Serialization

Descripción del ejercicio: Se solicita implementar una clase llamada Serializer que no pueda ser inicializada por el usuario de ninguna manera y que contenga los siguientes métodos estáticos:

- uintptr_t serialize(Data* ptr);

Este método toma un puntero y lo convierte al tipo entero sin signo uintptr_t.

- Data* deserialize(uintptr_t raw);

Este método toma un parámetro entero sin signo y lo convierte en un puntero a Data.

Además, se debe escribir un programa para probar que la clase Serializer funciona como se espera. Para ello:

1. Crear una estructura Data que no esté vacía, es decir, que contenga miembros de datos.
2. Utilizar serialize() en la dirección del objeto Data y pasar su valor de retorno a deserialize().
3. Asegurarse de que el valor devuelto por deserialize() sea igual al puntero original.

No olvidar entregar los archivos correspondientes a la estructura Data.

Explicación del ejercicio:

El objetivo principal de este ejercicio es comprender y aplicar los conceptos de serialización y deserialización de punteros en C++. **La serialización es el proceso de convertir una estructura de datos o un objeto en una secuencia de bits que puede ser almacenada o transmitida y posteriormente reconstruida**. En este caso, se trata de convertir un puntero a un tipo entero que pueda ser manejado de forma segura y luego reconvertido al puntero original.

Pasos para abordar el ejercicio en C++98:

1. **Definir la estructura Data:** Crear una estructura llamada Data que contenga al menos un miembro de datos. Por ejemplo:

```
struct Data {
  int value;
  // Otros miembros si es necesario
};
```

2. **Implementar la clase Serializer:** Esta clase debe contener dos métodos estáticos: serialize y deserialize. Además, para evitar que el usuario pueda crear instancias de Serializer, se puede declarar el constructor como privado o eliminarlo.
    a. **Método serialize:** Este método toma un puntero a Data y lo convierte a un entero sin signo de tipo uintptr_t. En C++, los punteros pueden ser reinterpretados como enteros para este propósito.
    b. **Método deserialize:** Este método toma un entero de tipo uintptr_t y lo convierte de vuelta a un puntero a Data.
3. **Probar la funcionalidad:** En la función main, crear una instancia de Data, inicializar sus miembros, y luego:
    a. Usar el método serialize para obtener la representación entera de la dirección del objeto Data.
    b. Pasar este valor al método deserialize para obtener de nuevo el puntero original.
    c. Verificar que el puntero obtenido es igual al puntero original y que los datos contenidos son los mismos.

Recursos recomendados:

A continuación, te proporciono una selección actualizada de recursos en español que te ayudarán a comprender y abordar con éxito el ejercicio propuesto:

1. **"Estructura de datos en C++: Creación de archivos - Serialización"**

    Este video explica cómo crear archivos serializados en C++, lo cual es fundamental para entender la serialización de estructuras de datos.

    Ver video

2. **"Todos sobre los Punteros en C++ – ¿Qué son, cómo hacerlos y usarlos"**

    Este tutorial ofrece una visión completa sobre los punteros en C++, incluyendo su declaración, uso y manipulación, aspectos clave para el ejercicio.

    Ver video

3. **"Curso C++. Punteros II. Sintaxis y uso. Vídeo 43"**

    En este video se profundiza en la sintaxis y el uso de los punteros en C++, proporcionando ejemplos prácticos que facilitan su comprensión.

    Ver video

4. **"Understanding the uintptr_t Data Type in C++"**

Aunque este video está en inglés, ofrece una explicación detallada sobre el tipo de dato uintptr_t en C++, esencial para la serialización de punteros.

Ver video

5. **"Curso Completo de C++ para Principiantes (2023)"**

Este curso completo abarca desde los conceptos básicos hasta temas más avanzados de C++, incluyendo la manipulación de punteros y estructuras de datos.

Ver video

Estos recursos te proporcionarán una base sólida para comprender los conceptos necesarios y abordar con éxito el ejercicio de serialización en C++.

El código

Este código implementa un mecanismo de **serialización y deserialización** de una estructura Data en C++, usando reinterpret_cast para convertir un puntero a un número entero (uintptr_t) y viceversa.

A continuación, explico **paso a paso** el propósito de cada archivo y cómo funciona el código:

1. Data.hpp

Este archivo define la estructura Data y declara las funciones de serialización y deserialización.

Código:

```
#pragma once

#include <cstdlib>     // std::rand, std::srand, std::atoi ...
#include <iostream>    // std::cout ...
#include <stdint.h>    // uint32_t
#include <string>      // std::string

#define GREEN "\033[1;32m"  // Define el color verde oscuro
#define RED "\033[1;31m"    // Define el color rojo oscuro
#define RESET "\033[0m"     // Define el color de reset

typedef struct data {
```

```
    std::string str;
} Data;  // Se define un struct con un solo miembro: una cadena de texto

uintptr_t serialize(Data* ptr);     // Función para serializar un puntero a Data
Data* deserialize(uintptr_t converted); // Función para deserializar un uintptr_t a un puntero Data
```

Explicación:

- Se **define** una estructura Data con un único miembro: std::string str.
- Se **declaran** dos funciones:
    - serialize(Data* ptr): Convierte un puntero Data* a un número entero uintptr_t.
    - deserialize(uintptr_t converted): Convierte el uintptr_t de vuelta a un Data*.

