Using C++ Classes in ROS: a minimal example

Wyatt Newman
February, 2015

ROS code can quickly become overly long. To improve productivity and cod re-use, it is desirable to use classes.

Use of classes in C++ is detailed in any text on C++. In general, it is desirable to:

- define a class in a header file
  - define prototypes for all member functions of the class
  - define member variables, which will be accessible to all member functions of the class
  - define a prototype for a class constructor function
- write a separate "implementation" file that
  - includes the above header file
  - contains the working code for the declared member functions
  - contains code to encapsulate necessary initializations in the constructor
  - contains a "main()" function that "instantiates" an instance of the new class

An example is provided in the shared package: example_ros_class. The header file of this example follows:

```
// example_ros_class.h header file //
// wsn; Feb, 2015
// include this file in "example_ros_class.cpp"

// here's a good trick--should always do this with header files:
// create a unique mnemonic for this header file, so it will get included if needed,
// but will not get included multiple times
#ifndef EXAMPLE_ROS_CLASS_H_
#define EXAMPLE_ROS_CLASS_H_

//some generically useful stuff to include...
#include <math.h>
#include <stdlib.h>
#include <string>
#include <vector>

#include <ros/ros.h> //ALWAYS need to include this

//message types used in this example code;  include more message types, as needed
#include <std_msgs/Bool.h>
#include <std_msgs/Float32.h>

#include <cwru_srv/simple_bool_service_message.h>
// this is a pre-defined service message, contained in shared "cwru_srv" package

// define a class, including a constructor, member variables and member functions
class ExampleRosClass
{
```

```
public:
    ExampleRosClass(ros::NodeHandle* nodehandle);
   //"main" will need to instantiate a ROS nodehandle, then pass it to the constructor
    // may choose to define public methods or public variables, if desired
private:
    // put private member data here;
    // "private" data will only be available to member functions of this class;
    ros::NodeHandle nh_; // we will need this, to pass between "main" and constructor
    // some objects to support subscriber, service, and publisher
    ros::Subscriber minimal_subscriber_;
      //these will be set up within the class constructor, hiding these ugly details
    ros::ServiceServer minimal_service_;
    ros::Publisher  minimal_publisher_;

    double val_from_subscriber_;
    //example member variable:
     //better than using globals; convenient way to pass data from a subscriber to other member functions
    double val_to_remember_;
     // member variables will retain their values even as callbacks come and go

    // member methods as well:
    void initializeSubscribers(); // we will define some helper methods to encapsulate the
    // gory details of initializing subscribers, publishers and services
    void initializePublishers();
    void initializeServices();

    void subscriberCallback(const std_msgs::Float32& message_holder);
    //prototype for callback of example subscriber
    //prototype for callback for example service
    bool serviceCallback(cwru_srv::simple_bool_service_messageRequest& request,
cwru_srv::simple_bool_service_messageResponse& response);
}; // note: a class definition requires a semicolon at the end of the definition

#endif  // this closes the header-include trick...ALWAYS need one of these to match #ifndef
```

The above header file defines the structure of a new class, “ExampleRosClass”. This class contains a constructor (ExampleRosClass(ros::NodeHandle* nodehandle);) , and a variety of private member objects, functions and variables. These entities are all accessible by member functions of the class. Defining objects for publisher, subscriber and service allows set-up for these to be performed by the constructor, thus simplifying the main program.

Prototypes for the member functions are defined there, declaring the names, return types and argument types. The executable code constituting the “implementation” of these functions is contained in one or more separate *.cpp files. One should avoid putting implementation code in a header (except for very short implementations, such as “get()” and “set()” functions).

Note that the entire header file is contained within a compiler macro starting with #ifndef EXAMPLE_ROS_CLASS_H_ and ending with #endif. This trick should always be used in writing

header files. It helps the compiler to avoid including redundant copies of headers.

