Cairo University
Faculty of Engineering
Computer Department

Computer Networks-1
CMP405-A /CMPN405
Fall 2023

Custom Messages and Noisy channels

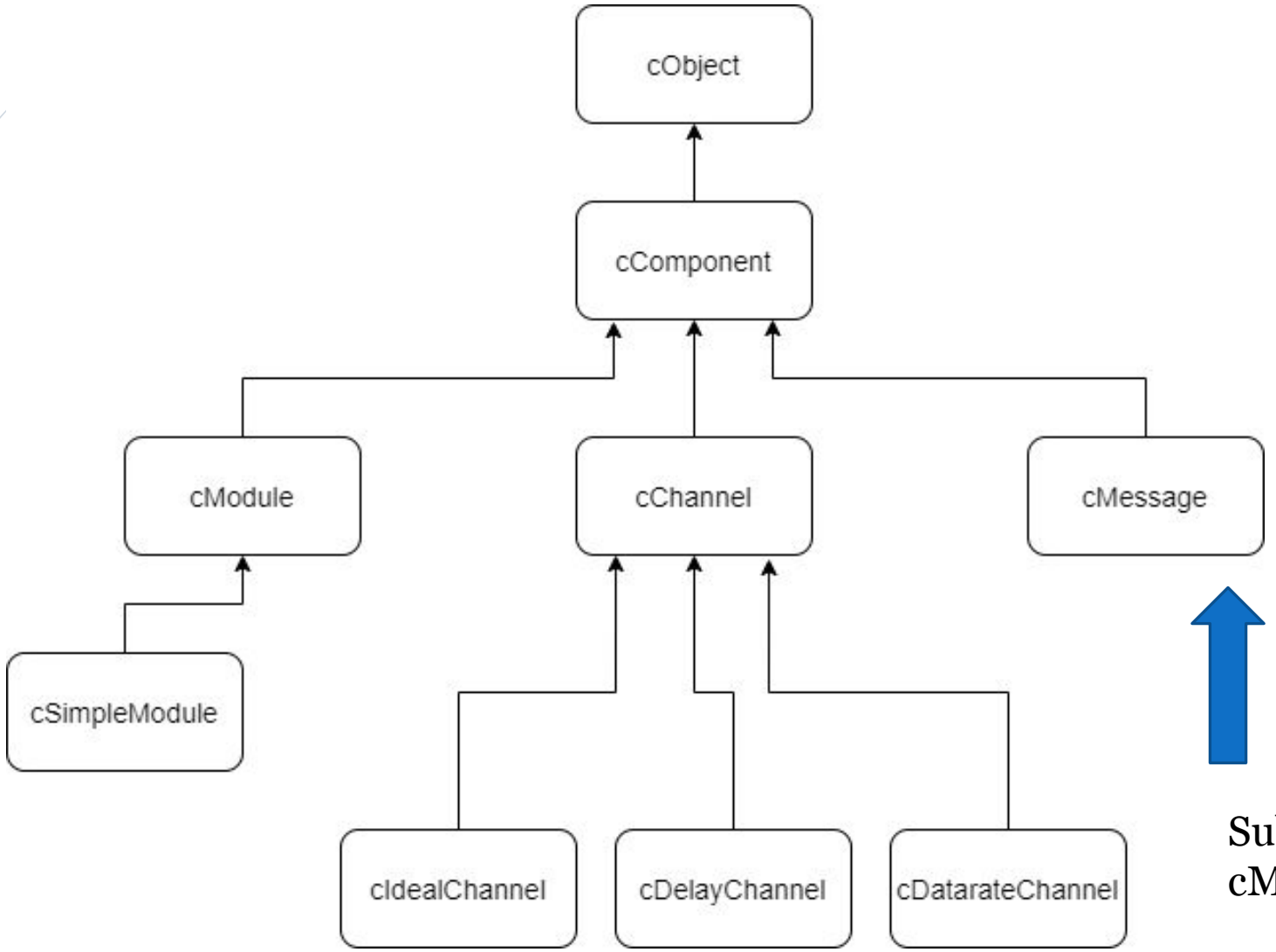
Lab Objectives

- Implement and use custom message classes.
- Simulate two of the effects of a noisy channel on messages:
  1. Sending delay
  2. Frame corruption

cMessage class inheritance

- Till now we have been working with messages that carry only one string as the data [the message name].
- What if we want to add some other data types to the message ?
- Even more, can we combine multiple data types like [sequence numbers , frame type , payload and checksum] in the message?
- The answer is yes, and we will do this through message class inheritance or sub-classing.

cMessage class inheritance



Sub-classing the cMessage class

The cMessage class

- The built in cMessage class has the following fields:
  1. The name field is a string (const char *)
  2. Message kind is an integer field. Some negative values are reserved by the simulation library, but zero and positive values can be freely used in the model for any purpose.
  3. The scheduling priority field is used by the simulation kernel to determine the delivery order of messages that have the same arrival time values. This field is rarely used in practice.

