

# EMC 2014 C++ and ROS

Sjoerd van den Dries

Eindhoven University of Technology  
Department of Mechanical Engineering



May 1, 2014

**TU** / **e** Technische Universiteit  
Eindhoven  
University of Technology

Where innovation starts

- ▶ We will use C++ as programming language
- ▶ One of the two core ROS languages
  - Packages `roscpp` and `roslib`
- ▶ C++ is object-oriented C
  - “C with Classes”
  - Encapsulate data and functionality within objects
- ▶ Many tutorials available, e.g.:
  - <http://www.cplusplus.com/doc/tutorial>
  - MIT's Introduction to C++

- ▶ C++ is a **compiled language**

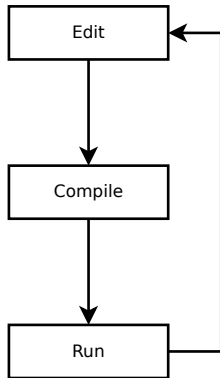
- ▶ C++ is a **compiled language**
- ▶ You write **source code**
  - A set of '*human-readable*' instructions

- ▶ C++ is a **compiled language**
- ▶ You write **source code**
  - A set of '*human-readable*' instructions
- ▶ which is **compiled** by a **compiler** into

- ▶ C++ is a **compiled language**
- ▶ You write **source code**
  - A set of '*human-readable*' instructions
- ▶ which is **compiled** by a **compiler** into
- ▶ **binary code**
  - A set of '*machine-readable*' instructions

- ▶ C++ is a **compiled language**
- ▶ You write **source code**
  - A set of '*human-readable*' instructions
- ▶ which is **compiled** by a **compiler** into
- ▶ **binary code**
  - A set of '*machine-readable*' instructions
- ▶ Advantage: **fast at run-time**
- ▶ Disadvantage: **slow at design-time**

- ▶ C++ is a **compiled language**
- ▶ You write **source code**
  - A set of '*human-readable*' instructions
- ▶ which is **compiled** by a **compiler** into
- ▶ **binary code**
  - A set of '*machine-readable*' instructions
- ▶ Advantage: **fast at run-time**
- ▶ Disadvantage: **slow at design-time**





hello.cpp

---

```
#include <iostream>

int main() {
    std::cout << "Hello World" << std::endl;
    return 0;
}
```

---

hello.cpp

---

```
#include <iostream>

int main() {
    std::cout << "Hello World" << std::endl;
    return 0;
}
```

---

- ▶ **Compile:** g++ hello.cpp -o hello

hello.cpp

---

```
#include <iostream>

int main() {
    std::cout << "Hello World" << std::endl;
    return 0;
}
```

---

- ▶ **Compile:** `g++ hello.cpp -o hello`
- ▶ **Run:** `./hello`

hello.cpp

---

```
#include <iostream>

int main() {
    std::cout << "Hello World" << std::endl;
    return 0;
}
```

---

- ▶ **Compile:** `g++ hello.cpp -o hello`
- ▶ **Run:** `./hello`

Hello World

hello.cpp

---

```
#include <iostream>

int main() {
    std::cout << "Hello World" << std::endl;
    return 0;
}
```

---

hello.cpp

---

```
#include <iostream>

int main() {
    std::cout << "Hello World" << std::endl;
    return 0;
}
```

---

- ▶ Execution starts with `main()`

hello.cpp

---

```
#include <iostream>

int main() {
    std::cout << "Hello World" << std::endl;
    return 0;
}
```

---

- ▶ Execution starts with `main()`
- ▶ Executes instructions one by one
  - Each separated by a `semi-colon ;`

hello.cpp

---

```
#include <iostream>

int main() {
    std::cout << "Hello World" << std::endl;
    return 0;
}
```

---

- ▶ Execution starts with `main()`
- ▶ Executes instructions one by one
  - Each separated by a `semi-colon ;`



hello.cpp

---

```
#include <iostream>

int main() {
    std::cout << "Hello World" << std::endl;
    return 0;
}
```

---

- ▶ Execution starts with `main()`
- ▶ Executes instructions one by one
  - Each separated by a `semi-colon ;`

hello.cpp

---

```
#include <iostream>

int main() {
    std::cout << "Hello World" << std::endl;
    return 0;
}
```

---

- ▶ Execution starts with `main()`
- ▶ Executes instructions one by one
  - Each separated by a `semi-colon ;`
- ▶ Stops after last instruction

hello.cpp

---

```
#include <iostream>

int main() {
    std::cout << "Hello World" << std::endl;
    return 0;
}
```

---

- ▶ Execution starts with `main()`
- ▶ Executes instructions one by one
  - Each separated by a `semi-colon ;`
- ▶ Stops after last instruction
- ▶ **Flow of control**

## hello.cpp

---

```
#include <iostream>

int main() {
    std::cout << "Hello World" << std::endl;
    return 0;
}
```

---

hello.cpp

---

```
#include <iostream>

int main() {
    std::cout << "Hello World" << std::endl;
    return 0;
}
```

---

- ▶ **#include** includes other libraries
  - **iostream** defines **std::cout** and **std::endl**

## hello.cpp

---

```
#include <iostream>

int main() {
    std::cout << "Hello World" << std::endl;
    return 0;
}
```

---

- ▶ **#include** includes other libraries
  - **iostream** defines **std::cout** and **std::endl**
  - **#include** in fact just **copies** the code into your program

## hello.cpp

---

```
#include <iostream>

int main() {
    std::cout << "Hello World" << std::endl;
    return 0;
}
```

---

- ▶ `#include` includes other libraries
  - `iostream` defines `std::cout` and `std::endl`
  - `#include` in fact just **copies** the code into your program
- ▶ `std::cout << ... << std::endl` prints to screen

