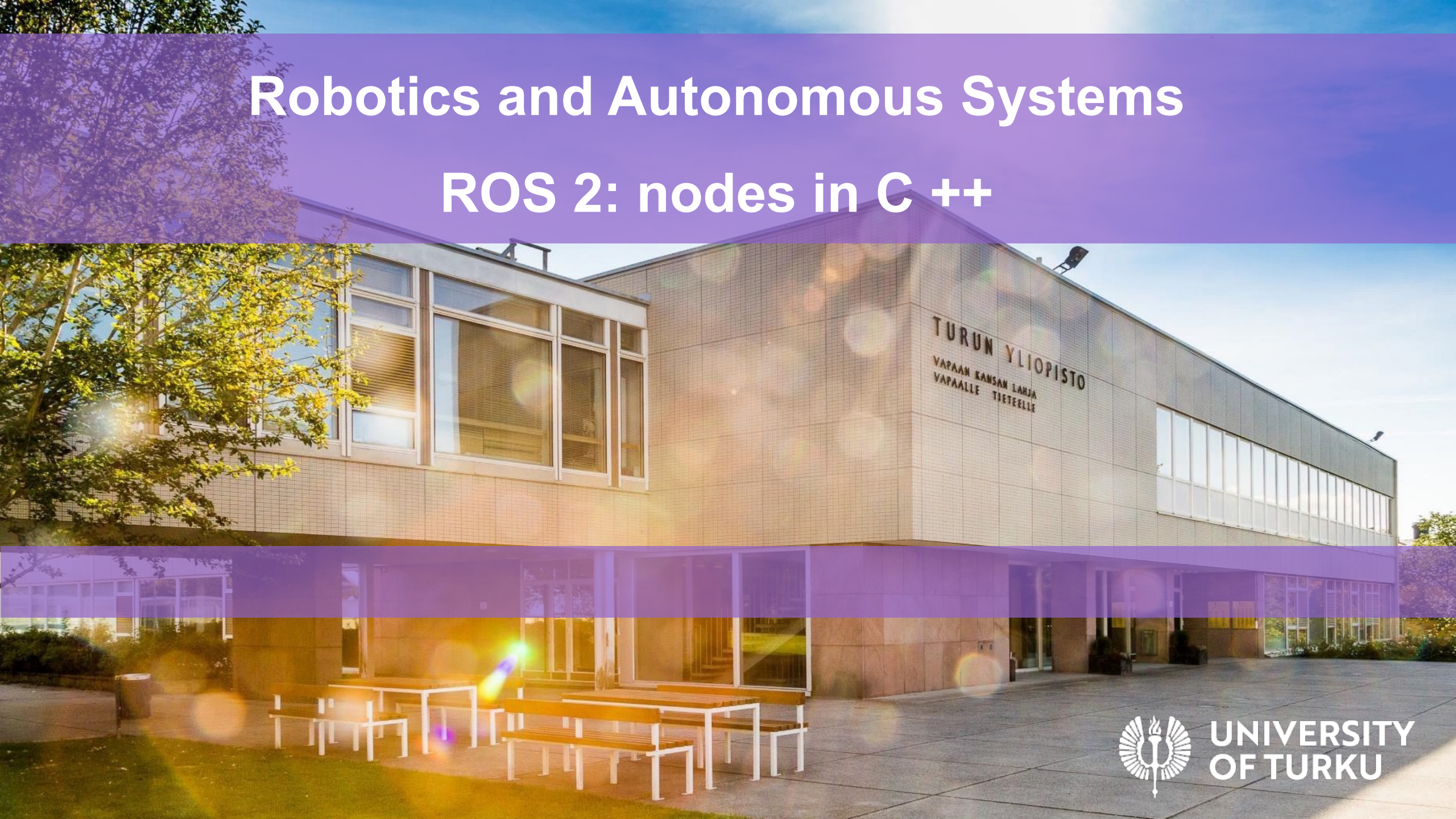
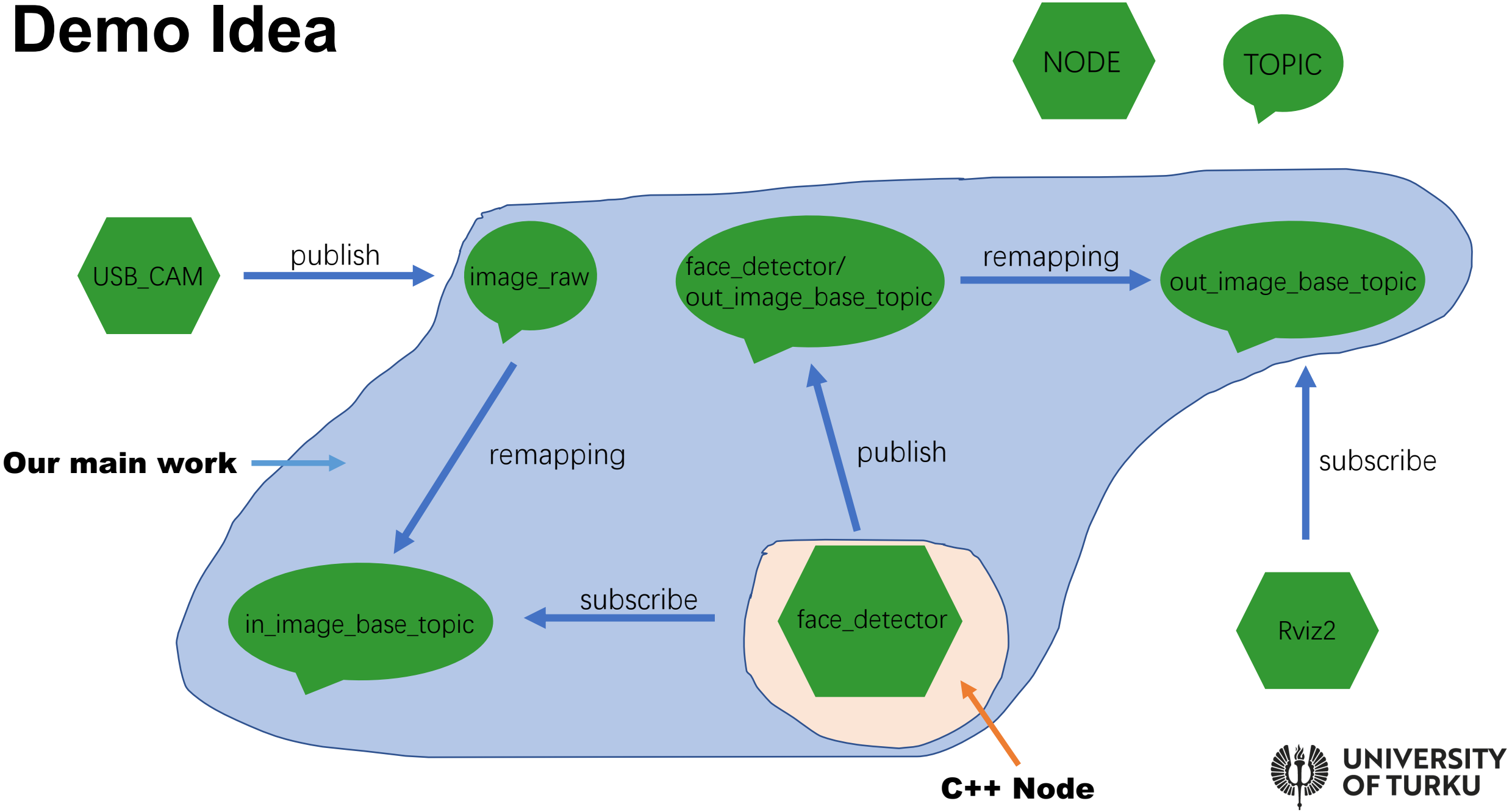
Robotics and Autonomous Systems