The cMessage class

- The built in cMessage class has the following fields:

4. The send time, arrival time, source module, source gate, destination module, destination gate fields are used internally by the simulation kernel while the message is in the future events set (FES), and should not be modified.
5. Time stamp (not to be confused with arrival time) is a utility field, which the programmer can freely use for any purpose
6. Message ID : a unique numeric field used for identifying the message in a recorded event log file

The cMessage class

- The built in cMessage class has the following methods:

  The constructor

  ```
  cMessage(const char *name=nullptr, short kind=0);
  ```

  Other setters

  ```
  void setName(const char *name);
  void setKind(short k);
  void setTimestamp();
  void setTimestamp(simtime_t t);
  void setSchedulingPriority(short p);
  ```

The cMessage class

- The built in cMessage class has the following methods:

Other getters

```
const char *getName() const;
short getKind() const;
simtime_t getTimestamp() const;
short getSchedulingPriority() const;
long getId() const;
```

The cMessage class

- The built in cMessage class has the following methods:

The dup method

cMessage *copy = msg->dup();

The resulting message (or packet) will be an exact copy of the original including message parameters and encapsulated messages, except for the message ID field

When sub classing cMessage we need to reimplement dup()

The cMessage class

- The built in cMessage class has the following methods:

The getDisplayString method

const char *getDisplayString() const;

Display strings affect the message's visualization in graphical user interfaces like Tkenv and Qtenv.

getDisplayString() method can be overridden in subclasses to return the desired string.

The cMessage class

- The built in cMessage class has the following methods:

  Other methods like

  bool isSelfMessage() const;

  can be used to determine if it is a self-message

  bool isScheduled() const;

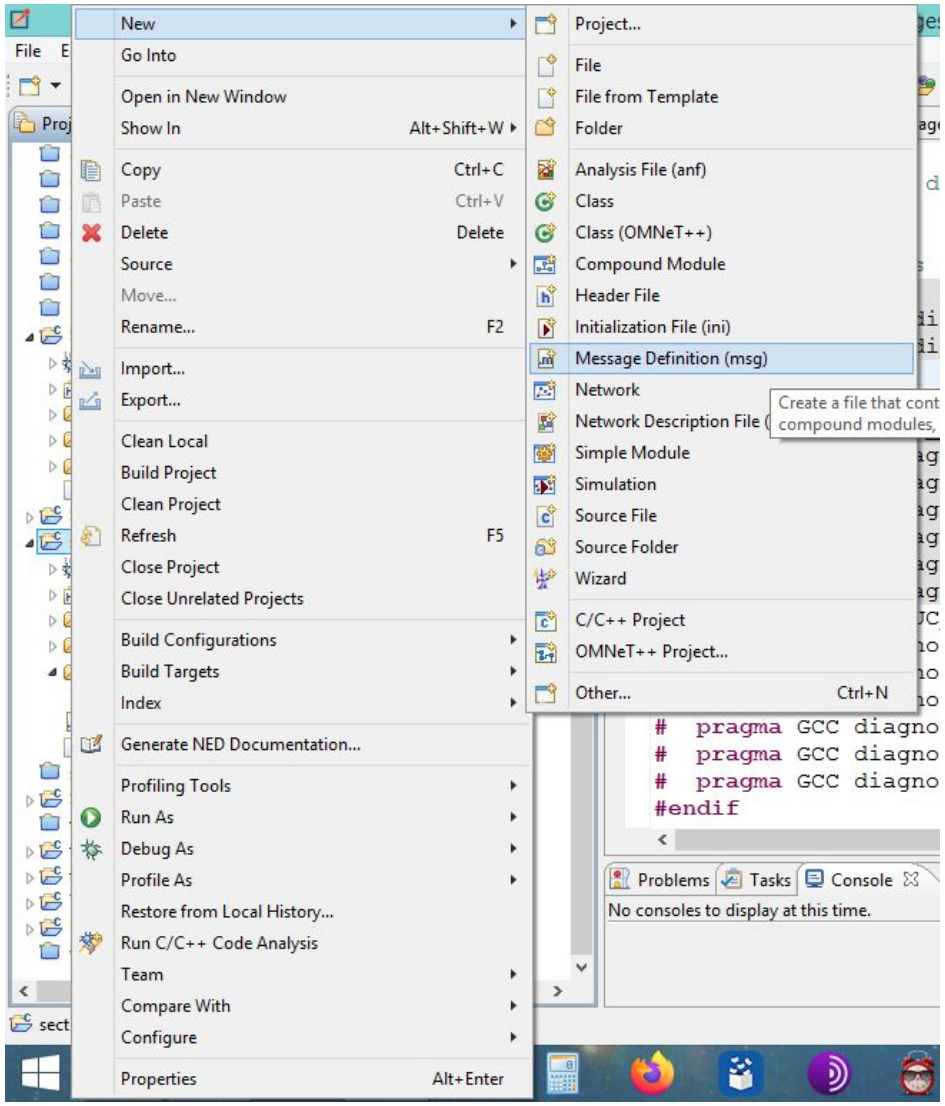
  returns true if the message is currently scheduled.