## hello.cpp

---

```
#include <iostream>

int main() {
    std::cout << "Hello World" << std::endl;
    return 0;
}
```

---

- ▶ **#include** includes other libraries
  - **iostream** defines **std::cout** and **std::endl**
  - **#include** in fact just **copies** the code into your program
- ▶ **std::cout << ... << std::endl** prints to screen
- ▶ **main()** must return an **error code**
  - **0** means **no errors**





## ► Expression

- A statement that has a **value**
- For example:
  - $3 * 5 + 7$
- is an expressions that **evaluates to**:
  - 22

## ▶ Expression

- A statement that has a **value**
- For example:
  - $3 * 5 + 7$
- is an expressions that **evaluates to**:
  - 22

## ▶ Variable

- A **name** for a **value** such that we can refer to it later

## ▶ Expression

- A statement that has a **value**
- For example:
  - $3 * 5 + 7$
- is an expressions that **evaluates to**:
  - 22

## ▶ Variable

- A **name** for a **value** such that we can refer to it later
- Variables have a **name**, a **type** and a **value**

## ▶ Expression

- A statement that has a **value**
- For example:
  - $3 * 5 + 7$
- is an expressions that **evaluates to**:
  - 22

## ▶ Variable

- A **name** for a **value** such that we can refer to it later
- Variables have a **name**, a **type** and a **value**
- `int i = 3 * 5 + 7`

---

```
#include <iostream>

int main() {
    int i;
    i = 3;
    int j = i * i;
    std::cout << i << "^2 = " << j << std::endl;
    return 0;
}
```

---

```
#include <iostream>

int main() {
    int i;
    i = 3;
    int j = i * i;
    std::cout << i << "^2 = " << j << std::endl;
    return 0;
}
```

- ▶ **Declare** variable with name **i** and type **int** (integer)

```
#include <iostream>

int main() {
    int i;
    i = 3;
    int j = i * i;
    std::cout << i << "^2 = " << j << std::endl;
    return 0;
}
```

- ▶ **Declare** variable with name **i** and type **int** (integer)
- ▶ **Assign** to **i** the value **3**



```
#include <iostream>

int main() {
    int i;
    i = 3;
    int j = i * i;
    std::cout << i << "^2 = " << j << std::endl;
    return 0;
}
```

- ▶ **Declare** variable with name **i** and type **int** (integer)
- ▶ **Assign** to **i** the value **3**
- ▶ **Declare** variable **j**, **evaluate**  $i \cdot i$  and **assign** the outcome to **j**

```
#include <iostream>

int main() {
    int i;
    i = 3;
    int j = i * i;
    std::cout << i << "^2 = " << j << std::endl;
    return 0;
}
```

- ▶ **Declare** variable with name **i** and type **int** (integer)
- ▶ **Assign** to **i** the value **3**
- ▶ **Declare** variable **j**, **evaluate**  $i \cdot i$  and **assign** the outcome to **j**
- ▶ **Outcome:**  $3^2 = 9$



- ▶ Instructions are executed **one-by-one**

- ▶ Instructions are executed **one-by-one**
- ▶ However, we can **decide** to **only** execute some instructions if a **condition** is met
  - **condition** = **boolean expression**
  - *something that evaluates to true or false*

- ▶ Instructions are executed **one-by-one**
- ▶ However, we can **decide** to **only** execute some instructions if a **condition** is met
  - **condition** = **boolean expression**
  - *something that evaluates to true or false*
  - `i == 3`
  - `i < 7`
  - `i >= j`

- ▶ Instructions are executed **one-by-one**
- ▶ However, we can **decide** to **only** execute some instructions if a **condition** is met
  - **condition** = **boolean expression**
  - *something that evaluates to true or false*
  - `i == 3`
  - `i < 7`
  - `i >= j`

---

```
if (CONDITION) {  
    // only happens if CONDITION is true  
} else {  
    // only happens if CONDITION is false  
}
```

---

---

```
#include <iostream>

int main() {
    int i = 7;
    if (i > 5) {
        std::cout << "Bigger than 5" << std::endl;
    } else {
        std::cout << "5 or smaller" << std::endl;
    }
    return 0;
}
```

---



---

```
#include <iostream>

int main() {
    int i = 7;
    if (i > 5) {
        std::cout << "Bigger than 5" << std::endl;
    } else {
        std::cout << "5 or smaller" << std::endl;
    }
    return 0;
}
```

---

---

```
#include <iostream>

int main() {
    int i = 7;
    if (i > 5) {
        std::cout << "Bigger than 5" << std::endl;
    } else {
        std::cout << "5 or smaller" << std::endl;
    }
    return 0;
}
```

---

---

```
#include <iostream>

int main() {
    int i = 7;
    if (i > 5) {
        std::cout << "Bigger than 5" << std::endl;
    } else {
        std::cout << "5 or smaller" << std::endl;
    }
    return 0;
}
```

---

---

```
#include <iostream>

int main() {
    int i = 7;
    if (i > 5) {
        std::cout << "Bigger than 5" << std::endl;
    } else {
        std::cout << "5 or smaller" << std::endl;
    }
    return 0;
}
```

---



- ▶ **Loops:**
  - Can be used to **repeat** a set of instructions

- ▶ **Loops:**
  - Can be used to **repeat** a set of instructions
  - **while** some condition is true

## ▶ Loops:

- Can be used to **repeat** a set of instructions
- **while** some condition is true

---

```
#include <iostream>

int main() {
    int i = 0;
    while (i < 10) {
        std::cout << i << std::endl;
        i++; /* This means: i = i + 1 */
    }
    return 0;
}
```

---



## ▶ Loops:

- Can be used to **repeat** a set of instructions
- **while** some condition is true

---

```
#include <iostream>

int main() {
    int i = 0;
    while (i < 10) {
        std::cout << i << std::endl;
        i++; /* This means: i = i + 1 */
    }
    return 0;
}
```