ROS 2: nodes in C ++



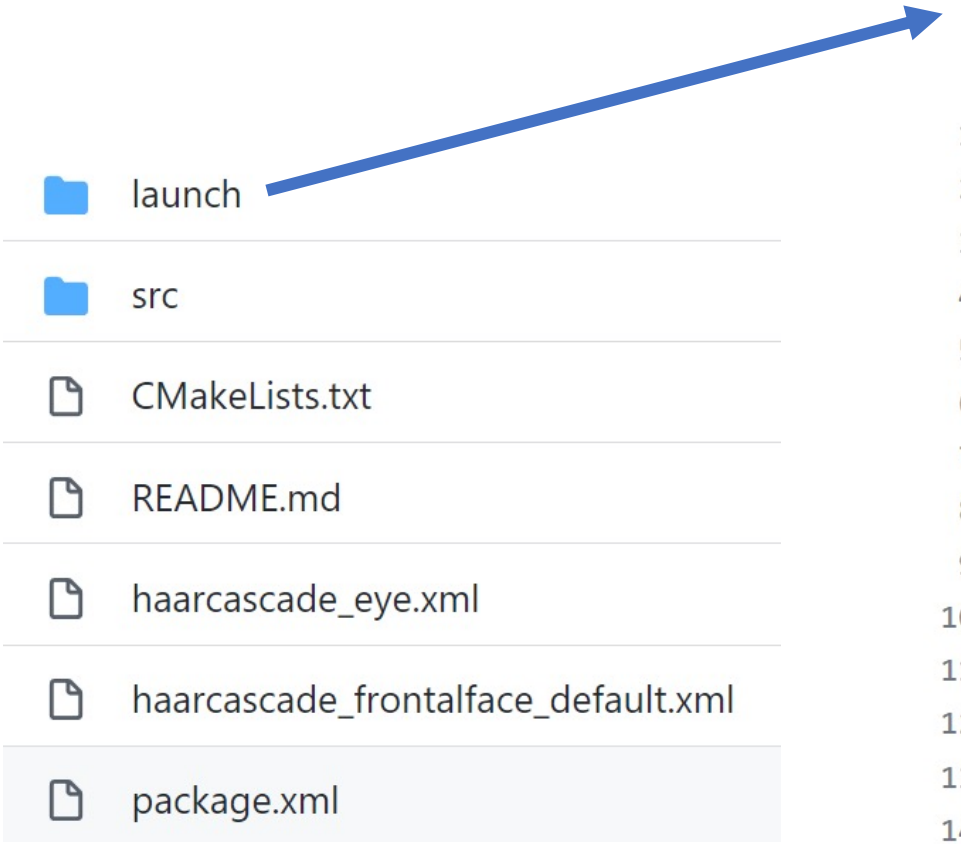
UNIVERSITY OF TURKU

Demo Idea



UNIVERSITY OF TURKU

File Structure



Launch nodes

```python
from launch import LaunchDescription
from launch_ros.actions import Node

def generate_launch_description():
    return LaunchDescription([
        Node(
            package='opencv_demos',
            executable='face_detector',
            name='face_detector',
            remappings=[
                ('in_image_base_topic', '/image_raw'),
                ('out_image_base_topic', '/face_detector/out_image_base_topic'),
            ]
        )
    ])
```

UNIVERSITY OF TURKU

File Structure

- launch
- src
- CMakeLists.txt
- README.md
- haarcascade_eye.xml
- haarcascade_frontalface_default.xml
- package.xml

Face detector

- face_detector.cpp
- haarcascade_eye.xml