The cMessage class inheritance

- Now let's define new sub class from the cMessage class to add more fields.
- The best and easiest way to do this is through the automatic utility provided by omnetpp.

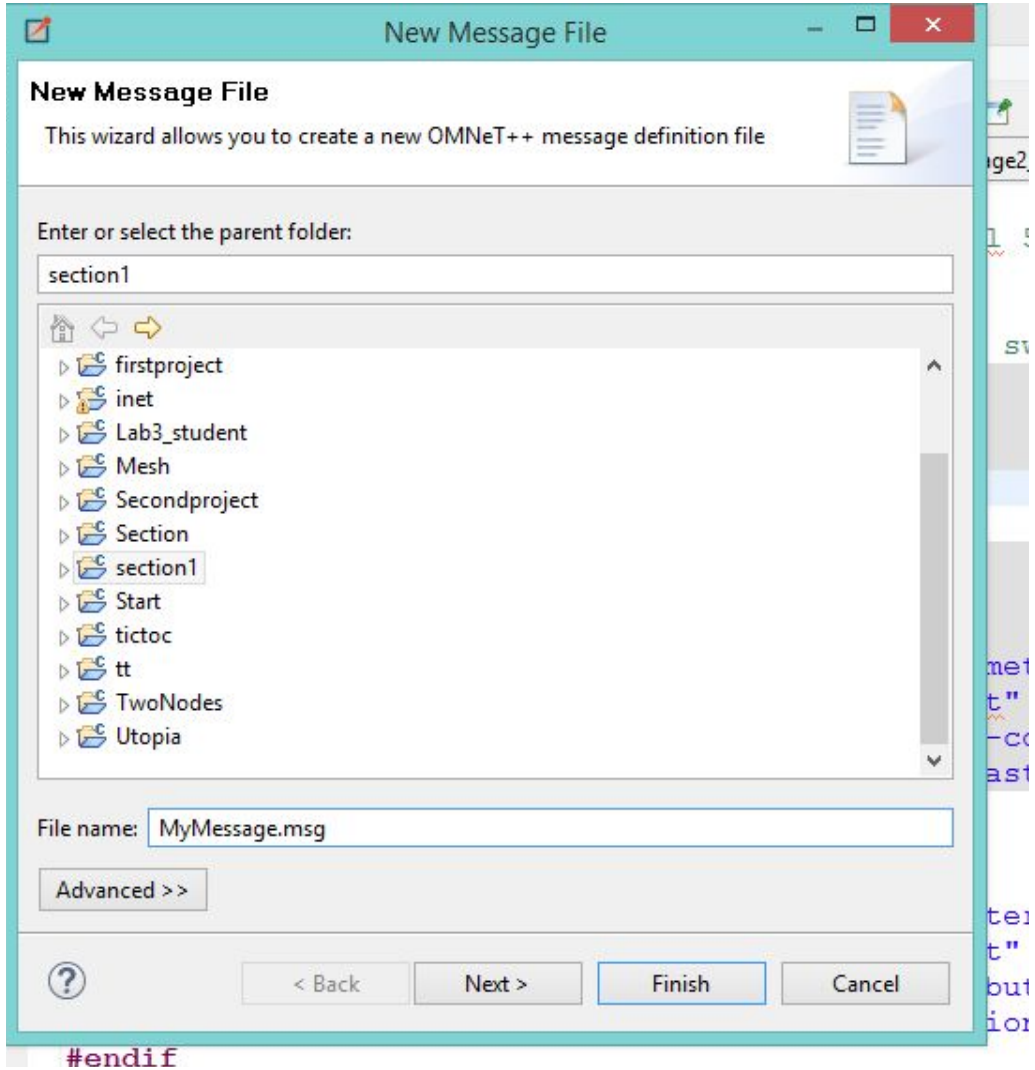
The cMessage class inheritance

- Inside your project:
  right click on the project folder.
  Select new
  Select Message Definition (.msg)