---

## ▶ Loops:

- Can be used to **repeat** a set of instructions
- **while** some condition is true

---

```
#include <iostream>

int main() {
    int i = 0;
    while (i < 10) {
        std::cout << i << std::endl;
        i++; /* This means: i = i + 1 */
    }
    return 0;
}
```

---

## ▶ Loops:

- Can be used to **repeat** a set of instructions
- **while** some condition is true

---

```
#include <iostream>

int main() {
    int i = 0;
    while (i < 10) {
        std::cout << i << std::endl;
        i++; /* This means: i = i + 1 */
    }
    return 0;
}
```

---

## ▶ Loops:

- Can be used to **repeat** a set of instructions
- **while** some condition is true

---

```
#include <iostream>

int main() {
    int i = 0;
    while (i < 10) {
        std::cout << i << std::endl;
        i++; /* This means: i = i + 1 */
    }
    return 0;
}
```

---

## ▶ Loops:

- Can be used to **repeat** a set of instructions
- **while** some condition is true

---

```
#include <iostream>

int main() {
    int i = 0;
    while (i < 10) {
        std::cout << i << std::endl;
        i++; /* This means: i = i + 1 */
    }
    return 0;
}
```

---

## ▶ Loops:

- Can be used to **repeat** a set of instructions
- **while** some condition is true

---

```
#include <iostream>

int main() {
    int i = 0;
    while (i < 10) {
        std::cout << i << std::endl;
        i++; /* This means: i = i + 1 */
    }
    return 0;
}
```

---



- ▶ For-loops:
  - More advanced loop control



- ▶ For-loops:
  - More advanced loop control
  - Init, condition and increment in 'one line'

- ▶ For-loops:
  - More advanced loop control
  - Init, condition and increment in 'one line'
  - `for(INIT; CONDITION; INCREMENT)`

► For-loops:

- More advanced loop control
- Init, condition and increment in 'one line'
- for(INIT; CONDITION; INCREMENT)

---

```
#include <iostream>

int main() {
    for (int i = 0; i < 10; i++) {
        std::cout << i << std::endl;
    }
    return 0;
}
```

---

► For-loops:

- More advanced loop control
- Init, condition and increment in 'one line'
- for(INIT; CONDITION; INCREMENT)

---

```
#include <iostream>

int main() {
    for (int i = 0; i < 10; i++) {
        std::cout << i << std::endl;
    }
    return 0;
}
```

---

► For-loops:

- More advanced loop control
- Init, condition and increment in 'one line'
- for(INIT; CONDITION; INCREMENT)

---

```
#include <iostream>

int main() {
    for (int i = 0; i < 10; i++) {
        std::cout << i << std::endl;
    }
    return 0;
}
```

---

► For-loops:

- More advanced loop control
- Init, condition and increment in 'one line'
- for(INIT; CONDITION; INCREMENT)

---

```
#include <iostream>

int main() {
    for (int i = 0; i < 10; i++) {
        std::cout << i << std::endl;
    }
    return 0;
}
```

---

► For-loops:

- More advanced loop control
- Init, condition and increment in 'one line'
- for(INIT; CONDITION; INCREMENT)

---

```
#include <iostream>

int main() {
    for (int i = 0; i < 10; i++) {
        std::cout << i << std::endl;
    }
    return 0;
}
```

---

► For-loops:

- More advanced loop control
- Init, condition and increment in 'one line'
- for(INIT; CONDITION; INCREMENT)

---

```
#include <iostream>

int main() {
    for (int i = 0; i < 10; i++) {
        std::cout << i << std::endl;
    }
    return 0;
}
```

---



---

```
#include <iostream>

void niceFunction() {
    int i = 3;
    std::cout << i << "^2 = " << i * i << std::endl;
}

int main() {
    niceFunction();
    return 0;
}
```

---

```
#include <iostream>

void niceFunction() {
    int i = 3;
    std::cout << i << "^2 = " << i * i << std::endl;
}

int main() {
    niceFunction();
    return 0;
}
```

- ▶ Define a function with name `niceFunction`

```
#include <iostream>

void niceFunction() {
    int i = 3;
    std::cout << i << "^2 = " << i * i << std::endl;
}

int main() {
    niceFunction();
    return 0;
}
```

- ▶ Define a function with name `niceFunction`
- ▶ The `body { ... }` of the function tells what the function *'does'*

```
#include <iostream>

void niceFunction() {
    int i = 3;
    std::cout << i << "^2 = " << i * i << std::endl;
}

int main() {
    niceFunction();
    return 0;
}
```

- ▶ Define a function with name `niceFunction`
- ▶ The `body { ... }` of the function tells what the function *'does'*
- ▶ The function can be `called`

```
#include <iostream>

void niceFunction() {
    int i = 3;
    std::cout << i << "^2 = " << i * i << std::endl;
}

int main() {
    niceFunction();
    return 0;
}
```

- ▶ Define a function with name `niceFunction`
- ▶ The `body { ... }` of the function tells what the function *'does'*
- ▶ The function can be `called`
- ▶ Flow of control ...

```
#include <iostream>

void niceFunction() {
    int i = 3;
    std::cout << i << "^2 = " << i * i << std::endl;
}

int main() {
    niceFunction();
    return 0;
}
```

- ▶ Define a function with name `niceFunction`
- ▶ The `body { ... }` of the function tells what the function *'does'*
- ▶ The function can be `called`
- ▶ Flow of control ...

```
#include <iostream>

void niceFunction() {
    int i = 3;
    std::cout << i << "^2 = " << i * i << std::endl;
}

int main() {
    niceFunction();
    return 0;
}
```

