar el tipo mediante una referencia.
4. Libera la memoria con delete ptr.

📌 Resumen general

✅ **¿Qué hace este código?**

- **Crea un sistema de identificación de clases derivadas en tiempo de ejecución sin usar std::typeinfo**.
- **Genera instancias aleatorias de A, B o C**.
- **Usa dynamic_cast para identificar el tipo del objeto** con dos enfoques:
  - **Con punteros (Base*)**, devolviendo NULL en caso de fallo.
  - **Con referencias (Base&)**, capturando excepciones en caso de fallo.
- **Ejecuta una prueba automática en main()**.

🔹 **Conceptos clave utilizados en el código:**

- Herencia en C++
- Uso de dynamic_cast con punteros y referencias
- Manejo de excepciones con try-catch
- Generación de números aleatorios (rand() y srand())