- haarcascade_frontalface_default.xml

UNIVERSITY OF TURKU

File Structure

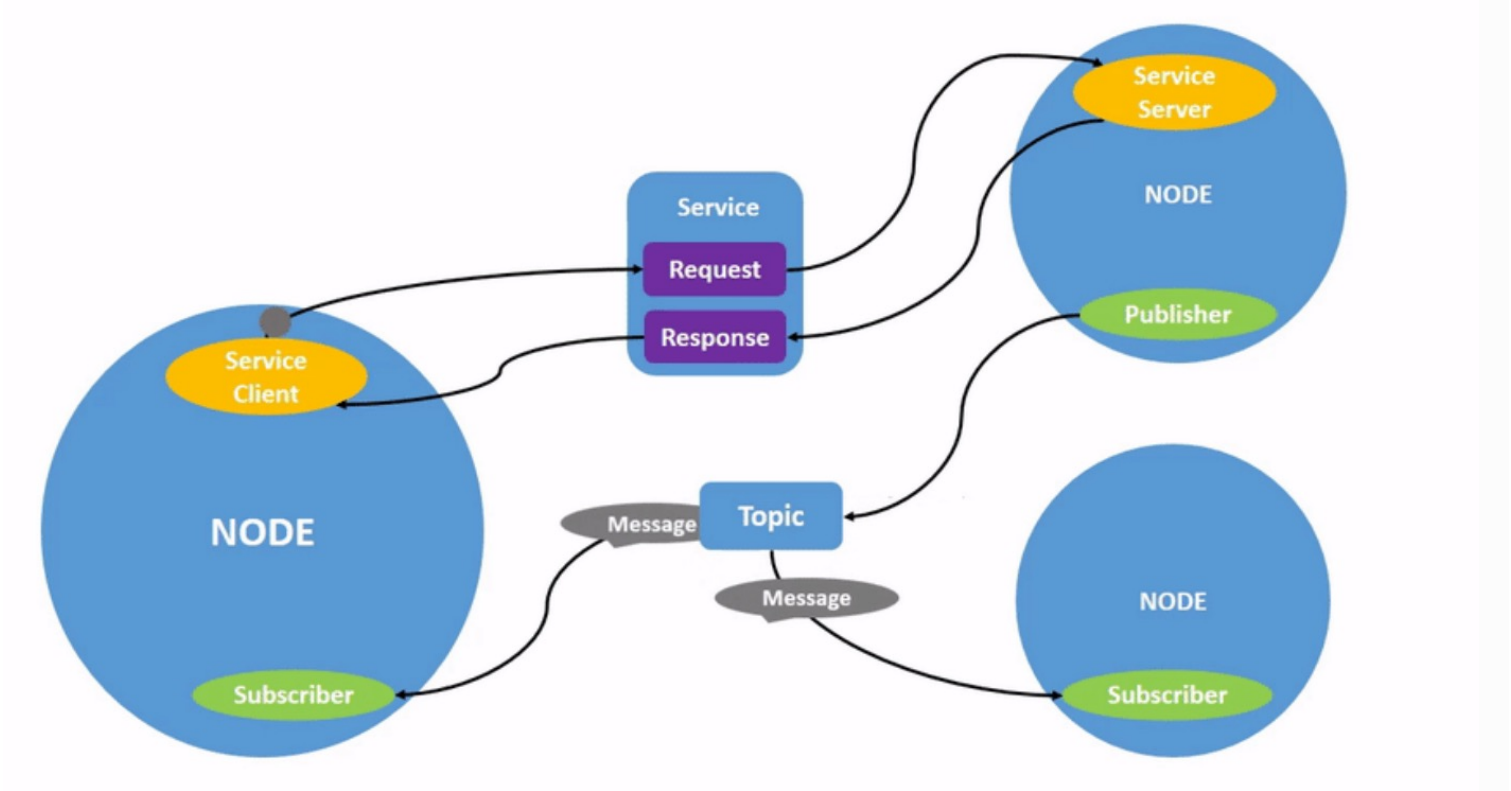
Dependency packages

```
<?xml version="1.0"?>
<?xml-model href="http://download.ros.org/schema/package_format3.xsd" schematypens="http://www.w3.org/2001/XMLSchema"?>
<package format="3">
  <name>opencv_demos</name>
  <version>0.0.0</version>
  <description>TODO: Package description</description>
  <maintainer email="robotos@todo.todo">robotos</maintainer>
  <license>TODO: License declaration</license>

  <buildtool_depend>ament_cmake</buildtool_depend>

  <test_depend>ament_lint_auto</test_depend>
  <test_depend>ament_lint_common</test_depend>

  <depend>rclcpp</depend>
  <depend>image_transport</depend>
  <depend>sensor_msgs</depend>
  <depend>cv_bridge</depend>

  <export>
    <build_type>ament_cmake</build_type>
  </export>
</package>
```