- ▶ Define a function with name `niceFunction`
- ▶ The `body { ... }` of the function tells what the function *'does'*
- ▶ The function can be `called`
- ▶ Flow of control ...

```
#include <iostream>

void niceFunction() {
    int i = 3;
    std::cout << i << "^2 = " << i * i << std::endl;
}

int main() {
    niceFunction();
    return 0;
}
```

- ▶ Define a function with name `niceFunction`
- ▶ The `body { ... }` of the function tells what the function *'does'*
- ▶ The function can be `called`
- ▶ Flow of control ...



```
#include <iostream>

void niceFunction() {
    int i = 3;
    std::cout << i << "^2 = " << i * i << std::endl;
}

int main() {
    niceFunction();
    return 0;
}
```

- ▶ Define a function with name `niceFunction`
- ▶ The `body { ... }` of the function tells what the function *'does'*
- ▶ The function can be `called`
- ▶ Flow of control ...

```
#include <iostream>

void niceFunction() {
    int i = 3;
    std::cout << i << "^2 = " << i * i << std::endl;
}

int main() {
    niceFunction();
    return 0;
}
```

- ▶ Define a function with name `niceFunction`
- ▶ The `body { ... }` of the function tells what the function *'does'*
- ▶ The function can be `called`
- ▶ Flow of control ...

```
#include <iostream>

void niceFunction() {
    int i = 3;
    std::cout << i << "^2 = " << i * i << std::endl;
}

int main() {
    niceFunction();
    return 0;
}
```

- ▶ Define a function with name `niceFunction`
- ▶ The `body { ... }` of the function tells what the function *'does'*
- ▶ The function can be `called`
- ▶ Flow of control ...

```
#include <iostream>

int niceFunction(int x, int y) {
    return x * x + y * y;
}

int main() {
    int i = 3;
    int j = niceFunction(i, 4);
    std::cout << j << std::endl;
    return 0;
}
```

- ▶ A function can also take **input** and return **output**:

```
#include <iostream>

int niceFunction(int x, int y) {
    return x * x + y * y;
}

int main() {
    int i = 3;
    int j = niceFunction(i, 4);
    std::cout << j << std::endl;
    return 0;
}
```

- ▶ A function can also take **input** and return **output**:

```
#include <iostream>

int niceFunction(int x, int y) {
    return x * x + y * y;
}

int main() {
    int i = 3;
    int j = niceFunction(i, 4);
    std::cout << j << std::endl;
    return 0;
}
```

- ▶ A function can also take **input** and return **output**:
  - Input arguments

```
#include <iostream>

int niceFunction(int x, int y) {
    return x * x + y * y;
}

int main() {
    int i = 3;
    int j = niceFunction(i, 4);
    std::cout << j << std::endl;
    return 0;
}
```

- ▶ A function can also take **input** and return **output**:
  - Input arguments
  - Return value

```
#include <iostream>

int niceFunction(int x, int y) {
    return x * x + y * y;
}

int main() {
    int i = 3;
    int j = niceFunction(i, 4);
    std::cout << j << std::endl;
    return 0;
}
```

- ▶ A function can also take **input** and return **output**:
  - Input arguments
  - Return value
  - Return type



- ▶ So far, we have seen:
  - How to **write**, **compile** and **run** a program
  - **Variables**
  - **Conditionals**
  - **Loops**
  - **Functions**

- ▶ So far, we have seen:
  - How to **write**, **compile** and **run** a program
  - **Variables**
  - **Conditionals**
  - **Loops**
  - **Functions**
  
- ▶ Enough to write a **procedural** program

- ▶ So far, we have seen:
  - How to **write**, **compile** and **run** a program
  - **Variables**
  - **Conditionals**
  - **Loops**
  - **Functions**
  
- ▶ Enough to write a **procedural** program
  - Step-by-step instructions
  - **re-use of code**
  - Advanced **control flow**

C++ has many **built-in** data types:

`int`

Integer

$\{\dots, -2, -1, 0, 1, \dots\}$

C++ has many **built-in** data types:

<code>int</code>	Integer	$\{\dots, -2, -1, 0, 1, \dots\}$
<code>unsigned int</code>	Positive integer	$\{0, 1, \dots\}$

C++ has many **built-in** data types:

<code>int</code>	Integer	$\{\dots, -2, -1, 0, 1, \dots\}$
<code>unsigned int</code>	Positive integer	$\{0, 1, \dots\}$
<code>bool</code>	Boolean	0 or 1

C++ has many **built-in** data types:

<code>int</code>	Integer	{..., -2, -1, 0, 1, ...}
<code>unsigned int</code>	Positive integer	{0, 1, ...}
<code>bool</code>	Boolean	0 or 1
<code>float</code>	Floating point	<i>e.g.</i> , -2.6, 1.0, 3.1415

C++ has many **built-in** data types:

<code>int</code>	Integer	{..., -2, -1, 0, 1, ...}
<code>unsigned int</code>	Positive integer	{0, 1, ...}
<code>bool</code>	Boolean	0 or 1
<code>float</code>	Floating point	<i>e.g.</i> , -2.6, 1.0, 3.1415
<code>double</code>	Double floating point	



C++ has many **built-in** data types:

<code>int</code>	Integer	{..., -2, -1, 0, 1, ...}
<code>unsigned int</code>	Positive integer	{0, 1, ...}
<code>bool</code>	Boolean	0 or 1
<code>float</code>	Floating point	<i>e.g.</i> , -2.6, 1.0, 3.1415
<code>double</code>	Double floating point	
<code>char</code>	Character	<i>e.g.</i> , 'a', 'B', '?'

