Laboratorio 20

Punteros inteligentes

1. Competencias

1.1. Competencias del curso

Conoce, comprende e implementa programas usando punteros inteligentes del lenguaje de programación C++.

1.2. Competencia del laboratorio

Conoce, comprende e implementa programas usando punteros inteligentes del lenguaje de programación C++.

2. Equipos y Materiales

- Un computador.
- IDE para C++.
- Compilador para C++.

3. Marco Teórico

3.1. Punteros inteligentes

Los punteros inteligentes (smart pointers) son un esfuerzo del C++ moderno, de la versión 11 en adelante, que intenta resolver el problema de la gestión de memoria.

El problema es que para alojar memoria en el heap debemos utilizar la palabra new y eliminar explícitamente las variables con el operador delete.

Los punteros inteligentes permiten ser eliminados de la memoria automáticamente cuando el nombre de la variable sale de su ámbito. La biblioteca <memory> ofrece distintos tipos, lo más comunes son:

- Punteros únicos std::unique_ptr
- Punteros compartidos std::shared_ptr
- Punteros débiles std::weak_ptr

3.1.1. Punteros únicos

Estos punteros tienen dos funcionalidades esenciales:

- Solo puede haber un puntero único en una dirección de memoria al mismo tiempo.
- La memoria se libera automáticamente cuando el puntero abandona su ámbito.

```
#include <iostream>
#include <memory>

class Punto
{
public:
	Punto(double x, double y) : x(x), y(y) {};

	void print()
	{
		std::cout << "(" << x << ", " << y << ")\n";
	}

private:
	double x{}, y{};
};

int main()
{
	std::unique_ptr<int> num{ new int(100) };
	std::unique_ptr<std::string> cad{ new std::string("Hola") };
	std::unique_ptr<Punto> punto{ new Punto(3, 7) };

	std::cout << *num << ", " << *cad << "\n";
	std::cout << num.get() << ", " << cad.get() << "\n";
	punto->print();
	system("pause");
	return 0;
}
```

Los punteros únicos cuentan con la función std::make_unique que se puede utilizar en lugar del operadores new y que ofrece algunas ventajas como la seguridad de que no

obtener nunca una pérdida de memoria cuando, en comparación, new si puede potencialmente fallar si el sistema no es capaz de reservar la memoria necesaria correctamente.

Se recomienda utilizar esta forma siempre excepto si se necesita un destructor personalizado o se está adoptando un puntero sin formato:

```
std::unique_ptr<int> num = std::make_unique<int>(100);
std::unique_ptr<std::string> cad = std::make_unique<std::string>("Hola");
std::unique_ptr<Punto> punto = std::make_unique<Punto>(3, 7);
```

Tal como hemos dicho un puntero único no se puede copiar, así previene que múltiples punteros apunten a la misma dirección de memoria:

```
std::unique_ptr<int> num1 = std::make_unique<int>(100);
std::unique_ptr<int> num2 = std::1; //Error
```

Lo que sí permiten es transferir la propiedad mediante std::move, de manera que el puntero original pierde el acceso y lo transfiere a otro:

```
std::unique_ptr<int> num1 = std::make_unique<int>(100);
std::cout << num1 << std::endl; // 0x257a14f12b0

// transferimos la propiedad
std::unique_ptr<int> num2 = std::move(num1);

std::cout << num1 << std::endl; // 0
std::cout << num2 << std::endl; // 0x257a14f12b0
```

También permiten reiniciarlos y establecerlos a un puntero nullptr:

```
std::unique_ptr<int> num1 = std::make_unique<int>(100);
std::cout << num1 << std::endl; // 0x25eb39f12b0
num1.reset();                   // lo reiniciamos a nullptr
std::cout << num1 << std::endl; // 0
```

Es posible crear un arreglo de objetos dinámicos con std::make_unique, sin embargo no podemos inicializarlos directamente y esto solo funcionará si tenemos un constructor por defecto:

```
auto arr_ptr = std::make_unique<Punto[]>(3);
for (size_t i{ 0 }; i < 3; i++)
{
	arr_ptr[i].print();
}
```

(0, 0)

(0, 0)

(0, 0)

Para inicializar los objetos en la definición deberemos usar un std::unique_ptr y crear el array con new, perdiendo algunas de las capacidades que nos ofrece esta funcionalidad:

```
auto arr_ptr = std::unique_ptr<Punto[]>(new Punto[3]{ Punto(1, 2), Punto(3,
4), Punto(5, 6) });

for (size_t i{ 0 }; i < 3; i++)
{
	arr_ptr[i].print();
}
```

(1, 2)

(3, 4)

(5, 6)

3.1.2. Punteros compartidos

Este tipo de puntero inteligente permite que múltiples punteros apunten a la misma dirección de memoria, es decir, podemos tener copias de la dirección. El sistema detectará automáticamente cuando no quede ninguna copia viva en su ámbito y entonces liberará automáticamente el espacio en la memoria.