The implementation code corresponding to the above header file follows:

```
//example_ros_class.cpp:
//wsn, Feb 2015
//illustrates how to use classes to make ROS nodes
// constructor can do the initialization work, including setting up subscribers, publishers and services
// can use member variables to pass data from subscribers to other member functions

// can test this function manually with terminal commands, e.g. (in separate terminals):
// rosrun example_ros_class example_ros_class
// rostopic echo exampleMinimalPubTopic
// rostopic pub -r 4 exampleMinimalSubTopic std_msgs/Float32 2.0
// rosservice call exampleMinimalService 1


// this header incorporates all the necessary #include files and defines the class "ExampleRosClass"
#include "example_ros_class.h"

//CONSTRUCTOR:  this will get called whenever an instance of this class is created
// want to put all dirty work of initializations here
// odd syntax: have to pass nodehandle pointer into constructor for constructor to build subscribers, etc
ExampleRosClass::ExampleRosClass(ros::NodeHandle* nodehandle):nh_(*nodehandle)
{ // constructor
   ROS_INFO("in class constructor of InteractivePathMaker");
   initializeSubscribers();
   // package up the messy work of creating subscribers; do this overhead in constructor
   initializePublishers();
   initializeServices();

   //initialize variables here, as needed
   val_to_remember_=0.0;

   // can also do tests/waits to make sure all required services, topics, etc are alive
}

//member helper function to set up subscribers;
// note odd syntax: &ExampleRosClass::subscriberCallback is a pointer
// to a member function of ExampleRosClass
// "this" keyword is required, to refer to the current instance of ExampleRosClass
void ExampleRosClass::initializeSubscribers()
{
   ROS_INFO("Initializing Subscribers");
   minimal_subscriber_ = nh_.subscribe("exampleMinimalSubTopic", 1,
&ExampleRosClass::subscriberCallback,this);
   // add more subscribers here, as needed
}
```

```
//member helper function to set up services:
// similar syntax to subscriber, required for setting up services outside of "main()"
void ExampleRosClass::initializeServices()
{
   ROS_INFO("Initializing Services");
   minimal_service_ = nh_.advertiseService("exampleMinimalService",
                                           &ExampleRosClass::serviceCallback,
                                           this);
   // add more services here, as needed
}

//member helper function to set up publishers;
void ExampleRosClass::initializePublishers()
{
   ROS_INFO("Initializing Publishers");
   minimal_publisher_ = nh_.advertise<std_msgs::Float32>("exampleMinimalPubTopic", 1, true);
   //add more publishers, as needed
   // note: COULD make minimal_publisher_ a public member function,
   // if want to use it within "main()"
}



// a simple callback function, used by the example subscriber.
// note, though, use of member variables and access to minimal_publisher_
// (which is a member method)
void ExampleRosClass::subscriberCallback(const std_msgs::Float32& message_holder) {
   // the real work is done in this callback function
   // it wakes up every time a new message is published on "exampleMinimalSubTopic"

   val_from_subscriber_ = message_holder.data;
   // copy the received data into member variable, so ALL member funcs of
  // ExampleRosClass can access it
   ROS_INFO("myCallback activated: received value %f",val_from_subscriber_);
   std_msgs::Float32 output_msg;
   val_to_remember_ += val_from_subscriber_;
   //can use a member variable to store values between calls; add incoming value each callback
   output_msg.data= val_to_remember_;
   // demo use of publisher--since publisher object is a member function
   minimal_publisher_.publish(output_msg); //output the square of the received value;
}

//member function implementation for a service callback function
bool ExampleRosClass::serviceCallback(cwru_srv::simple_bool_service_messageRequest& request,
cwru_srv::simple_bool_service_messageResponse& response) {
   ROS_INFO("service callback activated");
   response.resp = true; // boring, but valid response info
   return true;
}
```

```
int main(int argc, char** argv)
{
   // ROS set-ups:
   ros::init(argc, argv, "exampleRosClass"); //node name

   ros::NodeHandle nh; // create a node handle; need to pass this to the class constructor

   ROS_INFO("main: instantiating an object of type ExampleRosClass");
   ExampleRosClass exampleRosClass(&nh);
//instantiate an ExampleRosClass object and pass in pointer to nodehandle for constructor to use

   ROS_INFO("main: going into spin; let the callbacks do all the work");
   ros::spin();
   return 0;
}
```

In the above, the ROS service definition and the ROS publisher are similar to previous minimal examples. Note, though, that the subscriber is able to invoke the publisher—since the publisher is a member object that is accessible to all member functions. Further, the subscriber can copy its received data to member variables (val_from_subscriber_ , in this example), making this data available to all member functions. In addition, the subscriber callback function illustrates using a member variable (val_to_remember_) to store results between calls, since this member variable persistently holds its data between calls to the subscriber. The member variables thus behave similarly to global variables—but these variables are only available to class member methods, and thus this construction is preferred.

The constructor is responsible for setting up the example publisher, example subscriber and example service. A somewhat odd notation is required, though. Main() must instantiate a node handle for the constructor to use—and this value must be passed to the constructor. But the constructor is responsible for initializing class variables, which creates a chicken-and-egg problem. This is resolved with the somewhat odd notation:

```
ExampleRosClass::ExampleRosClass(ros::NodeHandle* nodehandle):nh_(*nodehandle)
```

which allows the main program to create an instance of the class ExampleRosClass, passing into the constructor the nodehandle created by “main.”

```
  ExampleRosClass exampleRosClass(&nh);
```

Note also the somewhat odd set-up of the subscriber and the service, which is performed within the constructor:

```
minimal_subscriber_ = nh_.subscribe("exampleMinimalSubTopic", 1,
&ExampleRosClass::subscriberCallback,this);
```

and

```
minimal_service_ = nh_.advertiseService("exampleMinimalService",
                                        &ExampleRosClass::serviceCallback,
                                        this);
```

The notation: &ExampleRosClass::subscriberCallback provides a pointer to the callback function to be used with the subscriber. This callback function is defined as a member function of the current class. Additionally, the keyword “this” tell the compiler that we are referring to the current instance of this class.

Although the notation is somewhat cumbersome, it is convenient design our ROS nodes in this fashion. The constructor is then able to encapsulate the details of setting up publishers, subscribers and services, as well as initialize all important variables and, as necessary, test that required topics and services from companion nodes are active and healthy, before releasing control to the “main” program.

In the present example, the main program is very short. It merely creates an instance of the new class, then it goes into a “spin.” All of the program work is then performed by callbacks of the new class object.

The example code can be tested by running (in separate terminals):

roscore

rosrun example_ros_class example_ros_class

Then peek/poke the various I/O options from a command line (in 3 separate terminals) with:

rostopic echo exampleMinimalPubTopic

rostopic pub -r 4 exampleMinimalSubTopic std_msgs/Float32 2.0

rosservice call exampleMinimalService 1