Explicación archivos ROS2_comienzo

robot_news_station.cpp

Este es un programa ROS2 en C++ que define un nodo llamado `RobotNewsStationNode`. Aquí está el desglose:

1. Inclusión de bibliotecas necesarias

```
#include "rclcpp/rclcpp.hpp"
#include "example_interfaces/msg/string.hpp"
```

Incluye las bibliotecas necesarias para usar ROS2 y los mensajes de tipo `String`.

2. Definición de la clase `RobotNewsStationNode`

```
class RobotNewsStationNode : public rclcpp::Node{
```

Define una nueva clase llamada `RobotNewsStationNode` que hereda de `rclcpp::Node`, lo que significa que esta clase es un nodo ROS2.

3. Constructor de la clase

```
public:
  RobotNewsStationNode() : Node("robot_news_station"){
    publisher_ =
create_publisher<example_interfaces::msg::String>("robot_news"
, 10);
    timer_ = create_wall_timer(std::chrono::milliseconds(500),
std::bind(&RobotNewsStationNode::publishNews, this));
    RCLCPP_INFO(get_logger(), "Robot News Station has been
started.");
}
```

En el constructor de la clase, se inicializa el nodo con el nombre "robot_news_station". Se crea un publicador que publica en el tema "robot_news". Se crea un temporizador que llama a la función `publishNews` cada 500 milisegundos. Finalmente, se imprime un mensaje de información.

4. Miembros de datos privados

```
private:
  std::string robot_name_{"R2D2"};

rclcpp::Publisher<example_interfaces::msg::String>::SharedPtr
publisher_{};
```

```
  rclcpp::TimerBase::SharedPtr timer_{};
```

Declara tres miembros de datos privados: `robot_name_` que es una cadena que contiene el nombre del robot, `publisher_` que es un puntero compartido a un objeto `Publisher`, y `timer_` que es un puntero compartido a un objeto `TimerBase`.

5. Método `publishNews`

```
   void publishNews(){
       auto msg = example_interfaces::msg::String();
       msg.data = std::string("Hi, this is ") + robot_name_ +
std::string(" from the Robots News Station");
       publisher_->publish(msg);
   }
```

Define un método que crea un mensaje de tipo `String`, establece su contenido y luego lo publica utilizando el publicador.

6. Función `main`

```
   int main(int argc, char **argv){
       rclcpp::init(argc, argv);
       auto node = std::make_shared<RobotNewsStationNode>();
       rclcpp::spin(node);
       rclcpp::shutdown();
       return 0;
   }
```

En la función `main`, se inicializa ROS2, se crea una instancia de `RobotNewsStationNode`, se inicia el ciclo de eventos de ROS2 con `rclcpp::spin`, y finalmente se cierra ROS2 con `rclcpp::shutdown`.

En resumen, este programa crea un nodo ROS2 que publica un mensaje cada 500 milisegundos en el tema "robot_news".

smartphone.cpp

1. Definición de la clase `SmartphoneNode`

```
 class SmartphoneNode : public rclcpp::Node{
```

Define una nueva clase llamada `SmartphoneNode` que hereda de `rclcpp::Node`, lo que significa que esta clase es un nodo ROS2.

2. Constructor de la clase

```
public:
  SmartphoneNode() : Node("smartphone"){
    subscriber_ =
create_subscription<example_interfaces::msg::String>
("robot_news", 10,
std::bind(&SmartphoneNode::callbackRobotNews, this,
std::placeholders::_1));
    RCLCPP_INFO(get_logger(), "Smartphone has been started.");
}
```

En el constructor de la clase, se inicializa el nodo con el nombre "smartphone". Se crea un suscriptor que se suscribe al tema "robot_news". Cuando se recibe un mensaje en este tema, se llama a la función `callbackRobotNews`. Finalmente, se imprime un mensaje de información.

3. Miembro de datos privado

```
private:
rclcpp::Subscription<example_interfaces::msg::String>::SharedP
tr subscriber_;
```

Declara un miembro de datos privado `subscriber_` que es un puntero compartido a un objeto `Subscription`.

4. Método `callbackRobotNews`

```
   void callbackRobotNews(const
example_interfaces::msg::String::SharedPtr msg){
       RCLCPP_INFO(get_logger(), "%s", msg->data.c_str());
   }
```

Define un método que se llama cuando se recibe un mensaje en el tema al que se suscribe el nodo. Este método imprime el contenido del mensaje.