UNIVERSITY OF TURKU

A simple publisher and subscriber (C++)

Nodes are executable processes that communicate over the ROS graph. The nodes will pass information in the form of string messages to each other over a topic. The example used here is a simple “talker” and “listener” system; one node publishes data and the other subscribes to the topic so it can receive that data.



UNIVERSITY OF TURKU

Node in c++

- Create a publisher and subscriber node using C++.

1 Create a package

2 Create a publisher node

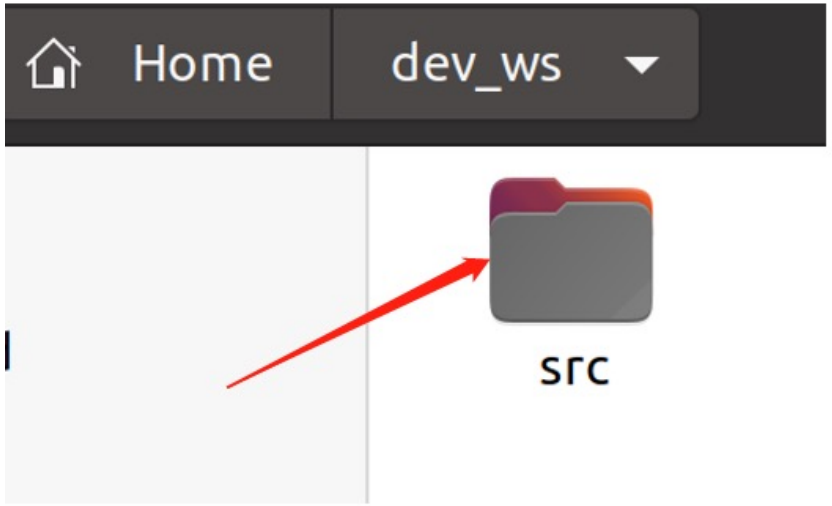
3 Set up publisher node dependencies

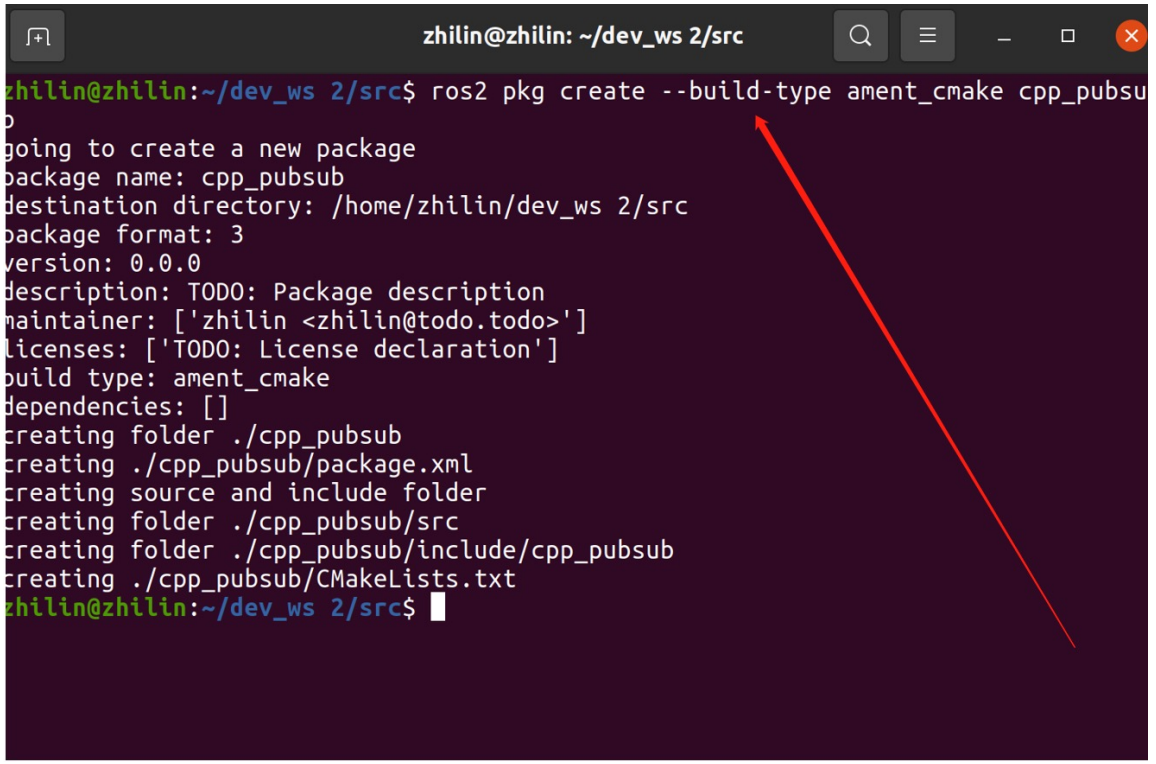
4 Set the publisher node compilation rules

