THREAD COMMUNICATION

AGENDA

- Communication design challenges
- Message Queue and Handler Design
- Implementation usage
- Consequences

COMMUNICATION DESIGN CHALLENGES

COMMUNICATION DESIGN CHALLENGES

- Individual threads wait for a condition to become true

COMMUNICATION DESIGN CHALLENGES

- Individual threads wait for a condition to become true
- Enter and leave critical sections using mutexes or semaphores
  - May happen multiple times in the space of one thread loop iteration

COMMUNICATION DESIGN CHALLENGES

- Individual threads wait for a condition to become true
- Enter and leave critical sections using mutexes or semaphores
  - May happen multiple times in the space of one thread loop iteration
- May even hold multiple resources which have to be synchronized between threads
  - The sequence in which resources are taken must be thought through.

COMMUNICATION DESIGN CHALLENGES

- Individual threads wait for a condition to become true
- Enter and leave critical sections using mutexes or semaphores
  - May happen multiple times in the space of one thread loop iteration
- May even hold multiple resources which have to be synchronized between threads
  - The sequence in which resources are taken must be thought through.

SUMMARY

- A design challenge ensuring that no deadlocks or timing issues exist
- Readability easily becomes an issue too
- High code complexity is the outcome

WHAT WE NEED ...

- *all* processing within a thread must *not* require locking
- however *other* threads must be able to pass control and/or data to a specific thread via some mechanism.
- *multiple* threads may concurrently decide to pass such control and/or data

A STEP BACKWARDS

- What is it in fact we are doing and what?
  - *Perform some action when a given condition becomes true or we get signaled*

A STEP BACKWARDS

- What is it in fact we are doing and what?
  - *Perform some action when a given condition becomes true or we get signaled*

We want events/messages!

EVENT DRIVEN PROGRAMMING

IS REACTIONARY PROGRAMMING

- *Each incoming message is processed by a specific handler*
  - E.g. a *specific function* is called upon receiving som specific *event*

EVENT DRIVEN PROGRAMMING

IS REACTIONARY PROGRAMMING

- *Each incoming message is processed by a specific handler*
  - E.g. a *specific function* is called upon receiving som specific *event*
- Types
  - Censor input
    - Temperature exceeded message → Turn down heat
    - Car detected wanting to enter car park message → Open garage door
  - Signal input
    - Exit button in GUI message → Exit program

EVENT DRIVEN PROGRAMMING (EVENT = MESSAGE)

CAN BE VIEWED AS A TWO PHASE PROCESS

1. ACQUIRE/SELECT NEW MESSAGE

- Handled by a Message Queue and ensures that a number messages can be in "queue" at a time

EVENT DRIVEN PROGRAMMING (EVENT = MESSAGE)

CAN BE VIEWED AS A TWO PHASE PROCESS

1. ACQUIRE/SELECT NEW MESSAGE

- Handled by a Message Queue and ensures that a number messages can be in "queue" at a time

2. PROCESS NEW MESSAGE IN HANDLER

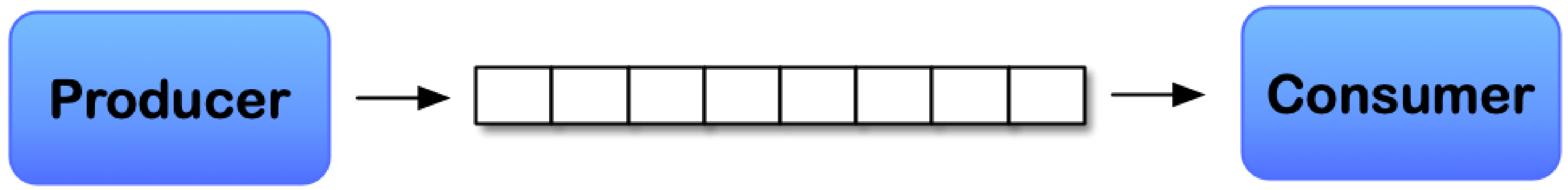
- Handled by casing out on the specific message

MESSAGE QUEUE & HANDLER DESIGN

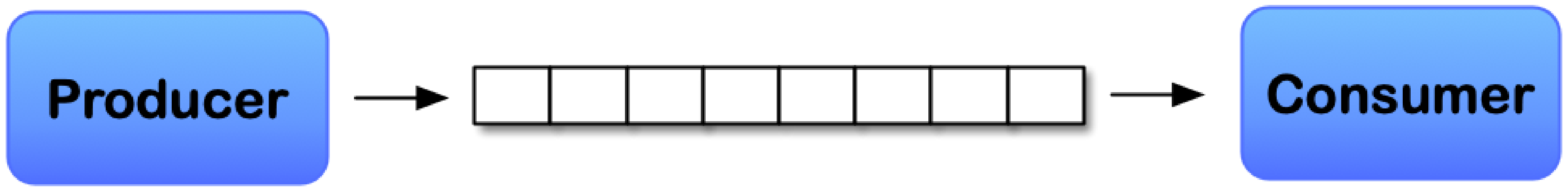
MESSAGE QUEUE & HANDLER DESIGN

- Revisit "Producer & Consumer problem"
- Further requirements
- Structure to *pass* around
- Using inheritance
- Message parsing
- Embedded Compiler configurations