C++ has many **built-in** data types:

<b>int</b>	Integer	{..., -2, -1, 0, 1, ...}
<b>unsigned int</b>	Positive integer	{0, 1, ...}
<b>bool</b>	Boolean	0 or 1
<b>float</b>	Floating point	<i>e.g.</i> , -2.6, 1.0, 3.1415
<b>double</b>	Double floating point	
<b>char</b>	Character	<i>e.g.</i> , 'a', 'B', '?'
<b>unsigned char</b>	Byte	{0, ..., 255}

- ▶ Built-in data types are nice, but limited

- ▶ Built-in data types are nice, but limited
  - Let's say we want to determine PICO's nearest laser point

- ▶ Built-in data types are nice, but limited
  - Let's say we want to determine PICO's nearest laser point
  - Such point can be represented with two variables:
    - double x
    - double y

- ▶ Built-in data types are nice, but limited
  - Let's say we want to determine PICO's nearest laser point
  - Such point can be represented with two variables:
    - double x
    - double y
  - Cumbersome to pass around

- ▶ Built-in data types are nice, but limited
  - Let's say we want to determine PICO's nearest laser point
  - Such point can be represented with two variables:
    - double x
    - double y
  - Cumbersome to pass around
  - Not very clear we're talking about a point (semantics)

- ▶ Built-in data types are nice, but limited
  - Let's say we want to determine PICO's nearest laser point
  - Such point can be represented with two variables:
    - double x
    - double y
  - Cumbersome to pass around
  - Not very clear we're talking about a point (semantics)
  - How to return both values from a function?



- ▶ **Built-in data types** are nice, but **limited**
  - Let's say we want to determine PICO's nearest **laser point**
  - Such **point** can be represented with two **variables**:
    - `double x`
    - `double y`
  - Cumbersome to **pass around**
  - Not very clear we're talking about a point (**semantics**)
  - How to **return** both values from a **function**?

---

```
...? determineNearestPoint() {  
    // Some nice code  
    double x = ...  
    double y = ...  
    return ...?  
}
```

---

- ▶ Solution: create your own **data structure**:

- ▶ Solution: create your own **data structure**:
  - “a group of data elements grouped together under one name”

- ▶ Solution: create your own **data structure**:
  - “a group of data elements grouped together under one name”
  - Are called **structs** in C++

- ▶ Solution: create your own **data structure**:
  - “a group of data elements grouped together under one name”
  - Are called **structs** in C++

---

```
struct Point { // This starts the definition of a struct
    double x; // This declares x as member of Point
    double y; // This declares y as member of Point
}
```

---

- ▶ Solution: create your own **data structure**:
  - “a group of data elements grouped together under one name”
  - Are called **structs** in C++

---

```
struct Point { // This starts the definition of a struct
    double x; // This declares x as member of Point
    double y; // This declares y as member of Point
}
```

---

- ▶ Now we can use **Point** as **data type**:

- ▶ Solution: create your own **data structure**:
  - “a group of data elements grouped together under one name”
  - Are called **structs** in C++

---

```
struct Point { // This starts the definition of a struct
    double x; // This declares x as member of Point
    double y; // This declares y as member of Point
}
```

---

- ▶ Now we can use **Point** as **data type**:

---

```
Point determineNearestPoint() {
    // Some nice code
    Point p; // declare variable p of type Point
    p.x = ... // set x member of p
    p.y = ... // set y member of p
    return p;
}
```

---

- ▶ Solution: create your own **data structure**:
  - “a group of data elements grouped together under one name”
  - Are called **structs** in C++

---

```
struct Point { // This starts the definition of a struct
    double x; // This declares x as member of Point
    double y; // This declares y as member of Point
}
```

---

- ▶ Now we can use **Point** as **data type**:

---

```
void main() {
    Point p = determineNearestPoint();
    std::cout << "x = " << p.x << std::endl;
    std::cout << "y = " << p.y << std::endl;
}
```

---



- ▶ We can even use **structs** as **member types**
  - This way, we can build **more complex data structures**:

- ▶ We can even use **structs** as **member types**
  - This way, we can build **more complex data structures**:

---

```
struct Line {  
    Point p1;    // p1 is member of Line with type Point  
    Point p2;    // p2 is member of Line with type Point  
}
```

---

- ▶ We can even use **structs** as **member types**
  - This way, we can build **more complex data structures**:

---

```
struct Line {  
    Point p1;    // p1 is member of Line with type Point  
    Point p2;    // p2 is member of Line with type Point  
}
```

---

```
// ...  
Line l;  
l.p1.x = 3.0;  
l.p1.y = 4.5;  
l.p2.x = 6.0;  
l.p2.y = 10.3;  
  
Point q = l.p2;  
  
std::cout << q.x << std::endl;  
// ...
```

---

- ▶ Sometimes we want to store an arbitrary number of values together

- ▶ Sometimes we want to store an arbitrary number of values together
- ▶ Solution: use the `std::vector` data structure

- ▶ Sometimes we want to store an arbitrary number of values together
- ▶ Solution: use the `std::vector` data structure
  - Can hold any type of data, but

- ▶ Sometimes we want to store an arbitrary number of values together
- ▶ Solution: use the `std::vector` data structure
  - Can hold any type of data, but
  - all elements in the vector must have the same type

- ▶ Sometimes we want to store an arbitrary number of values together
- ▶ Solution: use the `std::vector` data structure
  - Can hold any type of data, but
  - all elements in the vector must have the same type

---

```
#include <iostream>
#include <vector>

int main() {
    std::vector<int> v; /* Declare a vector that holds ints */
    v.push_back(10); /* Add '10' to the back of the vector */
    v.push_back(20); /* Add '20' to the back */
    v.push_back(30); /* Add '30' */

    for(unsigned int i = 0; i < v.size(); i++) {
        std::cout << v[i] << std::endl; /* Print i'th element */
    }
}
```



- ▶ Sometimes we want to store an arbitrary number of values together
- ▶ Solution: use the `std::vector` data structure
  - Can hold any type of data, but
  - all elements in the vector must have the same type