En resumen, este programa crea un nodo ROS2 que se suscribe al tema "robot_news" e imprime cualquier mensaje que reciba en este tema.

number_counter.cpp

La clase es similar a las otras dos anteriores, pero quiero aclarar el uso de `std::bind` y `std::placeholders.`

`std::bind` y `std::placeholders` son utilidades en C++ que permiten crear un objeto de función (también conocido como un "functor") que "envuelve" a otra función con algunos de sus argumentos "preestablecidos" o "vinculados".

`std::bind` toma una función y posiblemente algunos argumentos, y devuelve un nuevo objeto de función que, cuando se llama, invoca la función original con los argumentos dados.

`std::placeholders` se utiliza con `std::bind` para permitir que algunos de los argumentos de la función original se pasen en el momento de la invocación del objeto de función. Los marcadores de posición `_1`, `_2`, `_3`, etc., representan el primer, segundo, tercer, etc., argumento que se pasará al objeto de función cuando se invoque.

En el código, `std::bind(&NumberCounterNode::callbackNumber, this, std::placeholders::_1)` está creando un objeto de función que, cuando se invoca, llamará al método `callbackNumber` en el objeto `NumberCounterNode` actual (`this`), con el argumento que se pasa en el momento de la invocación (`std::placeholders::_1`). Esto es útil porque permite que el método `callbackNumber` se utilice como un callback que puede ser llamado con un solo argumento, a pesar de que es un método de un objeto y normalmente requeriría un puntero al objeto como primer argumento.

`std::bind` y las expresiones lambda en C++ pueden usarse para lograr objetivos similares. Ambos te permiten crear objetos de función "en el lugar" que pueden ser pasados a otras funciones o almacenados para su uso posterior. Sin embargo, hay algunas diferencias clave.

Las expresiones lambda son una característica más nueva en C++ (introducida en C++11) y son generalmente más flexibles y fáciles de usar que `std::bind`. Las lambdas pueden capturar variables del entorno circundante, pueden tener estados y pueden tener tipos de retorno inferidos automáticamente, lo que las hace muy convenientes para muchas situaciones.

Por otro lado, `std::bind` es una característica más antigua (introducida en C++98 como parte de la biblioteca Boost y luego estandarizada en C++11) y tiene una sintaxis más complicada. Sin embargo, `std::bind` puede ser útil en algunos casos donde necesitas un mayor control sobre cómo se pasan los argumentos a la función.

En este archivo, la línea que usa `std::bind` podría ser reemplazada por una expresión lambda de la siguiente manera:

```
subscriber_ =
create_subscription<example_interfaces::msg::Int64>
              ("number", 10, [this](const
example_interfaces::msg::Int64::SharedPtr& msg) {
                  callbackNumber(msg);
              });
```

Esta expresión lambda hace lo mismo que la versión con `std::bind`: crea un objeto de función que, cuando se invoca, llama al método `callbackNumber` en el objeto `NumberCounterNode` actual (`this`), con el argumento que se pasa en el momento de la invocación.

add_two_ints_server.cpp

```
void callback_add_two_ints(const
example_interfaces::srv::AddTwoInts::Request::SharedPtr
request,
const example_interfaces::srv::AddTwoInts::Response::SharedPtr
response){
            response->sum = request->a + request->b;
         }
```

En este código, request es un puntero compartido a la solicitud de servicio, que tiene los miembros a y b. response es un puntero compartido a la respuesta de servicio, que tiene el miembro sum. El callback suma a y b de la solicitud y almacena el

resultado en sum de la respuesta. La función no tiene ningún tipo de retorno, con rellenar el response es suficiente.

add_two_ints_client.cpp

Explicación de todo el archivo:

Este código define un nodo cliente ROS2 en C++ para el servicio "add_two_ints". Este servicio es de tipo `**example_interfaces::srv::AddTwoInts**`, que toma dos enteros y devuelve su suma.

La clase `AddTwoIntsClientNode` hereda de `rclcpp::Node` y tiene un método `callAddTwoIntsService` que realiza una llamada al servicio "add_two_ints".

En el constructor de `AddTwoIntsClientNode`, se crea un hilo que ejecuta el método `callAddTwoIntsService` con los argumentos 1 y 4.

El método `callAddTwoIntsService` hace lo siguiente:

1. Crea un cliente para el servicio "add_two_ints".
2. Espera hasta que el servicio esté disponible.
3. Crea una solicitud de servicio con los valores `a` y `b` proporcionados.
4. Envía la solicitud de servicio de forma asíncrona y obtiene un `std::future` que eventualmente contendrá la respuesta.
5. Intenta obtener la respuesta del `std::future`. Si la respuesta está disponible, imprime la suma. Si ocurre una excepción (por ejemplo, si el servicio no está disponible), imprime un mensaje de error.