REVISIT "PRODUCER & CONSUMER PROBLEM"

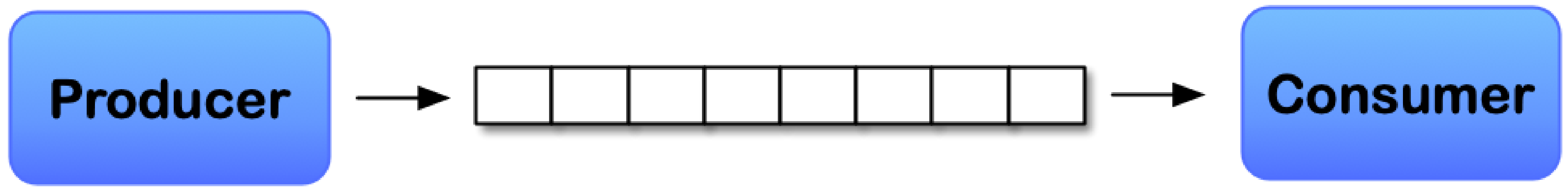


REVISIT "PRODUCER & CONSUMER PROBLEM"

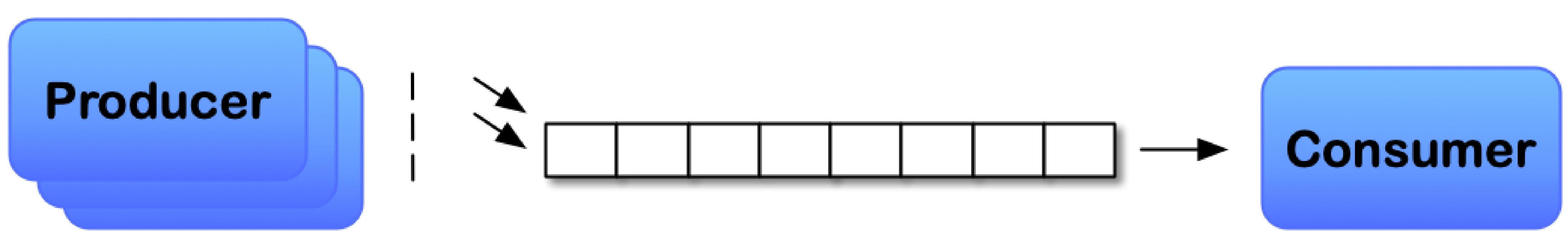


- The producer-consumer problem
  - A producer threads produce buffer items
  - A consumer thread consumes them
- Applied to our problem we get
  - A queue for messages - e.g. a Message Queue = MsgQueue
  - Handling multiple senders and in principle one receiver

REVISIT "PRODUCER & CONSUMER PROBLEM"



- The producer-consumer problem
  - A producer threads produce buffer items
  - A consumer thread consumes them
- Applied to our problem we get
  - A queue for messages - e.g. a Message Queue = MsgQueue
  - Handling multiple senders and in principle one receiver



FURTHER REQUIREMENTS FOR OUR MESSAGE QUEUE

- If the receiving queue is full, then the thread or threads wishing to pass control and/or data must block waiting for more space.
  - Implies that there *is* a maximum number of elements in a queue
- The consuming thread *must* block upon receiving from an empty queue
- Blocks are NOT to be done with polling (+ sleeps), *why?*
- What should we do then? - ***Conditionals***

WHAT IS THE STRUCTURE OF THE INFORMATION TO PASS AROUND?

- **void* or simple array of bytes**
  - Can contain anything
  - No type information - No type-safety (if we don’t know what it is - we don’t know how to delete

WHAT IS THE STRUCTURE OF THE INFORMATION TO PASS AROUND?

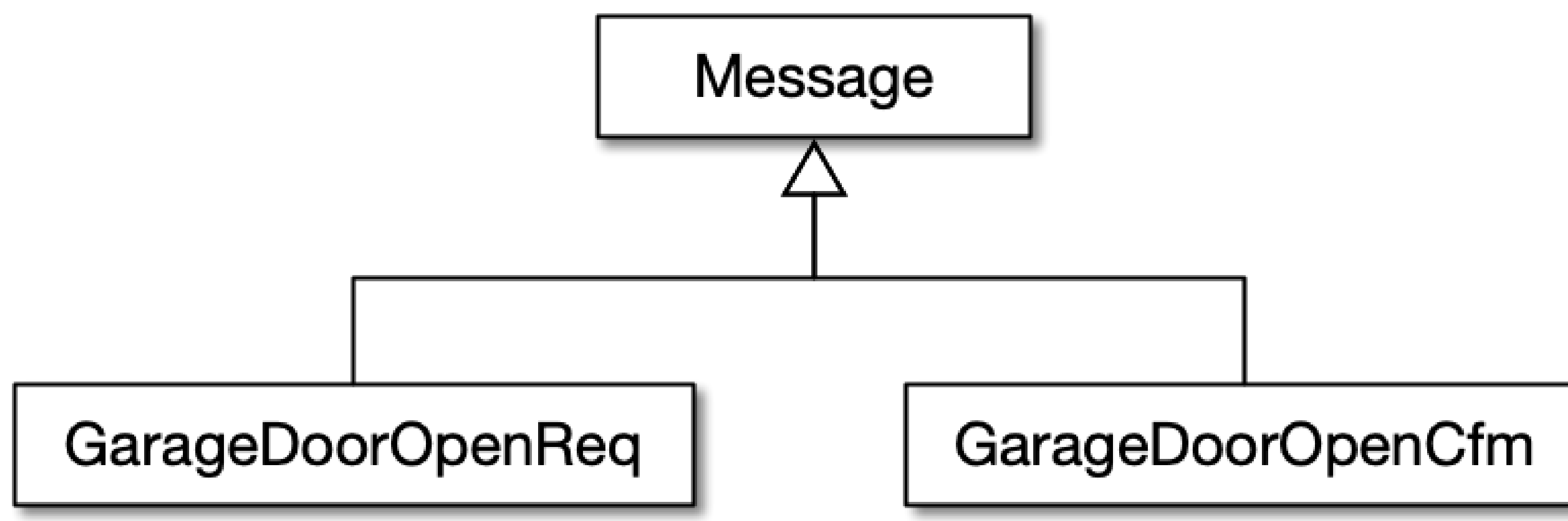
- **void* or simple array of bytes**
  - Can contain anything
  - No type information - No type-safety (if we don't know what it is - we don't know how to delete
- **template based**
  - Depends on the implementation, is a good solution but more complex
  - Type-safety