5 Create subscribers

6 Compile and run

· A simple publisher and subscriber (C++)





1、Create dev_ws/src folder

2、Run the package creation command:ros2 pkg create --build-type ament_cmake cpp_pubsub

· A simple publisher and subscriber (C++)

· A simple publisher and subscriber (C++)

The top of the code includes the standard C++ headers we will be using. After the standard C++ headers is the rclcpp/rclcpp.hpp include which allows us to use the most common pieces of the ROS 2 system. Last is std_msgs/msg/string.hpp, which includes the built-in message type we will use to publish data.

· A simple publisher and subscriber (C++)

This line creates the node class MinimalPublisher by inheriting from rclcpp::Node. Every this in the code is referring to the node.

```cpp
class MinimalPublisher : public rclcpp::Node
{
  public:
    MinimalPublisher()
    : Node("minimal_publisher"), count_(0)
    {
      publisher_ = this->create_publisher<std_msgs::msg::String>("topic", 10);
      timer_ = this->create_wall_timer(500ms, std::bind(&MinimalPublisher::timer_callback, this));
    }

  private:
    void timer_callback()
    {
      auto message = std_msgs::msg::String();
      message.data = "Hello, world! " + std::to_string(count_++);
      RCLCPP_INFO(this->get_logger(), "Publishing: '%s'", message.data.c_str());
      publisher_->publish(message);
    }
    rclcpp::TimerBase::SharedPtr timer_;
    rclcpp::Publisher<std_msgs::msg::String>::SharedPtr publisher_;
    size_t count_;
  };

  int main(int argc, char * argv[])
  {
    rclcpp::init(argc, argv);
    rclcpp::spin(std::make_shared<MinimalPublisher>());
    rclcpp::shutdown();
    return 0;
  }
```

· A simple publisher and subscriber (C++)

In the constructor, a publisher is first created, the published topic name is topic, the topic message is String, and the length of the queue for saving the message is 10, and then a timer timer_ is created, making a 500 millisecond timing. Every time the timer is triggered, the callback function timer_callback is run.

```cpp
class MinimalPublisher : public rclcpp::Node
{
public:
  MinimalPublisher()
  : Node("minimal_publisher"), count_(0)
  {
    publisher_ = this->create_publisher<std_msgs::msg::String>("topic", 10);
    timer_ = this->create_wall_timer(500ms, std::bind(&MinimalPublisher::timer_callback, this));
  }

private:
  void timer_callback()
  {
    auto message = std_msgs::msg::String();
    message.data = "Hello, world! " + std::to_string(count_++);
    RCLCPP_INFO(this->get_logger(), "Publishing: '%s'", message.data.c_str());
    publisher_->publish(message);
  }
  rclcpp::TimerBase::SharedPtr timer_;
  rclcpp::Publisher<std_msgs::msg::String>::SharedPtr publisher_;
  size_t count_;
};

int main(int argc, char * argv[])
{
  rclcpp::init(argc, argv);
  rclcpp::spin(std::make_shared<MinimalPublisher>());
  rclcpp::shutdown();
  return 0;
}
```

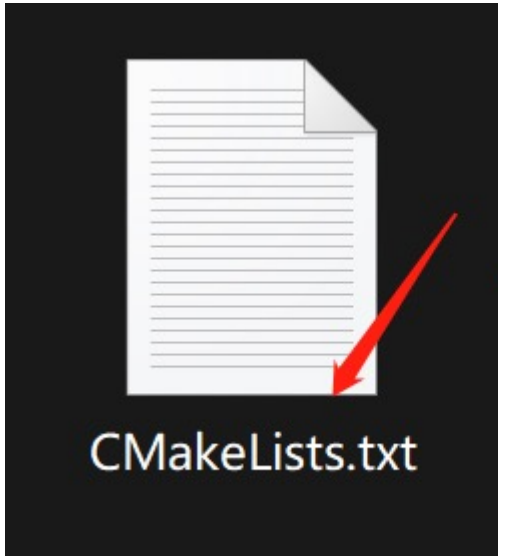
· A simple publisher and subscriber (C++)