Podemos crear estos punteros con new o std::make_shared, generalmente se prefiere la segunda opción por aportar más seguridad pero resta control del puntero:

```
// Ejemplos de punteros inteligentes
std::shared_ptr<int> num1 = std::make_shared<int>(100);
std::shared_ptr<int> num2 = num1;
// Son el mismo puntero
std::cout << num1 << ", " << num2 << "\n";
// Cantidad de veces que el puntero está en uso
std::cout << num1.use_count() << "\n";  // 2
// Liberamos la copia
num2.reset()
// Cantidad de veces que el puntero está en uso
std::cout << num1.use_count() << "\n";  // 1
```

Podemos transformar un puntero único a un puntero compartido mediante std::move, pero no a la inversa:

```
// De unique a shared se puede
std::unique_ptr<int> num1 = std::make_unique<int>(100);
std::shared_ptr<int> num2 = std::move(num1);
std::cout << num1 << ", " << num2 << "\n"; // 0, 0x1cd3b9912b0

// De shared a unique no se puede
std::shared_ptr<int> num1 = std::make_shared<int>(100);
std::unique_ptr<int> num2 = std::move(num1);  // Error
```

La sintaxis std::make_shared no está soportada para arreglos, desde C++17 la forma recomendada de manejar punteros compartidos es esta:

```
// Array de enteros y acceso secuencial
std::shared_ptr<int[]> numeros(new int[4]{ 11, 22, 33, 44 });

for (size_t i{}; i < 4; i++)
{
	numeros[i] *= 2;
	std::cout << numeros[i] << " "; // 22 44 66 88
}

// Array de objetos y acceso secuencial
std::shared_ptr<Punto[]> puntos(new Punto[4]{
	Punto(1, 2), Punto(2, 3), Punto(3, 4), Punto(4, 5) });

for (size_t i{}; i < 4; i++)
	puntos[i].print();
```

(1, 2)

(2, 3)

(3, 4)

(4, 5)

3.1.3. Punteros débiles

Finalmente, los punteros débiles contienen una referencia sin posesión (débil) a un objeto que es gestionado por std::shared_ptr. Debe convertirse a std::shared_ptr para poder acceder al objeto referenciado. No implementan operadores flecha -> ni asterisco *, por lo que no se pueden utilizar directamente para leer ni modificar los datos:

```
std::shared_ptr<int> num1 = std::make_shared<int>(100);
// Creamos un puntero débil a partir de un puntero compartido
std::weak_ptr<int> num2 = num1;

// Este puntero no permite acceso ni modificación
std::cout << num2 << std::endl; // Error

// Necesitamos transformarlo a un shared_ptr bloqueándolo
std::shared_ptr<int> num3 = num2.lock();

// Podemos comprobar que son una copia
std::cout << num1 << ", " << num3 << "\n";
// 0x22f5e6c12c0, 0x22f5e6c12c0
```

¿Qué utilidad puede tener un puntero compartido al que no se puede acceder?

Los punteros débiles std::weak_ptr son una buena forma de resolver el problema de los punteros colgantes. Usando punteros crudos (los clásicos) es imposible saber si los datos a los que se hace referencia han sido desasignados o no. En cambio, al permitir que un std::shared_ptr administre los datos y al proporcionar std::weak_ptr a los usuarios de los datos, los usuarios pueden verificar la validez de los datos llamando a los métodos expired() y lock().

Esto es algo que no podría hacerse únicamente con std::shared_ptr, ya que todas las instancias de std::shared_ptr comparten la propiedad de los datos que no se eliminan antes de que se eliminen todas las instancias del puntero compartido.