WHAT IS THE STRUCTURE OF THE INFORMATION TO PASS AROUND?

- **void* or simple array of bytes**
  - Can contain anything
  - No type information - No type-safety (if we don't know what it is - we don't know how to delete
- **template based**
  - Depends on the implementation, is a good solution but more complex
  - Type-safety
- **Inheritance**
  - Simple and extended via sub-classing
  - Type-safety / Type information - Delete via base pointer
  - Might incur overhead

INHERITANCE - OUR CHOICE

MESSAGE HIERARCHY




```cpp
class Message
{
public:
    virtual ~Message(){}
};
```

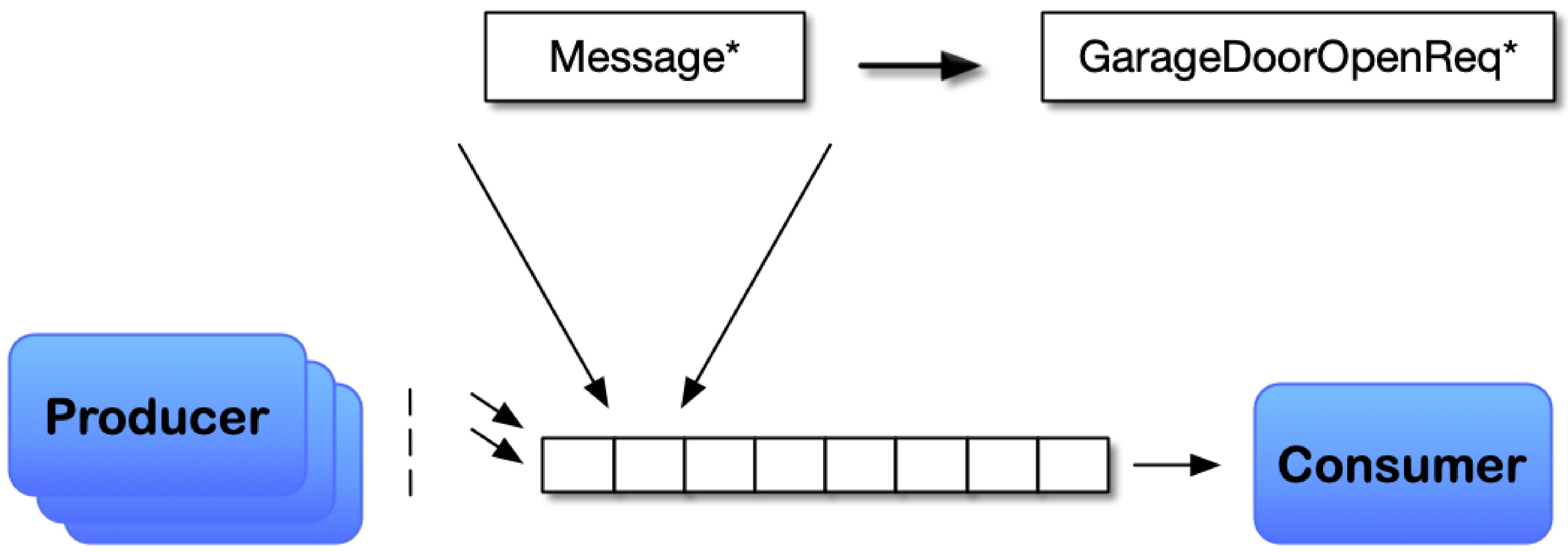
```cpp
struct GarageDoorOpenReq : public Message
{
    MsgQueue* mq_;
};
```

MESSAGE PARSING

- A producer creates and “sends” a GarageDoorOpenReq message
  - `class GarageDoorOpenReq` is therefore seen as a `Message*`