---

```
#include <iostream>
#include <vector>

int main() {
    std::vector<int> v; /* Declare a vector that holds ints */
    v.push_back(10); /* Add '10' to the back of the vector */
    v.push_back(20); /* Add '20' to the back */
    v.push_back(30); /* Add '30' */

    for(unsigned int i = 0; i < v.size(); i++) {
        std::cout << v[i] << std::endl; /* Print i'th element */
    }
}
```

- ▶ Sometimes we want to store an arbitrary number of values together
- ▶ Solution: use the `std::vector` data structure
  - Can hold any type of data, but
  - all elements in the vector must have the same type

---

```
#include <iostream>
#include <vector>

int main() {
    std::vector<int> v; /* Declare a vector that holds ints */
    v.push_back(10); /* Add '10' to the back of the vector */
    v.push_back(20); /* Add '20' to the back */
    v.push_back(30); /* Add '30' */

    for(unsigned int i = 0; i < v.size(); i++) {
        std::cout << v[i] << std::endl; /* Print i'th element */
    }
}
```

- ▶ Sometimes we want to store an arbitrary number of values together
- ▶ Solution: use the `std::vector` data structure
  - Can hold any type of data, but
  - all elements in the vector must have the same type

---

```
#include <iostream>
#include <vector>

int main() {
    std::vector<int> v; /* Declare a vector that holds ints */
    v.push_back(10);   /* Add '10' to the back of the vector */
    v.push_back(20);   /* Add '20' to the back */
    v.push_back(30);   /* Add '30' */

    for(unsigned int i = 0; i < v.size(); i++) {
        std::cout << v[i] << std::endl; /* Print i'th element */
    }
}
```

- ▶ Sometimes we want to **store** an **arbitrary** number of values **together**
- ▶ Solution: use the **std::vector** data structure
  - Can hold **any type of data**, but
  - all elements in the vector must have **the same type**

---

```
#include <iostream>
#include <vector>

int main() {
    std::vector<int> v; /* Declare a vector that holds ints */
    v.push_back(10); /* Add '10' to the back of the vector */
    v.push_back(20); /* Add '20' to the back */
    v.push_back(30); /* Add '30' */

    for(unsigned int i = 0; i < v.size(); i++) {
        std::cout << v[i] << std::endl; /* Print i'th element */
    }
}
```

- ▶ Sometimes we want to store an arbitrary number of values together
- ▶ Solution: use the `std::vector` data structure
  - Can hold any type of data, but
  - all elements in the vector must have the same type

---

```
#include <iostream>
#include <vector>

int main() {
    std::vector<int> v; /* Declare a vector that holds ints */
    v.push_back(10); /* Add '10' to the back of the vector */
    v.push_back(20); /* Add '20' to the back */
    v.push_back(30); /* Add '30' */

    for(unsigned int i = 0; i < v.size(); i++) {
        std::cout << v[i] << std::endl; /* Print i'th element */
    }
}
```

- ▶ Sometimes we want to store an arbitrary number of values together
- ▶ Solution: use the `std::vector` data structure
  - Can hold any type of data, but
  - all elements in the vector must have the same type

---

```
#include <iostream>
#include <vector>

int main() {
    std::vector<int> v; /* Declare a vector that holds ints */
    v.push_back(10); /* Add '10' to the back of the vector */
    v.push_back(20); /* Add '20' to the back */
    v.push_back(30); /* Add '30' */

    for(unsigned int i = 0; i < v.size(); i++) {
        std::cout << v[i] << std::endl; /* Print i'th element */
    }
}
```

- ▶ `std::vector` is a part of the C++ Standard Library

- ▶ `std::vector` is a part of the C++ Standard Library
- ▶ This library contains many useful `data structures`, `functions`, etc...



- ▶ `std::vector` is a part of the C++ Standard Library
- ▶ This library contains many useful `data structures`, `functions`, etc. . .
- ▶ Uses the `namespace std`:
  - `std::vector` means `vector` in the namespace `std`

- ▶ `std::vector` is a part of the C++ Standard Library
- ▶ This library contains many useful `data structures`, `functions`, etc...
- ▶ Uses the `namespace std`:
  - `std::vector` means `vector` in the namespace `std`
  - `std::cout`
  - `std::endl`
  - ...

- ▶ `std::vector` is a part of the C++ Standard Library
- ▶ This library contains many useful `data structures`, `functions`, etc...
- ▶ Uses the `namespace std`:
  - `std::vector` means `vector` in the namespace `std`
  - `std::cout`
  - `std::endl`
  - ...
- ▶ <http://www.cplusplus.com/reference/>

- ▶ std::string

- ▶ `std::string`
  - Represents text:
    - A sequence (`string`) of `characters`

- ▶ `std::string`
  - Represents text:
    - A sequence (`string`) of `characters`

---

```
#include <iostream>
#include <string>

int main() {
    std::string s1 = "Hello";
    std::string s2 = "World";
    std::string s3 = s1 + " " + s2; /* concatenate strings */

    std::cout << s3 << std::endl;
    std::cout << s3.size() << std::endl; /* print size */
}
```

---

- ▶ `std::string`
  - Represents text:
    - A sequence (`string`) of `characters`

---

```
#include <iostream>
#include <string>

int main() {
    std::string s1 = "Hello";
    std::string s2 = "World";
    std::string s3 = s1 + " " + s2; /* concatenate strings */

    std::cout << s3 << std::endl;
    std::cout << s3.size() << std::endl; /* print size */
}
```

---

## ► std::string

- Represents text:
  - A sequence (**string**) of **characters**

---

```
#include <iostream>
#include <string>

int main() {
    std::string s1 = "Hello";
    std::string s2 = "World";
    std::string s3 = s1 + " " + s2; /* concatenate strings */

    std::cout << s3 << std::endl;
    std::cout << s3.size() << std::endl; /* print size */
}
```

