④ ROS-Cpp Overview

1) Initialization and Shutdown

=> There are two levels of initialization of a roscpp node:

① ros::init()
- Provides command line arguments to ROS
- Allows you to name your node and specify other options.

② ros::NodeHandle

=> Initialization does not contact the master.

- When the first ros::NodeHandle is created it will call ros::start(). and when the last ros::NodeHandle is distroyed, it will call ros::shutdown()

=> The most common method to check for various state of shutdown is ros::ok():

returns false if the node has finished shutting down, else return true.

② Messages

⇒ Like all ROS client Libraries, roscpp takes msg files and generate C++ source code for them.

⇒ The pattern for this is:

package_name/msg/Foo.msg

→ package_name::Foo

⇒ Similarly for srv files.

⇒ The header files are generated with the same name as the filename of the msg/srv.

③ Timers

⇒ roscpp's Timers let you schedule a callback to happen at a specific rate through the same callback queue mechanism used by subscription, service, etc.

★ Creating a Timer

⇒ Creating a Timer is done using:

```
ros::Timer timer = nh.CreateTimer(ros::Duration(0.1)
                                  , timerCallback);
```

★ Callback Signature

```
void callback(const ros::TimerEvent&);
```

→ Structure

ros::TimerEvent

- ros::Time last_expected
- ros::Time last_real
- ros::Time current_expected
- ros::Time current_real
- ros::WallTime profile.last_duration

★ Wall-clock Timers

⇒ ros::Timer uses the ROS clock when available (usually when running in simulation)

↳ If you'd like a timer that always uses wall-clock time there is a ros::WallTimer that works exactly the same except replace Timer with WallTimer in all uses.

```
Void callback(const ros::WallTimerEvent& event)
{
...
}

...
ros::WallTimer timer = nh.CreateWallTimer
                       (ros::WallDuration(0.1), callback);
```

⑨ NodeHandles

⇒ The ros::NodeHandle class serves two purposes:

① → Startup and shutdown of internal node.

② → It provides an extra layer of namespace resolution that can make writing subcomponents easier.

★ Namespaces

⇒ NodeHandles lets you specify a namespace to their constructor:

```
ros::NodeHandle nh("my_namespace");
```

⇒ You can also specify a parent NodeHandle and a namespace to append:

```
ros::NodeHandle nh1("ns1");
ros::NodeHandle nh2(nh1, "ns2");
```

⑤ Callback and Spinning

⇒ roscpp allows your callbacks to be called from any number of threads if that's what you want.

* Single-threaded spinning

⇒ The simplest (and most common) version of single-threaded spinning is `ros::spin()`.

↳ In this application all user callbacks will be called from within the ros::spin() call.

↳ ros::spin() will not return until the node has been shutdown, either through a call to ros::shutdown() or Ctrl+C

⇒ Another common pattern is to call `ros::spinOnce()` periodically:

↳ will call all the callbacks waiting to be called at that point in time.

* Multi-threaded Spinning

⇒ There are two built-in options for this:

(a) ros::MultiThreadedSpinner

⇒ It is a blocking spinner, similar to ros::spin()

```
ros::MultiThreadedSpinner spinner(4);
spinner.spin();
```

- `spin()` → (Will not return until the node has been shutdown)
- `MultiThreadedSpinner(4)` → (number of threads to use)
  - If unspecified or set to 0, it will use a thread for each CPU core

(b) ros::AsyncSpinner

⇒ A more useful threaded spinner is the AsyncSpinner.

⇒ Instead of a blocking spin() call, it has start() and stop() calls, and will automatically stop when it is destroyed.

```
ros::AsyncSpinner spinner(4);
spinner.start();
```

*CallbackQueue :: callAvailable() and callOne()

⇒ You can create callback queues this way:

```
#include <ros/callback_queue.h>
:::
ros::CallbackQueue my_queue;
```

⇒ The CallbackQueue class has two ways of invoking the callbacks inside it:

- callAvailable()
- callOne()

callOne() → Will simply invoke the oldest callback on the queue

callAvailable() → Will take everything currently in the queue and invoke all of them

⇒ Both callAvailable() and callOne() can take in an optional timeout, which is the amount of time they will wait for a callback to be available before returning.

- default: 0 (ROS 0.11)

★ Advanced :: Using Different Callback Queues

⇒ By default all callbacks get assigned into that global queue

⇒ roscpp also lets you assign custom callback queues and service them separately.

⇒ This can be done in one of two granularities:

1. Per subscribe(), advertise(), advertiseService() etc.
2. Per NodeHandle

```
ros::NodeHandle nh;
nh.setCallbackQueue(&my_callback_queue);
```

→ This means ros::spin() and ros::spinOnce() will not call these callbacks.

→ You must service that queue separately.

⇒ The various *Spinner objects can also take a pointer to a callback queue to use rather than the default one:

```
ros::AsyncSpinner spinner(0, &my_callback_queue);
spinner.start();
```

(or)

```
ros::MultiThreadedSpinner spinner(0);
spinner.spin(&my_callback_queue);
```

(b) Time

(a) Time and Duration

⇒ ROS has builin time and duration primitive types, ~~(abbrev)~~ as the ros::Time and ros::Duration classes, respectively.

Time → Specific moment

Duration → Period of time
→ can be negative.

⇒ Times and duration have identical representations:

```
int32 sec
int32 nsec
```

★ Getting the Current Time

ros::Time::now()

↳ Gets the current time as a ros::Time instance.

⇒ When using simulated clock time, now() returns time 0 until first message has been received on /clock.

↳ 0 means essentially that the client does not know clock time yet.

★ Converting Time and Duration Instance

```
double secs = ros::Time::now().toSec();
ros::Duration d(0.5);
secs = d.toSec();
```

⑥ Sleeping and Rates

```
ros::Duration(0.5).sleep();
```

↳ sleep for half a second

```
ros::Rate r(10);  ⟶ 10 hz
while (ros::ok())
{
  --- do some work. ---
  r.sleep();
}
```

⇒ Rate instance will attempt to keep the loop at 10hz by accounting for the time used by the work done during the loop.

ⓒ Wall Time

ros::WallTime
ros::WallDuration
ros::WallRate
} Identical to above

↳ If you want to access to the actual wall-clock time even if running inside simulation.