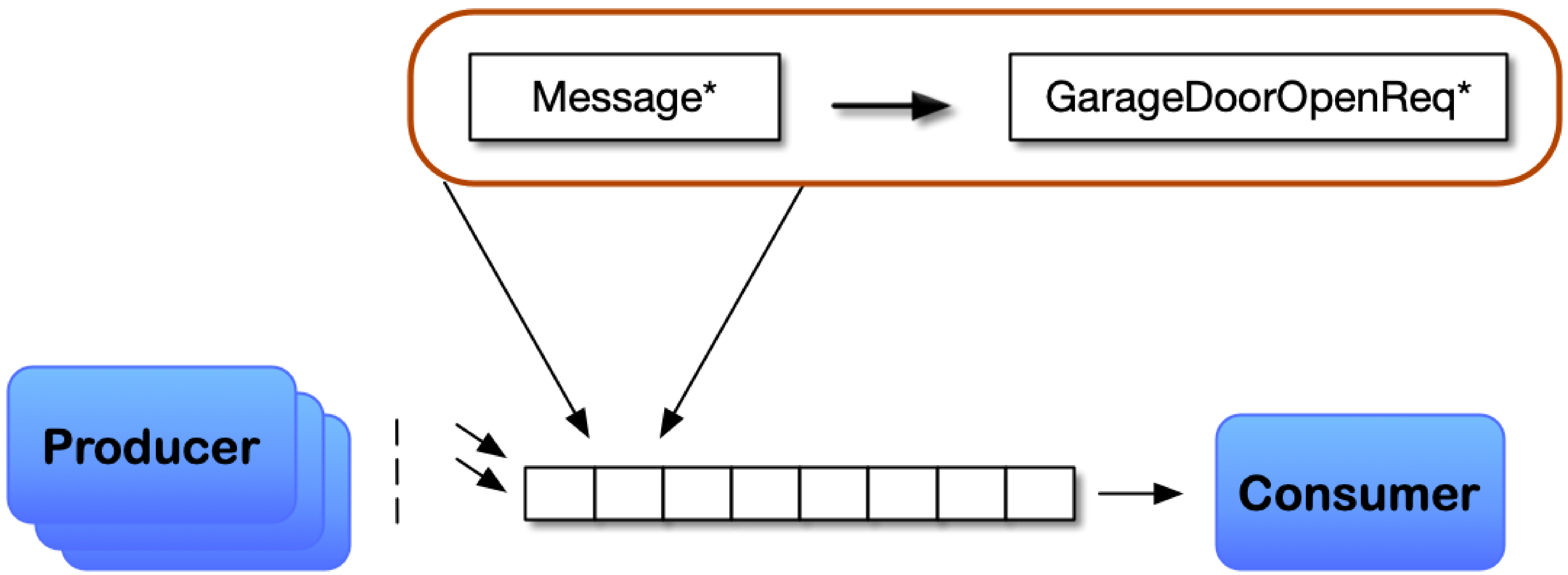
MESSAGE PARSING

- A producer creates and “sends” a GarageDoorOpenReq message
  - `class GarageDoorOpenReq` is therefore seen as a `Message*`



MESSAGE PARSING

- A producer creates and "sends" a GarageDoorOpenReq message
  - `class GarageDoorOpenReq` is therefore seen as a `Message*`



How does receiver determine which message e.g. type it really is?

DETERMINING REAL MESSAGE TYPE

- How do we convert a Message *to a GarageOpenDoorReq*?

DETERMINING REAL MESSAGE TYPE

- How do we convert a Message *to a GarageOpenDoorReq*?
  - Via using `dynamic_cast<>`

```
GarageDoorOpenReq gdor;
Message* msg_ = &gdor; // Illustration!

GarageDoorOpenReq* req = dynamic_cast<GarageDoorOpenReq*>(msg_);
// Runtime check, req == NULL if not correct
```

DETERMINING REAL MESSAGE TYPE

- How do we convert a Message *to a GarageOpenDoorReq*?
  - Via using `dynamic_cast<>`

```
GarageDoorOpenReq gdor;
Message* msg_ = &gdor; // Illustration!

GarageDoorOpenReq* req = dynamic_cast<GarageDoorOpenReq*>(msg_);
// Runtime check, req == NULL if not correct
```