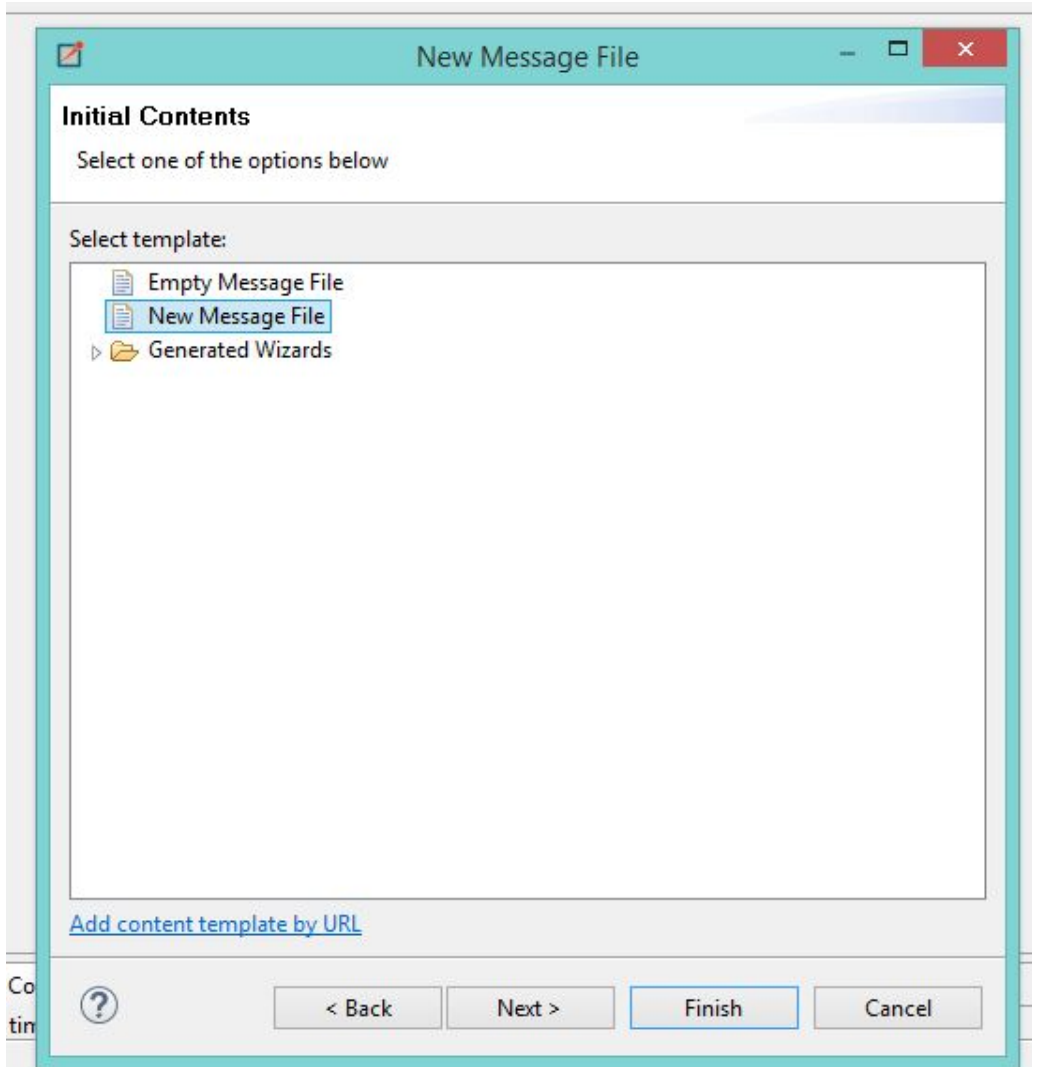
The cMessage class inheritance

- Give it a new name. and click next.