Finalmente, en la función `main`, se crea una instancia de `AddTwoIntsClientNode`, se inicia el nodo con `rclcpp::spin(node)`, y luego se cierra el nodo con `rclcpp::shutdown()`.

std::future y std::shared_future

`` `auto future{client->async_send_request(request)};` `` está haciendo una solicitud de servicio asincrónica en ROS2.

El método `` `async_send_request` `` de un cliente de servicio envía una solicitud al servicio y devuelve un objeto `**std::future**`. Este objeto `future` puede ser utilizado para obtener la respuesta del servicio una vez que esté disponible.

Un `**std::future**` es una promesa de que un valor estará disponible en algún momento en el futuro. Puedes llamar al método `get()` en un `**std::future**` para obtener el valor. Si el valor aún no está disponible (es decir, la respuesta del servicio aún no ha llegado), `get()` bloqueará hasta que el valor esté disponible.

Por lo tanto, esta línea de código envía una solicitud de servicio y guarda la promesa de una respuesta en la variable `future`. Luego puedes hacer algo como `auto response = future.get();` para obtener la respuesta una vez que esté disponible.

El tipo de `future` en este contexto es `` `std::shared_future<example_interfaces::srv::AddTwoInts::Response::SharedPtr>` ``.

Cuando se llama a `` `async_send_request` ``, se devuelve un `**std::shared_future**` que eventualmente contendrá la respuesta del servicio. Un `**std::shared_future**` es similar a un `**std::future**`, pero permite que el valor futuro sea accedido por múltiples hilos de ejecución.

Sobre los hilos

Cuando se crea un nuevo hilo y se le pasa una función para ejecutar, esa función se ejecuta en paralelo con el resto del programa. Esto se conoce como "multithreading" o "multihilos".

En el código, cuando se crea el objeto `AddTwoIntsClientNode`, se lanza un nuevo hilo que ejecuta la función `` `callAddTwoIntsService` ``. Esta función se ejecuta en paralelo con el resto del programa.

Mientras tanto, el hilo principal del programa continúa ejecutándose. Llama a `` `rclcpp::spin(node)` ``, que bloquea y entra en un bucle de procesamiento de eventos hasta que se cierra el nodo. Durante este tiempo, la función `` `callAddTwoIntsService` `` puede seguir ejecutándose en su propio hilo.

Cuando un hilo termina su ejecución, simplemente se detiene. Los recursos que el sistema operativo asignó al hilo se liberan y el hilo ya no existe.

En el código, cuando la función `` `callAddTwoIntsService` `` termina de ejecutarse, el hilo que se creó para ejecutar esa función también termina.

Es importante tener en cuenta que si el hilo principal del programa termina (por ejemplo, si llega al final de la función `main`), todos los hilos secundarios se

detendrán automáticamente, independientemente de si han terminado de ejecutarse o no.

Por lo tanto, si necesitas que un programa espere a que un hilo secundario termine antes de que el programa pueda terminar, puedes llamar al método `join` en el objeto `**std::thread**`. Esto bloqueará el hilo principal hasta que el hilo secundario haya terminado.

thread pools

Un "thread pool" o "pool de hilos" es un patrón de diseño de software en el que se crea un conjunto de hilos de trabajo al inicio de un programa y estos hilos se reutilizan para realizar tareas a lo largo de la vida del programa.

En lugar de crear un nuevo hilo cada vez que se necesita realizar una tarea en paralelo, se puede asignar una tarea a uno de los hilos del pool de hilos. Cuando un hilo del pool termina una tarea, vuelve al pool y espera la próxima tarea.

El uso de un pool de hilos puede mejorar el rendimiento del programa al reducir el tiempo y los recursos del sistema necesarios para crear y destruir hilos. También permite limitar el número de hilos que se ejecutan simultáneamente, lo que puede ser útil para controlar el uso de recursos en sistemas con recursos limitados.

En C++, puedes usar la biblioteca estándar para crear un pool de hilos. La clase `**std::thread**` se utiliza para representar los hilos individuales, y puedes usar una estructura de datos como `**std::vector**` o `**std::queue**` para almacenar los hilos del pool. También necesitarás algún tipo de mecanismo de sincronización, como `**std::mutex**` y `**std::condition_variable**`, para coordinar el acceso a los hilos del pool.