  - Via `typeid()`

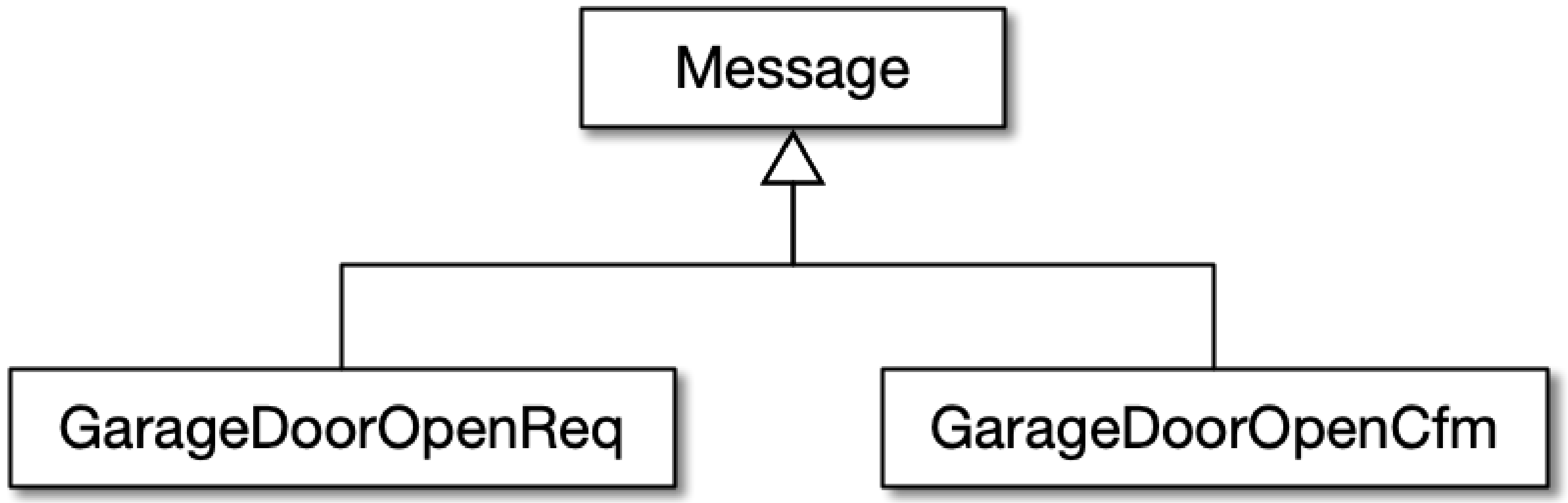
```
GarageDoorOpenReq gdor;
Message* msg_ = &gdor; // Illustration!

if(typeid(*msg_) == typeid(GarageDoorOpenReq))
{
  // Runtime check - evaluates to true if pointer is of said type
  GarageDoorOpenReq* req = static_cast<GarageDoorOpenReq*>(msg_);
}
```

DETERMINING REAL MESSAGE TYPE

- How do we convert a Message *to a GarageOpenDoorReq*?
  - Using a special identifier
  - Associating an id with the message

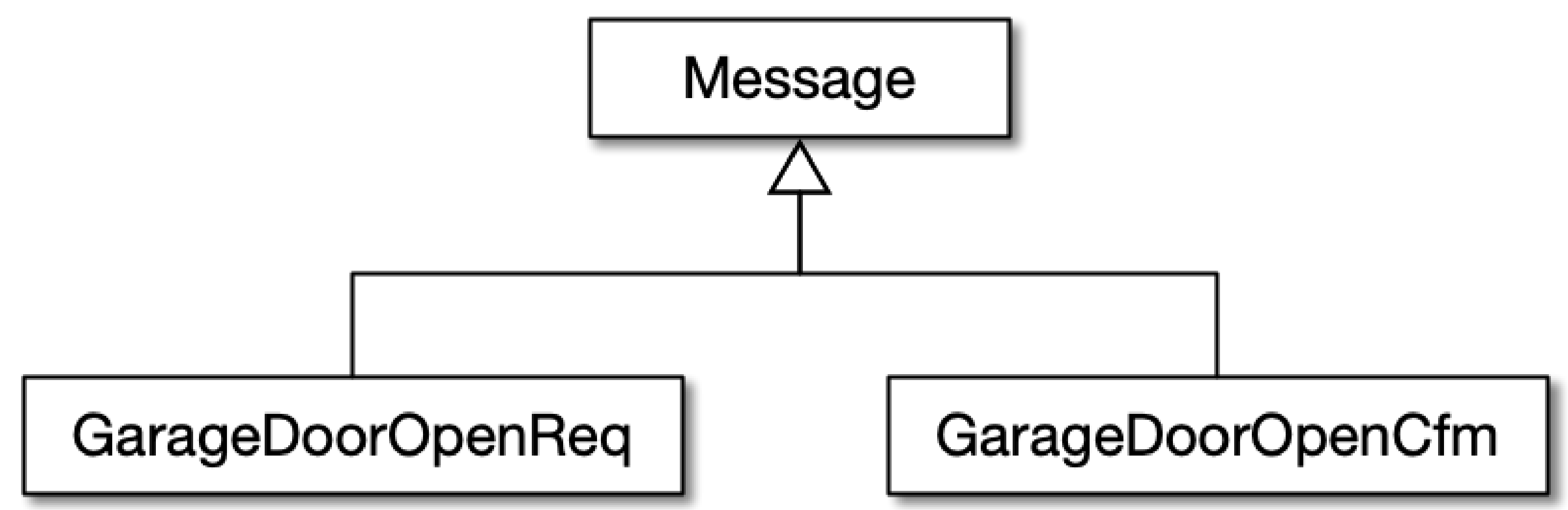
```
enum
{
    ID_GARAGE_DOOR_OPEN_REQ=0,
    ID_GARAGE_DOOR_OPEN_CFM=1,
    ID_XXX=2,
    ID_YYY=3
};
```



DETERMINING REAL MESSAGE TYPE

- How do we convert a Message *to a GarageOpenDoorReq*?
  - Using a special identifier
  - Associating an id with the message

```
enum
{
     ID_GARAGE_DOOR_OPEN_REQ=0,
     ID_GARAGE_DOOR_OPEN_CFM=1,
     ID_XXX=2,
     ID_YYY=3
};
```