En la práctica es la forma de resolver el problema de la dependencia cíclica. Para ilustrarlo supongamos esta clase:

```
#include <iostream>
#include <memory>

class Persona
{
public:
	Persona() = default;
	Persona(std::string nombre) : nombre(nombre) {};
	~Persona()
	{
		std::cout << nombre << " liberado de la memoria\n";
	}

	void set_amigo(std::shared_ptr<Persona> p)
	{
		this->amigo = p;
	}

	void print_amigo()
	{
		std::cout << "El amigo de " << nombre << " es "
			<< amigo->nombre << std::endl;
	}

private:
	std::string nombre{ "Sin nombre" };
	std::shared_ptr<Persona> amigo; // nullptr compartido
};
```

Una persona contiene un puntero llamado amigo que permite establecer otra instancia de Persona.

El problema inherente de este código es que, en caso de tener dos personas A y B, que tienen como amigos B y A respectivamente, se generará una dependencia cíclica implicando que ambas instancias nunca se liberarán de la memoria, pues siempre una de ella apuntará a la otra y el contador interno del puntero compartido será mayor que cero impidiendo la liberación:

```
int main()
{
	// Dependencia cíclica
	std::shared_ptr<Persona> A = std::make_shared<Persona>("Gustavo");
	std::shared_ptr<Persona> B = std::make_shared<Persona>("Fernando");

	A->set_amigo(B);
	B->set_amigo(A);
```

```
        A->print_amigo(); //
        B->print_amigo(); //

        return 0;
    }
```

El resultado es el siguiente:

```
El amigo de Gustavo es Fernando
El amigo de Fernando es Gustavo
```

¿Y los destructores?

Si en lugar de un puntero compartido, establecemos un puntero débil para el amigo, podemos evitar el problema de la dependencia cíclica, pues el contador compartido ahora se decrementará correctamente:

```
class Persona
{
public:
        // ...

        void set_amigo(std::shared_ptr<Persona> p)
        {
                // Conversión de shared a weak implícita
                this->amigo = p;
        }

        void print_amigo()
        {
                // Para utilizar el puntero débil lo bloqueamos
                std::shared_ptr<Persona> p_amigo = amigo.lock();
                std::cout << "El amigo de " << nombre << " es "
                        << p_amigo->nombre << std::endl;
        }

private:
        std::string nombre{ "Sin nombre" };
        std::weak_ptr<Persona> amigo; // nullptr débil
};
```

El resultado sería el siguiente:

El amigo de Gustavo es Fernando

El amigo de Fernando es Gustavo

Fernando liberado de la memoria

Gustavo liberado de la memoria

4. Ejercicios

Resolver los siguientes ejercicios planteados:

Implemente el siguiente código que usa punteros sin procesar:

1. Implemente el siguiente código que usa punteros sin procesar y explique lo que hace:

```
{
        double* d = new double(1.0);
        Point* pt = new Point(1.0, 2.0);

        *d = 2.0;
        (*pt).X(3.0);
        (*pt).Y(3.0);

        pt->X(3.0);
        pt->Y(3.0);

        delete d;
        delete pt;
}
```

2. Transfiera el código anterior reemplazando los punteros sin formato por std::unique_ptr.

3. Implementar el código para las clases C1 y C2, cada una de las cuales contiene el objeto compartido d anterior, por ejemplo:

```
class C1
        {
        private:
                std::shared_ptr<double> d;
        public:
                C1(std::shared_ptr<double> value) : d(value) {}
                virtual ~C1() { cout << "\nC1 destructor\n"; }
                void print() const { cout << "Valor " << *d; }
        };
```

4. Transfiera el código anterior reemplazando los punteros sin formato por std::shared_ptr<Point> p;
5. Al anterior código implemente un puntero débil a un puntero el cual no puede estar vacío.

5. Entregables

Al final estudiante deberá:

1. Compactar el código elaborado y subirlo al aula virtual de trabajo. Agregue sus datos personales como comentario en cada archivo de código elaborado.
2. Elaborar un documento que incluya tanto el código como capturas de pantalla de la ejecución del programa. Este documento debe de estar en formato PDF.
3. El nombre del archivo (comprimido como el documento PDF), será su LAB20_GRUPO_A/B/C_CUI_1erNOMBRE_1erAPELLIDO.

   (Ejemplo: LAB20_GRUPO_A _2022123_PEDRO_VASQUEZ).
4. Debe remitir el documento ejecutable con el siguiente formato:

   LAB20_GRUPO_A/B/C_CUI_ EJECUTABLE_1erNOMBRE_1erAPELLIDO

   (Ejemplo: LAB20_GRUPO_A_EJECUTABLE_2022123_PEDRO_VASQUEZ).

En caso de encontrarse trabajos similares, los alumnos involucrados no tendrán evaluación y serán sujetos a sanción.