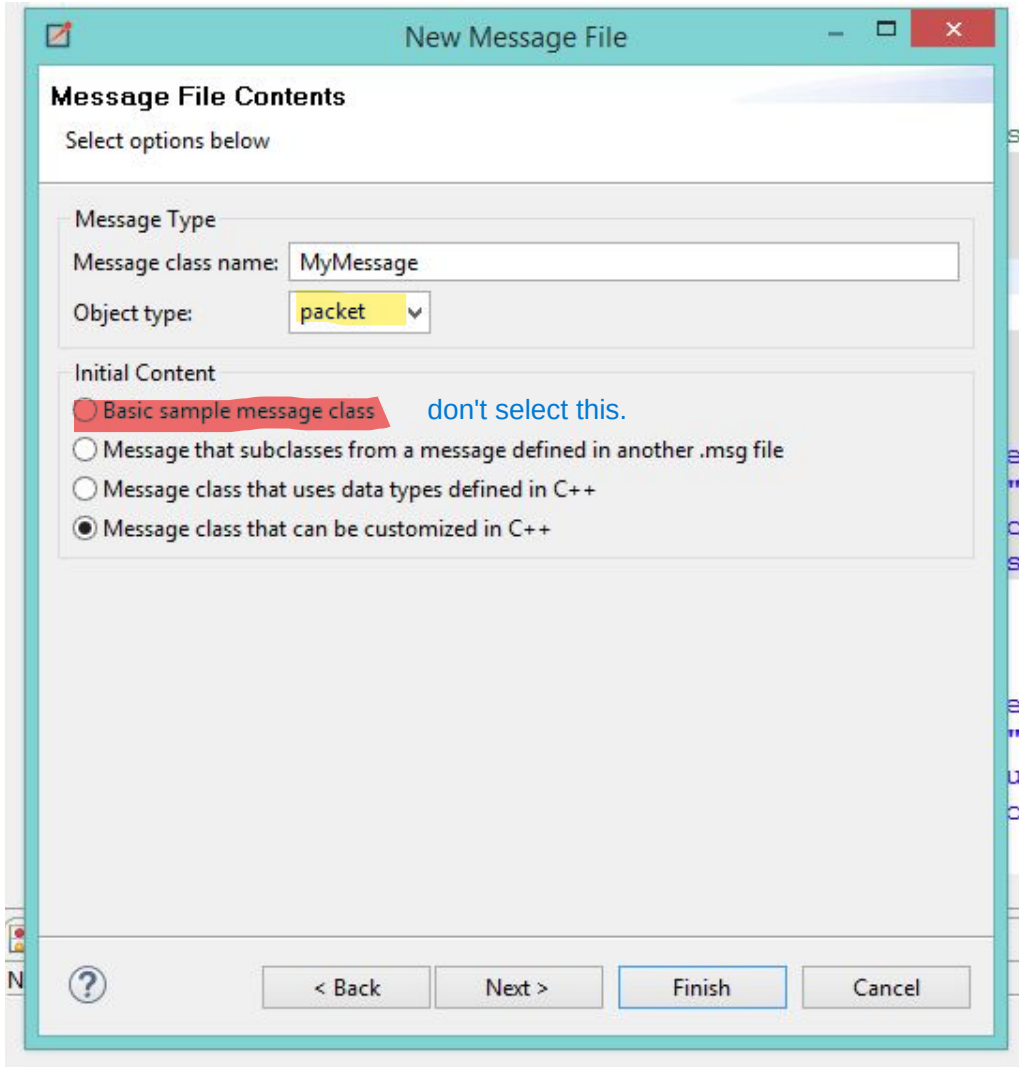
The cMessage class inheritance

- Choose a New Message File. and finish.



The cMessage class inheritance

- Choose Message class that can be customized in c++ and finish.



The cMessage class inheritance

- You will find dummy data fields generated for your reference.
- We will modify that.

```
//
// TODO generated message class
//
packet MyMessage {
    @customize(true);  // see the gene
    int someField;
    abstract int anotherField;
}
```

The cMessage class inheritance

6.6.1 Data Types

The following data types can be used for fields:

- C/C++ primitive data types: **`bool`**, **`char`**, **`short`**, **`int`**, **`long`**, **`unsigned char`**, **`unsigned short`**, **`unsigned int`**, **`$2`**unsigned long, **`float`**, **`double`**.
- **`string`**. Getters and setters use the **`const char*`** data type; **`nullptr`** is not allowed. Setters store a copy of the string, not just the pointer.
- C99-style fixed-size integer types: **`int8_t`**, **`int16_t`**, **`int32_t`**, **`int64_t`**, **`uint8_t`**, **`uint16_t`**, **`uint32_t`**, **`uint64 t`**.