---



## ► std::string

- Represents text:
  - A sequence (**string**) of **characters**

---

```
#include <iostream>
#include <string>

int main() {
    std::string s1 = "Hello";
    std::string s2 = "World";
    std::string s3 = s1 + " " + s2; /* concatenate strings */

    std::cout << s3 << std::endl;
    std::cout << s3.size() << std::endl; /* print size */
}
```

---

- ▶ `std::string`
  - Represents text:
    - A sequence (`string`) of `characters`

---

```
#include <iostream>
#include <string>

int main() {
    std::string s1 = "Hello";
    std::string s2 = "World";
    std::string s3 = s1 + " " + s2; /* concatenate strings */

    std::cout << s3 << std::endl;
    std::cout << s3.size() << std::endl; /* print size */
}
```

---

- ▶ `std::string`
  - Represents text:
    - A sequence (`string`) of `characters`

---

```
#include <iostream>
#include <string>

int main() {
    std::string s1 = "Hello";
    std::string s2 = "World";
    std::string s3 = s1 + " " + s2; /* concatenate strings */

    std::cout << s3 << std::endl;
    std::cout << s3.size() << std::endl; /* print size */
}
```

---

- ▶ std::map

- ▶ `std::map`
  - Creates **mapping** between two **data types**

## ▶ std::map

- Creates **mapping** between two **data types**
- Can think of it as:  $K \rightarrow V$

## ▶ std::map

- Creates **mapping** between two **data types**
- Can think of it as:  $K \rightarrow V$
- Declaration: `std::map<KeyType, ValueType>`

## ▶ std::map

- Creates **mapping** between two **data types**
- Can think of it as:  $K \rightarrow V$
- Declaration: `std::map<KeyType, ValueType>`

---

```
#include <iostream>
#include <map>

int main() {
    std::map<std::string, int> student_nr;
    student_nr["Pete Johnson"] = 47823;
    student_nr["John Snow"] = 49529;

    std::cout << student_nr["John Snow"] << std::endl;
}
```

---



## ▶ std::map

- Creates **mapping** between two **data types**
- Can think of it as:  $K \rightarrow V$
- Declaration: `std::map<KeyType, ValueType>`

---

```
#include <iostream>
#include <map>

int main() {
    std::map<std::string, int> student_nr;
    student_nr["Pete Johnson"] = 47823;
    student_nr["John Snow"] = 49529;

    std::cout << student_nr["John Snow"] << std::endl;
}
```

---

## ▶ std::map

- Creates **mapping** between two **data types**
- Can think of it as:  $K \rightarrow V$
- Declaration: `std::map<KeyType, ValueType>`

---

```
#include <iostream>
#include <map>

int main() {
    std::map<std::string, int> student_nr;
    student_nr["Pete Johnson"] = 47823;
    student_nr["John Snow"] = 49529;

    std::cout << student_nr["John Snow"] << std::endl;
}
```

---

## ▶ std::map

- Creates **mapping** between two **data types**
- Can think of it as:  $K \rightarrow V$
- Declaration: `std::map<KeyType, ValueType>`

---

```
#include <iostream>
#include <map>

int main() {
    std::map<std::string, int> student_nr;
    student_nr["Pete Johnson"] = 47823;
    student_nr["John Snow"] = 49529;

    std::cout << student_nr["John Snow"] << std::endl;
}
```

---

## ▶ std::map

- Creates **mapping** between two **data types**
- Can think of it as:  $K \rightarrow V$
- Declaration: `std::map<KeyType, ValueType>`

---

```
#include <iostream>
#include <map>

int main() {
    std::map<std::string, int> student_nr;
    student_nr["Pete Johnson"] = 47823;
    student_nr["John Snow"] = 49529;

    std::cout << student_nr["John Snow"] << std::endl;
}
```

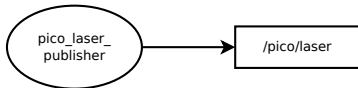
---



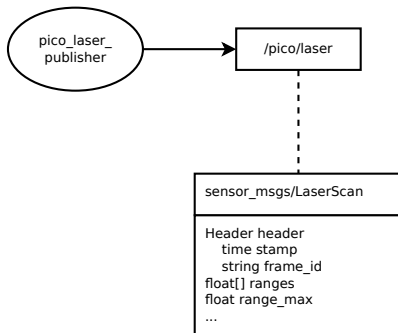
- ▶ **Node:** process that performs computation
- ▶ **Master:** provides name registration and lookup
- ▶ **Messages:** nodes communicate with each other by passing messages
- ▶ **Topics:** named buses over which nodes exchange messages

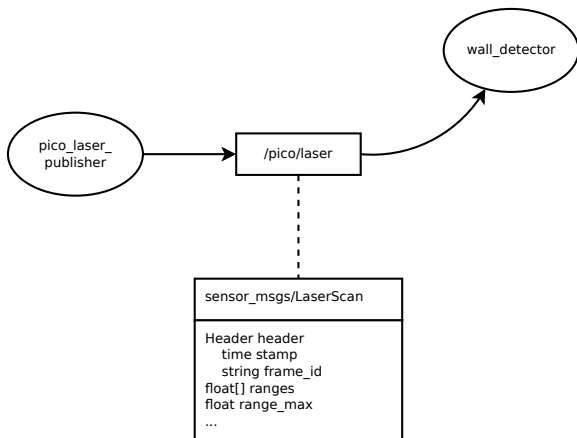


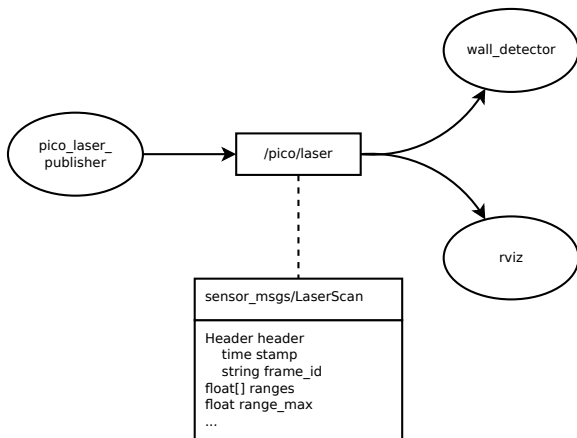
pico\_laser\_  
publisher











sensor\_msgs/LaserScan

Header header  
  time stamp  
  string frame\_id  
float[] ranges  
float range\_max  
...