`ID_GARAGE_DOOR_OPEN_REQ`* is associated with the type *`GarageDoorOpenReq`
`ID_GARAGE_DOOR_OPEN_CFM`* is associated with the type *`GarageDoorOpenCfm`

DETERMINING REAL MESSAGE TYPE

- How do we convert a Message *to a GarageOpenDoorReq*?
  - Using a special identifier
  - Associating an id with the message

- Need extra variable to denote type
- Switch on value to cast correctly

```
unsigned long id; // Contains a value designating the exact type
Message* msg; // Contains the message

switch(id)
{
  case ID_GARAGE_DOOR_OPEN_REQ:
  {
    GarageDoorOpenReq* gdor = static_cast<GarageDoorOpenReq*>(msg);
    /* ... */
  }
  break;

  case ID_GARAGE_DOOR_OPEN_CFM:
  {
    GarageDoorOpenCfm* gdoc = static_cast<GarageDoorOpenCfm*>(msg);
    /* ... */
  }
  break;
}
```

EMBEDDED COMPILER CONFIGURATIONS

- However certain embedded compilers are compiled without support for RTTI and exception.
  - RTTI - Run Time Type Information
    - Costs in the form of space - *Yes it costs, but what are the consequences?*
  - Exceptions
    - The perception is:
      - Costs in the form of space - *What would the code handling normal errors costs?*
      - It is difficult to do correctly - *Thats certainly correct, but it is not impossible*
      - Errors are not tolerated at all, they must all be found - ***That*** is If you have the time and money, depends on the amount money

EMBEDDED COMPILER CONFIGURATIONS

- Based on these inputs the following requirement is added:
  - It is acknowledged that the use of RTTI will improve program readability, however due to the increase in code size it is denounced
    - Meaning no use of: (in our design)
      - `dynamic_cast<>` - Runtime check whether the cast is permissible or not
      - `typeid()` - Uniquely identify a given object

DETERMINING REAL MESSAGE TYPE

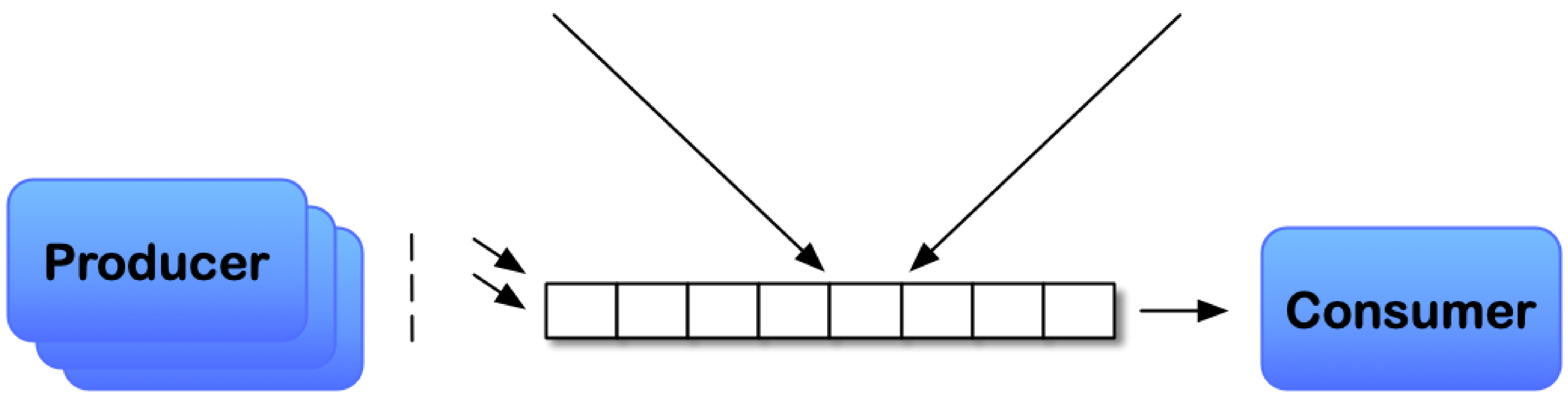
SELECTING APPROACH

- 3 possibilities
  - `dynamic_cast<>`
  - `typeid()`
  - *associating an id*
- Selected
  - *associating an id*
- Why
  - Choice impacted due to compiler considerations
  - Improved readability

MESSAGE QUEUE ELEMENTS

- `id_` is the identifier which is to be send
- `msg_` is the message to be passed