The cMessage class inheritance

- We can also use the cplusplus code blocks to directly generate the code.
- Specifically for custom types that needs includes

As far as a the code fragment is concerned, the message compiler does not try to make sense of it, just simply copies it into the generated source file at the requested location. The code fragment should be formatted so that it does not contain a double close curly brace (}}) because it would be interpreted as end of the fragment block.

> [Should this ever be a problem, just insert a space between the two braces, or use the automatic concatenation of adjacent string literals feature of C/C++ if they occur within a string constant, i.e. break up `"foo}}bar"` into `"foo}" "}bar"`.]

```
cplusplus {{
#include "FooDefs.h"
#define SOME_CONSTANT 63
}}
```

The cMessage class inheritance

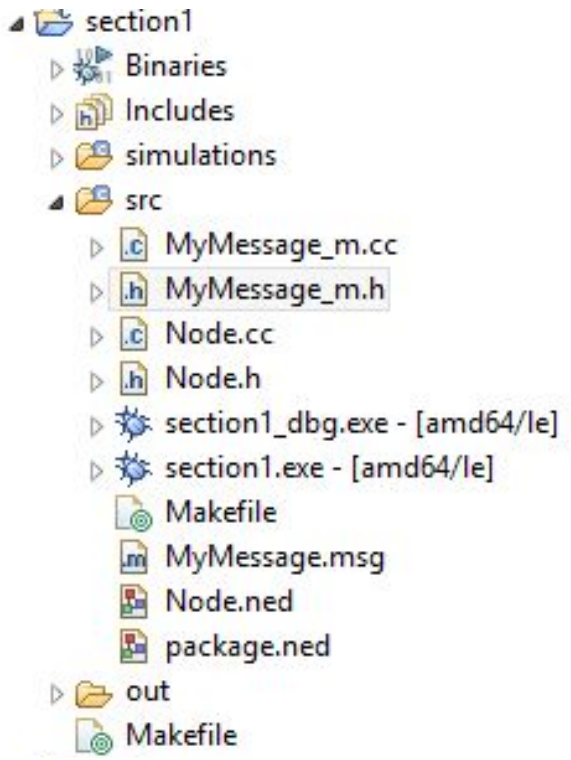
- You should add your data members at this step.
- If you want to include STL library types you should do this inside a cpluplus block and define it as noncobject as well because the .msg file is an ned file.

```
// TODO generated message class
//
cplusplus {{
#include <bitset>
typedef  std::bitset<8> bits;
}}
class noncobject bits;

packet MyMessage {
    @customize(true);  // see the
    int Seq_Num;
    int M_Type;
    string M_Payload;
    bits mycheckbits;
}
```

The cMessage class inheritance

- Now drag your .msg file inside the src folder and build your project.
- After any change to the .msg file you need to re-build your project so that the .h and .cc files get updated
- The compiler will automatically generate .h and .cc files for your custom message class.
- You will find in the .cc and .h: automatic setters, getters and prints for your fields.



The cMessage class inheritance

```
std::string MyMessageDescriptor::getFieldValueAsString(void *object, in
{
    omnetpp::cClassDescriptor *basedesc = getBaseClassDescriptor();
    if (basedesc) {
        if (field < basedesc->getFieldCount())
            return basedesc->getFieldValueAsString(object,field,i);
        field -= basedesc->getFieldCount();
    }
    MyMessage_Base *pp = (MyMessage_Base *)object; (void)pp;
    switch (field) {
        case 0: return long2string(pp->getSeq_Num());
        case 1: return long2string(pp->getM_Type());
        case 2: return oppstring2string(pp->getM_Payload());
        case 3: return pp->getMycheckbits().to_string();
        default: return "";
    }
}
```

- If you use STL data types you might find some errors in the print function as these types cannot be casted to strings automatically.

The cMessage class inheritance

- Now you can use your custom message class like you use cMessage .
- In you c++ files include the .h file
- Go to your .h file and find the class name to use it here it happens to be named “MyMessage_Base”.

Node.h | Node.cc | MyMessage_m.h | MyMessage_m.cc

```
#include "Node.h"
#include "MyMessage_m.h"
Define_Module(Node);

void Node::initialize()
{
    // TODO - Generated method body
    if ( strcmp(getName(),"Tic")==0)
    {
        MyMessage_Base * msg = new MyMessage_Base("Hello");
        msg->setM_Payload("hi there!");
        msg->setM_Type(1);
        msg->setSeq_Num(12);
        std::bitset<8> temp('A');
        msg->setMycheckbits(temp);
      send(msg,"out");
    }
}
```

The cMessage class inheritance

- make sure that the constructor is public in the .h file.
- implement the dup virtual function.