Is in fact:

sensor_msgs/LaserScan
Header header time stamp string frame_id float[] ranges float range_max ...

---

```
struct LaserScan {
    Header header;
    float range_max;
    std::vector<float> ranges;
}

struct Header {
    std::string frame_id;
    Time stamp;
}

struct Time {
    int secs;
    int nsecs;
}
```

---

Is in fact:

sensor_msgs/LaserScan
Header header time stamp string frame_id float[] ranges float range_max ...

---

```
struct LaserScan {
  Header header;
  float range_max;
  std::vector<float> ranges;
}

struct Header {
  std::string frame_id;
  Time stamp;
}

struct Time {
  int secs;
  int nsecs;
}
```

---

Is in fact:

sensor_msgs/LaserScan
Header header time stamp string frame_id float[] ranges float range_max ...

---

```
struct LaserScan {
  Header header;
  float range_max;
  std::vector<float> ranges;
}

struct Header {
  std::string frame_id;
  Time stamp;
}

struct Time {
  int secs;
  int nsecs;
}
```

---





- ▶ To use ROS in your program:

---

```
#include <ros/ros.h>
```

---

- ▶ To use ROS in your program:

---

```
#include <ros/ros.h>
```

---

- ▶ Register your program as a node to the ROS master:

---

```
ros::init(..., "your_node_name");
```

---

- ▶ To use ROS in your program:

---

```
#include <ros/ros.h>
```

---

- ▶ Register your program as a node to the ROS master:

---

```
ros::init(..., "your_node_name");
```

---

- ▶ Let ROS know you want to listen to a certain topic:

---

```
ros::NodeHandle n;  
ros::Subscriber sub = n.subscribe("/pico/laser", 1,  
                                callbackFunction);
```

---

- ▶ Let ROS know you want to **listen** to a certain **topic**:

---

```
ros::NodeHandle n;  
ros::Subscriber sub = n.subscribe("/pico/laser", 1,  
                                  callbackFunction);
```

---

- ▶ Let ROS know you want to **listen** to a certain **topic**:

---

```
ros::NodeHandle n;  
ros::Subscriber sub = n.subscribe("/pico/laser", 1,  
                                  callbackFunction);
```

---

- ▶ Start listening to the topics:

---

```
ros::spin();
```

---

- ▶ Let ROS know you want to **listen** to a certain **topic**:

---

```
ros::NodeHandle n;  
ros::Subscriber sub = n.subscribe("/pico/laser", 1,  
                                  callbackFunction);
```

---

- ▶ Start listening to the topics:

---

```
ros::spin();
```

---

- ▶ This **function** is called every time the node **receives a message**:

---

```
void callbackFunction(sensor_msgs::LaserScan scan) {  
    // do something  
    std::cout << scan.header.stamp << std::endl;  
}
```

---

---

```
#include <ros/ros.h>           // include ROS
#include <sensor_msgs/LaserScan.h> // include LaserScan
                                   // message type

void callback(sensor_msgs::LaserScan scan) {
    // do something
    std::cout << scan.header.stamp << std::endl;
}

int main(int argc, char** argv) { // ignore argc and argv
    ros::init(argc, char** argv, "example"); // register node
                                              // to master

    ros::NodeHandle n;
    ros::Subscriber sub = n.subscribe("/pico/laser",
                                      1, callback);

    ros::spin(); // Keep receiving messages
    return 0;
}
```

---





- ▶ The **callback function** gets the ROS message as **input argument**:

---

```
void callback(sensor_msgs::LaserScan scan) {  
    // do something  
    std::cout << scan.header.stamp << std::endl;  
}
```

---

- ▶ The **callback function** gets the ROS message as **input argument**:

---

```
void callback(sensor_msgs::LaserScan scan) {  
    // do something  
    std::cout << scan.header.stamp << std::endl;  
}
```

---

- ▶ Remember:

---

```
struct LaserScan {  
    Header header;  
    float range_max;  
    std::vector<float> ranges;  
}
```

---

- ▶ So, we can take a look a the data inside:

- ▶ So, we can take a look at the data inside:

---

```
void callback(sensor_msgs::LaserScan scan) {  
    for(unsigned int i = 0; i < scan.ranges.size(); i++) {  
        if (scan.ranges[i] < 0.3) {  
            std::cout << "HELP!" << std::endl;  
            // ...  
        }  
    }  
}
```

---

- ▶ So, we can take a look at the data inside:

---

```
void callback(sensor_msgs::LaserScan scan) {  
  
    for(unsigned int i = 0; i < scan.ranges.size(); i++) {  
        if (scan.ranges[i] < 0.3) {  
            std::cout << "HELP!" << std::endl;  
            // ...  
        }  
    }  
  
}
```

---

- ▶ Try to get this example running
  - Together with the [pico\\_example](#) package, it's a good [start for your project](#)

- ▶ C++ Data Types
- ▶ Structs
- ▶ C++ Standard Library
  - `std::vector`
  - `std::string`
  - `std::map`
- ▶ ROS Message as C++ struct
- ▶ ROS C++ Subscriber Example