timer_callback is the key here, a topic message will be published every time it is triggered. The string stored in the message is Hello world plus a count value, and then the log information is printed once through the RCLCPP_INFO macro function, and then the message is published through the publisher's publish method. Finally there are the declarations of the timer, publisher and counter.

```
class MinimalPublisher : public rclcpp::Node
{
  public:
    MinimalPublisher()
    : Node("minimal_publisher"), count_(0)
    {
      publisher_ = this->create_publisher<std_msgs::msg::String>("topic", 10);
      timer_ = this->create_wall_timer(500ms, std::bind(&MinimalPublisher::timer_callback, this));
    }

  private:
    void timer_callback()
    {
      auto message = std_msgs::msg::String();
      message.data = "Hello, world! " + std::to_string(count_++);
      RCLCPP_INFO(this->get_logger(), "Publishing: '%s'", message.data.c_str());
      publisher_->publish(message);
    }
    rclcpp::TimerBase::SharedPtr timer_;
    rclcpp::Publisher<std_msgs::msg::String>::SharedPtr publisher_;
    size_t count_;
};

  int main(int argc, char * argv[])
  {
    rclcpp::init(argc, argv);
    rclcpp::spin(std::make_shared<MinimalPublisher>());
    rclcpp::shutdown();
    return 0;
  }
```

· A simple publisher and subscriber (C++)

Add dependencies



Set Compilation Rules

A simple publisher and subscriber (C++)

The overall code flow of the subscriber is similar to that of the publisher. The current node name is minimal_subscriber. The subscriber is created in the constructor, subscribes to String messages, the topic name of subscription is "topic", and the queue length for saving messages is 10. When the data is received, it will enter the callback function topic_callback. The topic name and message type used by publisher and subscriber must match in order to communicate.

```cpp
#include <memory>
#include "rclcpp/rclcpp.hpp"
#include "std_msgs/msg/string.hpp"
using std::placeholders::_1;

class MinimalSubscriber : public rclcpp::Node
{
  public:
    MinimalSubscriber()
    : Node("minimal_subscriber")
    {
      subscription_ = this->create_subscription<std_msgs::msg::String>(
      "topic", 10, std::bind(&MinimalSubscriber::topic_callback, this, _1));
    }

  private:
    void topic_callback(const std_msgs::msg::String::SharedPtr msg) const
    {
      RCLCPP_INFO(this->get_logger(), "I heard: '%s'", msg->data.c_str());
    }
    rclcpp::Subscription<std_msgs::msg::String>::SharedPtr subscription_;
};

int main(int argc, char * argv[])
{
  rclcpp::init(argc, argv);
  rclcpp::spin(std::make_shared<MinimalSubscriber>());
  rclcpp::shutdown();
  return 0;
}
```

· A simple publisher and subscriber (C++)

The String message will be received in the callback function, and then print.

For the publisher node, spinning meant starting the timer, but for the subscriber it simply means preparing to receive messages whenever they come.

```cpp
#include <memory>
#include "rclcpp/rclcpp.hpp"
#include "std_msgs/msg/string.hpp"
using std::placeholders::_1;

class MinimalSubscriber : public rclcpp::Node
{
  public:
    MinimalSubscriber()
    : Node("minimal_subscriber")
    {
      subscription_ = this->create_subscription<std_msgs::msg::String>(
      "topic", 10, std::bind(&MinimalSubscriber::topic_callback, this, _1));
    }

  private:
    void topic_callback(const std_msgs::msg::String::SharedPtr msg) const
    {
      RCLCPP_INFO(this->get_logger(), "I heard: '%s'", msg->data.c_str());
    }
    rclcpp::Subscription<std_msgs::msg::String>::SharedPtr subscription_;
};

int main(int argc, char * argv[])
{
  rclcpp::init(argc, argv);
  rclcpp::spin(std::make_shared<MinimalSubscriber>());
  rclcpp::shutdown();
  return 0;
}
```

· A simple publisher and subscriber (C++)

Before compiling, make sure that the dependencies of the function package are all installed.

```
zhilin@zhilin:~/dev_ws$ rosdep install -i --from-path src --rosdistro eloquent -y
```

```
zhilin@zhilin:~/dev_ws 2$ colcon build --packages-select cpp_pubsub
Starting >>> cpp_pubsub
Finished <<< cpp_pubsub [6.88s]

Summary: 1 package finished [6.99s]
```