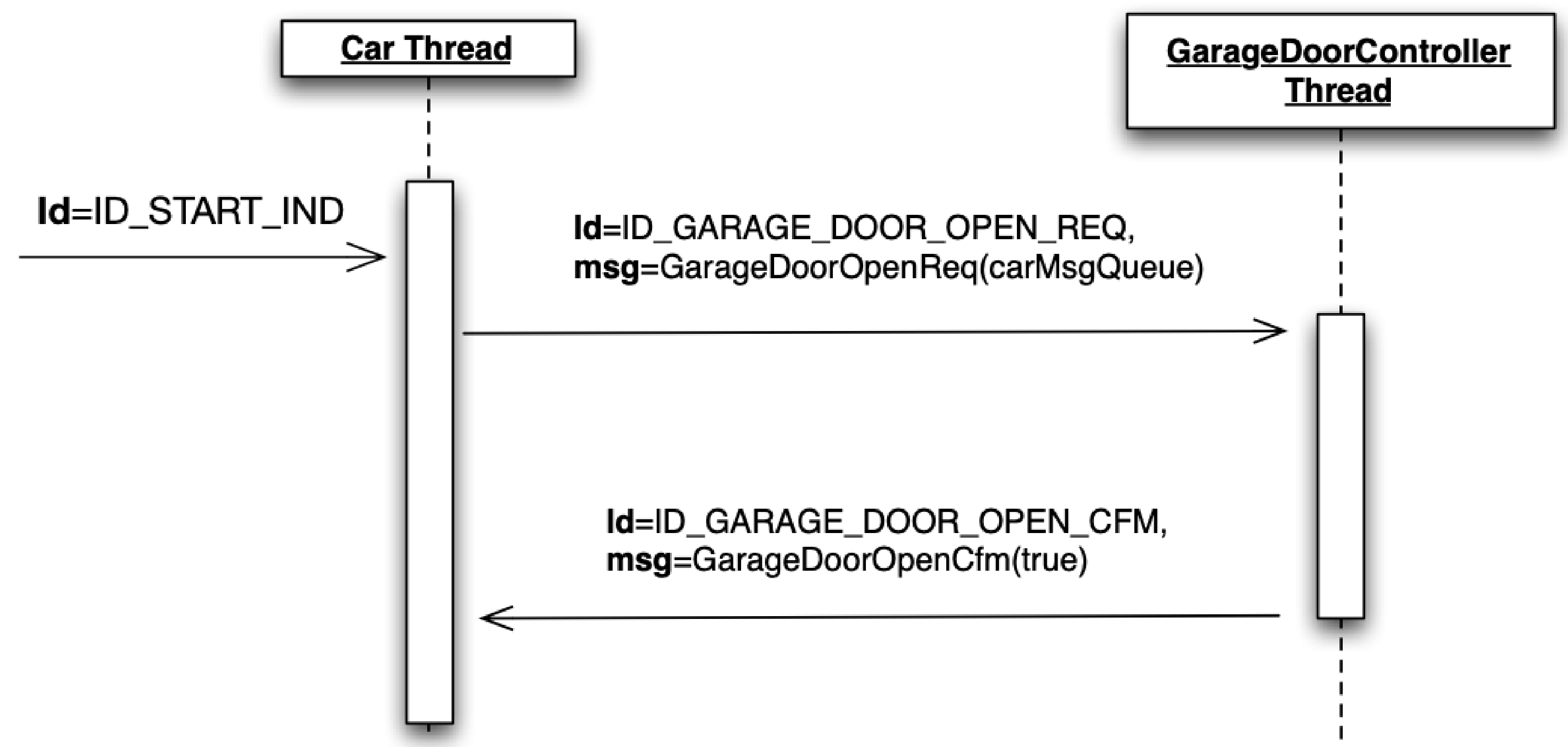
```
struct Item
{
    unsigned long id_;
    Message*      msg_;
}
```