2. Serialization.hpp

Este archivo define una clase Serialization con métodos estáticos para realizar la serialización y deserialización.

Código:

```
#pragma once

#include "Data.hpp"

class Serialization {

  public:
    static uintptr_t serialize(Data* ptr);       // Serializa un puntero a uintptr_t
    static Data* deserialize(uintptr_t converted); // Deserializa un uintptr_t a un puntero Data

  private:
    Serialization();                    // Constructor privado
    ~Serialization();                    // Destructor privado
    Serialization(Serialization const &src);      // Constructor de copia
    Serialization &operator=(Serialization const &src); // Operador de asignación
};
```

Explicación:

- La clase Serialization **no se puede instanciar** porque:
  - Su constructor, destructor, constructor de copia y operador de asignación están en la sección private.
- Contiene dos **métodos estáticos**:
  - serialize(Data* ptr): Convierte un puntero en uintptr_t.
  - deserialize(uintptr_t converted): Convierte un uintptr_t en un puntero a Data.

3. Serialization.cpp

Este archivo implementa las funciones de serialización y deserialización.

Código:

```
#include "Serialization.hpp"

uintptr_t serialize(Data* ptr) {
   return reinterpret_cast<uintptr_t>(ptr);
} // Convierte un puntero Data* a un uintptr_t

Data* deserialize(uintptr_t converted) {
   return reinterpret_cast<Data*>(converted);
} // Convierte un uintptr_t a un puntero Data*
```

Explicación:

- serialize(Data* ptr):
  - Recibe un puntero a Data y lo **convierte** en un número entero (uintptr_t).
  - Usa reinterpret_cast para hacer la conversión de tipos sin modificar el contenido.
- deserialize(uintptr_t converted):
  - Recibe un número entero (uintptr_t) y lo **convierte** de vuelta en un puntero Data*.
  - Se usa reinterpret_cast para obtener el puntero original.

4. main.cpp

Este archivo contiene la función main que prueba el mecanismo de serialización y deserialización.

Código:

```
#include "Serialization.hpp"

using std::cerr; using std::cout; using std::endl;

int       main() {
   Data *ptr;                  // Se declara un puntero a Data
   Data *ptrNew = NULL;            // Se declara otro puntero y se inicializa en NULL

   ptr = new Data;                // Se asigna memoria para un objeto Data
   ptr->str = "Up2u 🤗 ";           // Se asigna una cadena de texto a str

   cout << "\n";
   cout << GREEN << "* * * Executing Serialization * * *\n" << RESET << endl;
   cout << "    Original ptr string: " << ptr->str << endl; // Se muestra la cadena original

   cout << "\n";

   cout << "    Original ptr : " << RED << ptr << RESET << endl;  // Se imprime la dirección del puntero
original
   uintptr_t converted = serialize(ptr);  // Se serializa el puntero
   cout << "      Serialized : " << converted << endl;  // Se muestra el valor convertido
   ptrNew = deserialize(converted);  // Se deserializa el uintptr_t a un puntero
   cout << "    Deserialized : " << RED << ptrNew << RESET << endl; // Se imprime la dirección después
de deserializar

   cout << "\n";
   cout << "    And, new ptr string: " << ptrNew->str << endl; // Se imprime la cadena del puntero
deserializado
   cout << "\n";
   cout << GREEN << "* * *  Ending  Serialization  * * *\n" << RESET << endl;

   delete ptr;  // Se libera la memoria
}
```

Explicación paso a paso:

1. Se declara un puntero ptr y otro puntero ptrNew inicializado en NULL.
2. Se asigna memoria a ptr y se le asigna la cadena "Up2u 🤗 ".

3. Se imprime la cadena de ptr.
4. Se imprime la **dirección de memoria** del puntero ptr.
5. Se llama a serialize(ptr), convirtiéndolo en un uintptr_t (valor numérico).
6. Se imprime el valor serializado.
7. Se llama a deserialize(converted), obteniendo un nuevo puntero ptrNew.
8. Se imprime la dirección del puntero después de la deserialización.
9. Se imprime la cadena de ptrNew para verificar que **sigue intacta**.
10. Se libera la memoria con delete ptr.

Ejemplo de salida del programa

(Suponiendo que la dirección de ptr en memoria es 0x600003e40)

* * * Executing Serialization * * *

Original ptr string: Up2u 🤗

Original ptr : 0x600003e40
Serialized : 1056964608
Deserialized : 0x600003e40

And, new ptr string: Up2u 🤗

* * * Ending Serialization * * *

¿Qué está sucediendo?

1. serialize(ptr) convierte la dirección 0x600003e40 en un número (uintptr_t).
2. deserialize(converted) recupera la dirección original (0x600003e40).
3. Se verifica que ptrNew es igual a ptr y que la información sigue intacta.

Conclusión

- Este código demuestra cómo **serializar y deserializar un puntero** en C++.
- Se usa uintptr_t para **almacenar direcciones de memoria de forma portable**.
- reinterpret_cast permite **convertir tipos de datos sin modificar su contenido**.
- La prueba en main.cpp confirma que el proceso funciona correctamente.

Este mecanismo es útil cuando se necesita almacenar o transmitir punteros de forma segura.