· A simple publisher and subscriber (C++)

```
zhilin@zhilin:~/dev_ws 2$ . install/setup.bash
zhilin@zhilin:~/dev_ws 2$ ros2 run cpp_pubsub listener
[INFO] [1649190492.902187731] [minimal_subscriber]: I heard: 'Hello, world! 17'
[INFO] [1649190493.402260577] [minimal_subscriber]: I heard: 'Hello, world! 18'
[INFO] [1649190493.902117126] [minimal_subscriber]: I heard: 'Hello, world! 19'
[INFO] [1649190494.402165529] [minimal_subscriber]: I heard: 'Hello, world! 20'
[INFO] [1649190494.902164623] [minimal_subscriber]: I heard: 'Hello, world! 21'
[INFO] [1649190495.402055045] [minimal_subscriber]: I heard: 'Hello, world! 22'
[INFO] [1649190495.902018591] [minimal_subscriber]: I heard: 'Hello, world! 23'
[INFO] [1649190496.402317443] [minimal_subscriber]: I heard: 'Hello, world! 24'
[INFO] [1649190496.902171896] [minimal_subscriber]: I heard: 'Hello, world! 25'
```

```
zhilin@zhilin:~/dev_ws 2$ ros2 run cpp_pubsub talker
[INFO] [1649190484.401848471] [minimal_publisher]: Publishing: 'Hello, world! 0'
[INFO] [1649190484.901967552] [minimal_publisher]: Publishing: 'Hello, world! 1'
[INFO] [1649190485.401704394] [minimal_publisher]: Publishing: 'Hello, world! 2'
[INFO] [1649190485.901700835] [minimal_publisher]: Publishing: 'Hello, world! 3'
[INFO] [1649190486.401885916] [minimal_publisher]: Publishing: 'Hello, world! 4'
[INFO] [1649190486.901638192] [minimal_publisher]: Publishing: 'Hello, world! 5'
[INFO] [1649190487.401647972] [minimal_publisher]: Publishing: 'Hello, world! 6'
[INFO] [1649190487.901710971] [minimal_publisher]: Publishing: 'Hello, world! 7'
[INFO] [1649190488.401637222] [minimal_publisher]: Publishing: 'Hello, world! 8'
[INFO] [1649190488.901664951] [minimal_publisher]: Publishing: 'Hello, world! 9'
[INFO] [1649190489.401627798] [minimal_publisher]: Publishing: 'Hello, world! 10'
[INFO] [1649190489.901622702] [minimal_publisher]: Publishing: 'Hello, world! 11'
```

The preparation before build & run

package.xml

- file containing meta information about the package

```cpp
#include <stdlib.h>
#include <functional>
#include <iostream>

#include "rclcpp/rclcpp.hpp"
#include <sensor_msgs/image_encodings.hpp>
#include <cv_bridge/cv_bridge.h>
#include <image_transport/image_transport.hpp>
#include <opencv2/objdetect/objdetect.hpp>
#include <opencv2/imgproc/imgproc.hpp>
#include <opencv2/highgui/highgui.hpp>
#include <opencv2/videoio/videoio.hpp>
```

ROS2 dependency library in C++ code

```xml
<?xml version="1.0"?>
<?xml-model
  href="http://download.ros.org/schema/package_format3.xsd"
  schematypens="http://www.w3.org/2001/XMLSchema"?>
<package format="3">
  <name>opencv_demos</name>
  <version>0.0.0</version>
  <description>TODO: Package description</description>
  <maintainer email="robotos@todo.todo">robotos</maintainer>
  <license>TODO: License declaration</license>

  <buildtool_depend>ament_cmake</buildtool_depend>

  <test_depend>ament_lint_auto</test_depend>
  <test_depend>ament_lint_common</test_depend>

  <depend>rclcpp</depend>
  <depend>image_transport</depend>
  <depend>sensor_msgs</depend>
  <depend>cv_bridge</depend>

  <export>
    <build_type>ament_cmake</build_type>
  </export>
</package>
```

package.xml file

CMakeList.txt

- file that describes how to build the code within the package

```
#include <stdlib.h>
#include <functional>
#include <iostream>

#include "rclcpp/rclcpp.hpp"
#include <sensor_msgs/image_encodings.hpp>
#include <cv_bridge/cv_bridge.h>
#include <image_transport/image_transport.hpp>
#include <opencv2/objdetect/objdetect.hpp>
#include <opencv2/imgproc/imgproc.hpp>
#include <opencv2/highgui/highgui.hpp>
#include <opencv2/videoio/videoio.hpp>
```

#include in C++ code

```
cmake_minimum_required(VERSION 3.5)
project(opencv_demos)
# Default to C++14
if(NOT CMAKE_CXX_STANDARD)
  set(CMAKE_CXX_STANDARD 14)
endif()
set(cv_bridge_DIR /usr/local/share/cv_bridge/cmake)
set(OpenCV_DIR /usr/include/opencv4)
# find dependencies
find_package(ament_cmake REQUIRED)
find_package(image_transport REQUIRED)
find_package(sensor_msgs REQUIRED)
find_package(cv_bridge REQUIRED)
find_package(OpenCV 4 REQUIRED
  COMPONENTS
  opencv_core
  opencv_imgproc
  opencv_highgui
  opencv_objdetect
  opencv_videoio
  opencv_video
  opencv_imgcodecs
  CONFIG
)
if(BUILD_TESTING)
  find_package(ament_lint_auto REQUIRED)
  ament_lint_auto_find_test_dependencies()
endif()
include_directories(${OpenCV_INCLUDE_DIRS})
add_executable(face_detector src/face_detector.cpp)
ament_target_dependencies(face_detector rclcpp image_transport sensor_msgs cv_bridge)
install(TARGETS
        face_detector
        DESTINATION lib/${PROJECT_NAME})
ament_package()
```

CmakeList.txt file

Launch File

- file that allows you to start up and configure
a number of executables containing ROS 2 nodes simultaneously.

```python
from launch import LaunchDescription
from launch_ros.actions import Node

def generate_launch_description():
    return LaunchDescription([
        Node(
            package='opencv_demos',
            executable='face_detector',
            name='face_detector',
            remappings=[
                ('in_image_base_topic', '/image_raw'),
                ('out_image_base_topic', '/face_detector/out_image_base_topic'),
            ]
        )
    ])
```