Es importante tener en cuenta que la gestión de un pool de hilos puede ser compleja, especialmente en lo que respecta a la sincronización de hilos y la gestión de la vida útil de los hilos. En muchos casos, puede ser más fácil y seguro usar una biblioteca de terceros que proporcione una implementación de pool de hilos.

battery.cpp

Casi todo se ha visto ya, pero nunca está de más repasar.

La clase '**BatteryNode'** tiene dos miembros privados:

std::thread thread1_{}; - Un objeto de la clase std::thread que puede ser utilizado para lanzar un nuevo hilo de ejecución.

bool battery_state_{}; - Una variable booleana que representa el estado de la batería. El valor por defecto es false (indicado por {}).

En el constructor, `thread1_ = std::thread(&BatteryNode::toggleBatteryState, this);` Aquí se está creando un nuevo hilo de ejecución que ejecuta el método `toggleBatteryState` de la instancia actual de '**BatteryNode'**. Esto significa que `toggleBatteryState` se ejecutará en paralelo al hilo principal del programa.

El método `` `toggleBatteryState()` `` realiza las siguientes operaciones en un bucle infinito:

1. Si `**battery_state_**` es `true`, el hilo actual se pone a dormir durante 4 segundos. Si `**battery_state_**` es `false`, el hilo actual se pone a dormir durante 6 segundos. Esto se logra utilizando `` `std::this_thread::sleep_for(std::chrono::seconds(n))` ``, que bloquea la ejecución del hilo actual durante el tiempo especificado.

2. Después de dormir, `**battery_state_**` se invierte utilizando el operador `!`. Si `**battery_state_**` era `true`, se convierte en `false`, y viceversa.

3. Luego, se llama al método `` `callSetLed(3, battery_state_)` ``. Esto envía una solicitud al servicio "set_led" para cambiar el estado del LED número 3 al valor actual de `**battery_state_**`.

4. Este método se ejecuta en un bucle infinito, por lo que continuará alternando el estado de la batería y enviando solicitudes al servicio "set_led" hasta que el programa se detenga.

El método público `callSetLed(int a, bool b)` realiza lo siguiente:

1. Crea un cliente de servicio con el nombre "set_led". Este cliente puede ser utilizado para enviar solicitudes al servicio "set_led".

2. Espera hasta que el servicio esté disponible. Si el servicio no está disponible, imprime una advertencia y vuelve a intentarlo cada segundo.

3. Una vez que el servicio está disponible, crea una solicitud con los parámetros a y b que representan el número de LED y el estado, respectivamente.

4. Envía la solicitud al servicio de forma asíncrona. Esto significa que el programa no se bloqueará y podrá realizar otras tareas mientras espera la respuesta del servicio.

5. Una vez que recibe la respuesta del servicio, imprime un mensaje de información. Si ocurre una excepción (por ejemplo, si el servicio no responde), imprime un mensaje de error.

led_panel.cpp

En esta clase no hay nada nuevo, todo lo que contiene ya se ha visto en las clases anteriores.

std::ref en tortuga_spawner2.cpp

El operador `**&**` y `**std::ref**` son cosas diferentes y se utilizan en contextos diferentes en C++.

El operador `&` se utiliza para obtener la dirección de memoria de una variable, es decir, crea un puntero a la variable. Este operador se utiliza comúnmente cuando se quiere pasar una variable por referencia a una función en C.

Por otro lado, `std::ref` es una función de la biblioteca estándar de C++ que crea un objeto de tipo `**std::reference_wrapper**`. Este objeto puede utilizarse para pasar una referencia a una función que espera un argumento por valor. Esto es útil cuando se trabaja con hilos o con cualquier otra función de la biblioteca estándar de C++ que requiere que los argumentos sean pasables por valor.

En este caso, se trabaja con `std::thread`, que requiere que los argumentos sean pasables por valor. Por lo tanto, necesitas utilizar `std::ref` para pasar una referencia a `array_tortugas_.lista_tortugas.back()`. Si intentas utilizar el operador `&` en su lugar, se obtiene un puntero, lo cual no es lo que se requiere en este caso.

Diferencia .at(0) y .begin() en tortuga_controller2.cpp

La diferencia entre estas dos líneas de código radica en lo que devuelven y cómo se utilizan.

`tortugas_vivas->lista_tortugas.at(0)`; devuelve una **referencia al primer elemento del vector** lista_tortugas. Puedes usarlo para leer o modificar el primer elemento del vector.

Por otro lado, `tortugas_vivas->lista_tortugas.begin();` devuelve un **iterador** al primer elemento del vector lista_tortugas. Los iteradores se utilizan para recorrer los elementos de un contenedor, como un vector o una lista.