Node.h | Node.cc | MyMessage_m.h | MyMessage_m.cc

```
    bool operator==(const MyMessage_Base&);
    // make constructors protected to avoid instantiation

    MyMessage_Base(const MyMessage_Base& other);
    // make assignment operator protected to force the user override
    MyMessage_Base& operator=(const MyMessage_Base& other);

  public:
    MyMessage_Base(const char *name=nullptr, short kind=0);
    virtual ~MyMessage_Base();
    virtual MyMessage_Base *dup() const override {
        return new MyMessage_Base(*this);}
```

The cMessage class inheritance

- At the receiver side, the handleMessage method only accepts parameters of type cMessage.
- So we need to cast down it to our message class type.
- Like this :

```
MyMessage_Base *mmsg = check_and_cast<MyMessage_Base *>(msg);
```

The cMessage class inheritance

- We can access our setters easily then.

```
void Node::handleMessage(cMessage *msg)
{
    // TODO - Generated method body
    MyMessage_Base *mmsg = check_and_cast<MyMessage_Base *>(msg);
    EV<<"received message with sequence number ...    ";
    EV << mmsg->getSeq_Num();
    EV<<"  and payload of ... ";
    EV<< mmsg->getM_Payload();
    EV<<"   and check bits of ...";
    EV<< mmsg->getMycheckbits().to_string();
    send(mmsg,"out");

}
```

The cMessage class inheritance

- The output …


```
** Initializing network
Initializing module Network, stage 0
Initializing module Network.Tic, stage 0
Initializing module Network.Toc, stage 0
** Event #1  t=0  Network.Toc (Node, id=3)  on Hello (MyMessage_Base, id=0)
received message with sequence number ...   12  and payload of ... hi there!   and check bits of ...01000001
** Event #2  t=0  Network.Tic (Node, id=2)  on Hello (MyMessage_Base, id=0)
received message with sequence number ...   12  and payload of ... hi there!   and check bits of ...01000001
```

Noisy channel

- In the omnetpp we have three built in channel types:
  - cIdealChannel
  - cDelayChannel
  - cDatarateChannel
- They can be used to represent different errors on the channel but not to the level of details and control we want in our project scope.
- So we will only use the default cIdealChannel and simulate the errors at the sender side before sending. And you should also do that in the project.

Simulation of message delay

- Modeling the message delay will be done in the sender side .
- i.e. once we send the message it will be received directly at the receiver so we will make the delay at the sender himself.
- This can be done through multiple ways, one way is using the **sendDelayed** method.
- The other way is using **scheduleAt**

```
sendDelayed(cMessage *msg, double delay, const char *gateName, int index);
sendDelayed(cMessage *msg, double delay, int gateId);
sendDelayed(cMessage *msg, double delay, cGate *gate);
```

```
scheduleAt(simTime() + exponential(1.0), msg);
```

Simulation of message delay

- You can handle the delay interval using random variables.
- You can even handle the probability of delay or not using the same.

```
int rand=uniform(0,1)*10;
EV<<"rand is "<<std::to_string(rand)<<endl;
if(rand>=7) // prob to delay the message
{
    EV<<"delaying message with 0.2"<<endl;
    sendDelayed(mmsg,0.2,"out");
}
else
{
send(mmsg,"out");
}
```