Main C++ file

Details of face_detector.cpp

- Face detection node based on opencv Haar cascade classifier (C++)

```cpp
//import header file
#include <stdlib.h>
#include <functional>
#include <iostream>
#include "rclcpp/rclcpp.hpp"
#include <sensor_msgs/image_encodings.hpp>
#include <cv_bridge/cv_bridge.h>
#include <image_transport/image_transport.hpp>
#include <opencv2/objdetect/objdetect.hpp>
#include <opencv2/imgproc/imgproc.hpp>
#include <opencv2/highgui/highgui.hpp>
#include <opencv2/videoio/videoio.hpp>
```

```cpp
class ImageConverter : public rclcpp::Node     // This line creates the node class ImageConverter by inheriting from rclcpp::Node.
{
    private:
        image_transport::Subscriber sub_;
        image_transport::Publisher pub_;
    public:
        cv::CascadeClassifier face_cascade;    //Load  detector: apply CascadeClassifier and instantiate, write file path
        cv::CascadeClassifier eyes_cascade;
        cv::String face_cascade_name = "/home/zhilin/DEV/test/src/ros2_opencv_demos/src/haarcascade_frontalface_default.xml";
        cv::String eyes_cascade_name = "/home/zhilin/DEV/test/src/ros2_opencv_demos/src/haarcascade_eye.xml";
        ImageConverter() : Node("image_converter") {
            //-- 1. Load the cascades
            if( !face_cascade.load( face_cascade_name ) )
            {   std::cout << "--(!)Error loading face cascade\n"; };
            if( !eyes_cascade.load( eyes_cascade_name ) )
            {   std::cout << "--(!)Error loading eyes cascade\n"; };
            rmw_qos_profile_t custom_qos = rmw_qos_profile_default;
            // use image_transport  to subscribe to and publish images.
            sub_ = image_transport::create_subscription(this, "in_image_base_topic",
                        std::bind(&ImageConverter::imageCallback, this, std::placeholders::_1), "raw", custom_qos);
                        //The callback function of sub is imageCallback
            pub_ = image_transport::create_publisher(this, "out_image_base_topic", custom_qos);
        }
    //  Declaration of the imageCallback function
    void imageCallback(const sensor_msgs::msg::Image::ConstSharedPtr &msg);
    //  Declaration of the faceDetection function
    void faceDetection( cv::Mat& frame );
};


void ImageConverter::imageCallback(const sensor_msgs::msg::Image::ConstSharedPtr &msg)
{…
}
void ImageConverter::faceDetection( cv::Mat& frame )
{…
}

int main(int argc, char** argv)
{
    //  initializes ROS 2
    rclcpp::init(argc, argv);
    //  rclcpp::spin starts processing data from the node
    rclcpp::spin(std::make_shared<ImageConverter>());
    rclcpp::shutdown();
    return 0;
}
```

Details of face_detector.cpp

- Face detection node based on opencv Haar cascade classifier (C++)

```cpp
void ImageConverter::imageCallback(const sensor_msgs::msg::Image::ConstSharedPtr &msg)
{
    cv_bridge::CvImagePtr cv_ptr;
    try
    {
        //Convert ROS image to OpenCV image (C++)
        cv_ptr = cv_bridge::toCvCopy(msg, msg->encoding);
    }
    catch (cv_bridge::Exception& e)
    {
        RCLCPP_ERROR(this->get_logger(), "cv_bridge exception: %s", e.what());
        return;
    }
        // OpenCV Image Processing
    faceDetection(cv_ptr->image);
    cv::waitKey(3);

    pub_.publish(cv_ptr->toImageMsg());
}
```

```cpp
void ImageConverter::faceDetection( cv::Mat& frame )
{
    //using Haar cascade classifier to detect human face and eyes
    cv::Mat frame_gray;
    cv::cvtColor( frame, frame_gray, cv::COLOR_BGR2GRAY );
    cv::equalizeHist( frame_gray, frame_gray );
    //-- Detect faces
    std::vector<cv::Rect> faces;
    face_cascade.detectMultiScale( frame_gray, faces );
    for ( size_t i = 0; i < faces.size(); i++ )
    {
        cv::Point center( faces[i].x + faces[i].width/2, faces[i].y + faces[i].height/2 );
        cv::ellipse( frame, center, cv::Size( faces[i].width/2, faces[i].height/2 ), 0, 0, 360, cv::Scalar( 255, 0, 255 ), 4 );
        cv::Mat faceROI = frame_gray( faces[i] );
        //-- In each face, detect eyes
        std::vector<cv::Rect> eyes;
        eyes_cascade.detectMultiScale( faceROI, eyes );
        for ( size_t j = 0; j < eyes.size(); j++ )
        {
            cv::Point eye_center( faces[i].x + eyes[j].x + eyes[j].width/2, faces[i].y + eyes[j].y + eyes[j].height/2 );
            int radius = cvRound( (eyes[j].width + eyes[j].height)*0.25 );
            circle( frame, eye_center, radius, cv::Scalar( 255, 0, 0 ), 4 );
        }
    }
}
```

UNIVERSITY
OF TURKU