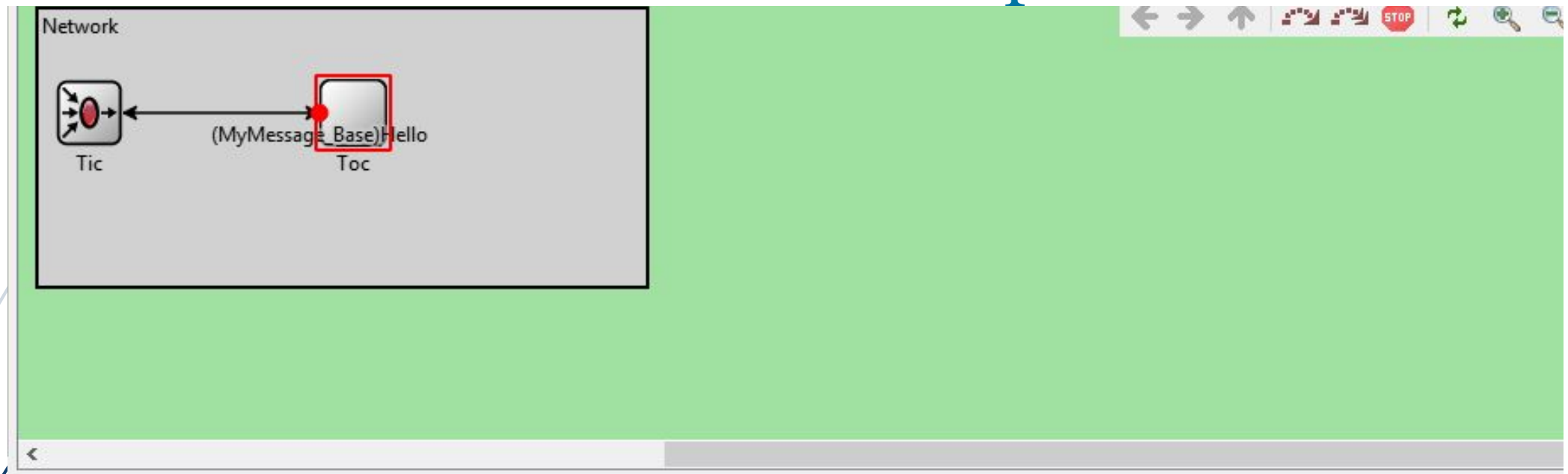
Simulation of frame corruption

- Modeling the frame corruption can be done through various ways.
- All of them will make use of random variables to model the probability of errors.
- Here we will modify the first character of the payload with a 80% probability.

```
int rand=uniform(0,1)*10;
    EV<<"rand is "<<std::to_string(rand)<<endl;
    if(rand<8) // prob to delay the message
    {
        std::string mypayload= mmsg->getM_Payload();
        mypayload[0]=mypayload[0]+5;
        mmsg->setM_Payload(mypayload.c_str());
    }

send(mmsg,"out");
```

Simulation of frame corruption




```
** Initializing network
Initializing module Network, stage 0
Initializing module Network.Tic, stage 0
Initializing module Network.Toc, stage 0
** Event #1  t=0  Network.Toc (Node, id=3)  on Hello (MyMessage_Base, id=0)
INFO:received message with sequence number ...   12  and payload of ... hi there!   and check bits of ...01000001rand is 5
** Event #2  t=0  Network.Tic (Node, id=2)  on Hello (MyMessage_Base, id=0)
INFO:received message with sequence number ...   12  and payload of ... mi there!   and check bits of ...01000001rand is 5
** Event #3  t=0  Network.Toc (Node, id=3)  on Hello (MyMessage_Base, id=0)
INFO:received message with sequence number ...   12  and payload of ... ri there!   and check bits of ...01000001rand is 7
** Event #4  t=0  Network.Tic (Node, id=2)  on Hello (MyMessage_Base, id=0)
INFO:received message with sequence number ...   12  and payload of ... wi there!   and check bits of ...01000001rand is 8
```

Lab5 requirement

- Redo lab4 using the customized message class that has the following fields:
  - M_Header ‘one char’
  - M_Payload ‘string of characters’
  - M_Trailer ‘one char’
  - M_Type ‘ int [2,1,0]’ for the types [data,ack, notack]
- When the receiver responds with ack/notack the sender send a new message. i.e. just ignore this ack/notack.
- Every sent message in your system should be printed in the EV with all it’s fields .
- The sender will modify any single bit in the message payload with a 50%chance.
- If the sender will add error, it should print a message “adding error” to the EV.

Submission

- Work in pairs
- Submit one .zip project folder contains the solution to lab4 and lab5
- In the private comments write your names and Id’s.
- We will have the submitted labs scheduled for discussion later on isA.

Thank You !!

Extra materials: check chapter 6 in the manual

- You can define arrays in the msg definition too

When a default value is given, it is interpreted as a scalar for filling the array with. There is no syntax for initializing an array with a list of values.

```
int route[4] = -1; // all elements set to -1
```

6.7.7 Variable-Size Arrays

If the array size is not known in advance, the field can be declared to have a variable size by using an empty pair in brackets:

```
int route[];
```

In this case, the generated class will have extra methods in addition to the getter and setter: one for resizing the array, one for getting the array size, plus methods for inserting an element at a given position, appending an element, and erasing an element at a given position.

Extra materials: check chapter 6 in the manual

- The std::to_string() is very helpful in converting custom types to be presentable.

```
std::string Location::str() const
{
    return "(" + std::to_string(getLat()) + "," + std::to_string(getLon()) + ")";
}
```