IMPLEMENTATION USAGE

IMPLEMENTATION USAGE

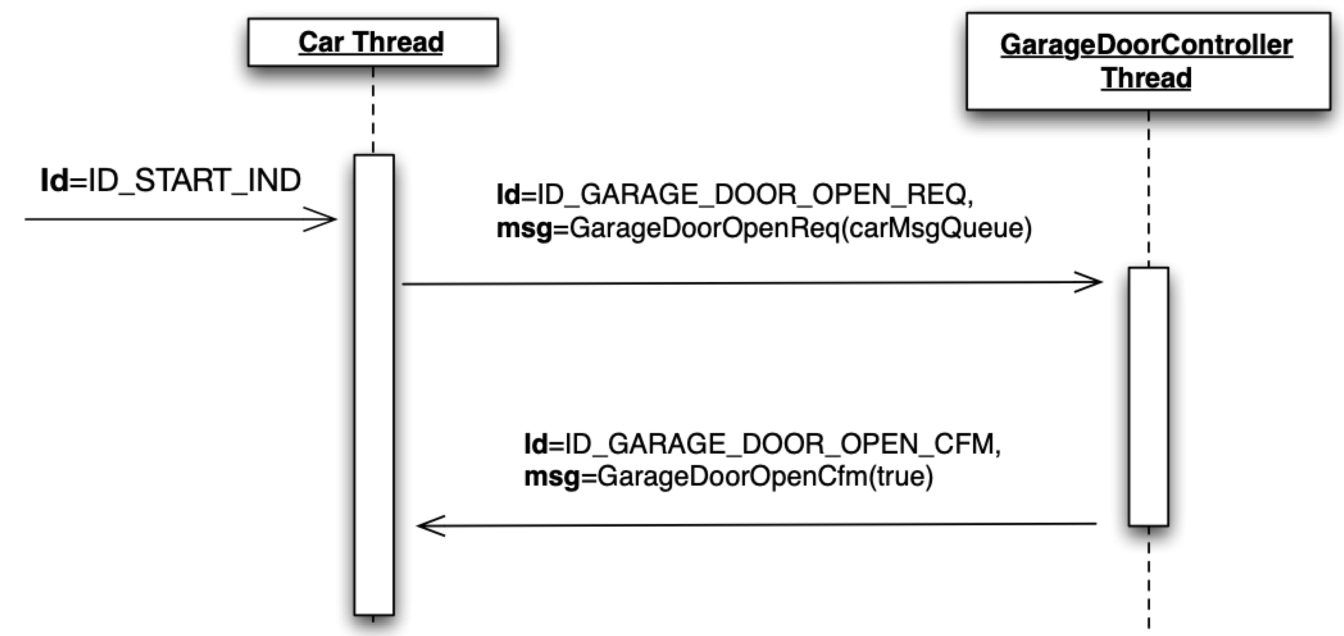
Assume the following



- 3 types of signals
    1. Request - Named XXXReq *(a request requires a confirm)*
    2. Confirm - Named XXXCfm
    3. Indication - Named XXXInd *(purely one-way)*

THE CAR THREAD

- Inspect diagram on the Car part
  - Which message does it receive / need to handle?
  - Incoming messages are, that need to be *handled*
    - `ID_START_IND`
    - `ID_GARAGE_DOOR_OPEN_CFM`
  - Messages sent
    - `ID_GARAGE_DOOR_OPEN_REQ`



THE CAR THREAD IMPLEMENTATION

```
enum { ID_START_IND, ID_GARAGE_DOOR_OPEN_CFM }

pthread_t carId;
MsgQueue carMq;

void carHandler(unsigned id, Message* msg) {
  switch(id) {
    case ID_START_IND:
      carHandleIdStartInd();
      break;
    case ID_GARAGE_DOOR_OPEN_CFM:
      carHandleIdGarageDoorOpenCfm(
        static_cast<GarageDoorOpenCfm*>(msg));
      break;
  }
}

void* car(void*) {
  for(;;) {
    unsigned long id;
    Message* msg = carMq.receive(id);
    carHandler(id, msg);
    delete(msg);
  }
}

void startCarThread() {
  pthread_create(&carId, nullptr, car, nullptr);
  carMq.send(ID_START_IND);
}
```

THE CAR THREAD IMPLEMENTATION

- Create thread

```
enum { ID_START_IND, ID_GARAGE_DOOR_OPEN_CFM }

pthread_t carId;
MsgQueue carMq;

void carHandler(unsigned id, Message* msg) {
  switch(id) {
    case ID_START_IND:
      carHandleIdStartInd();
      break;
    case ID_GARAGE_DOOR_OPEN_CFM:
      carHandleIdGarageDoorOpenCfm(
        static_cast<GarageDoorOpenCfm*>(msg));
      break;
  }
}

void* car(void*) {
  for(;;) {
    unsigned long id;
    Message* msg = carMq.receive(id);
    carHandler(id, msg);
    delete(msg);
  }
}

void startCarThread() {
  pthread_create(&carId, nullptr, car, nullptr);
  carMq.send(ID_START_IND);
}
```

AARHUS UNIVERSITY
AARHUS UNIVERSITY SCHOOL OF ENGINEERING

THE CAR THREAD IMPLEMENTATION

- Create thread
- Event loop - VERY IMPORTANT!!!
  - Idea is
    - Always waiting to receive an event!
    - E.g. thread is blocked waiting

```
enum { ID_START_IND, ID_GARAGE_DOOR_OPEN_CFM }

pthread_t carId;
MsgQueue carMq;

void carHandler(unsigned id, Message* msg) {
  switch(id) {
    case ID_START_IND:
      carHandleIdStartInd();
      break;
    case ID_GARAGE_DOOR_OPEN_CFM:
      carHandleIdGarageDoorOpenCfm(
        static_cast<GarageDoorOpenCfm*>(msg));
      break;
  }
}

void* car(void*) {
  for(;;) {
    unsigned long id;
    Message* msg = carMq.receive(id);
    carHandler(id, msg);
    delete(msg);
  }
}

void startCarThread() {
  pthread_create(&carId, nullptr, car, nullptr);
  carMq.send(ID_START_IND);
}
```

THE CAR THREAD IMPLEMENTATION

- Create thread
- Event loop - VERY IMPORTANT!!!
  - Idea is
    - Always waiting to receive an event!
    - E.g. thread is blocked waiting
- Receive message and id
  - Id denotes which message is received

```cpp
enum { ID_START_IND, ID_GARAGE_DOOR_OPEN_CFM }

pthread_t carId;
MsgQueue carMq;

void carHandler(unsigned id, Message* msg) {
  switch(id) {
    case ID_START_IND:
      carHandleIdStartInd();
      break;
    case ID_GARAGE_DOOR_OPEN_CFM:
      carHandleIdGarageDoorOpenCfm(
        static_cast<GarageDoorOpenCfm*>(msg));
      break;
  }
}

void* car(void*) {
  for(;;) {
    unsigned long id;
    Message* msg = carMq.receive(id);
    carHandler(id, msg);
    delete(msg);
  }
}

void startCarThread() {
  pthread_create(&carId, nullptr, car, nullptr);
  carMq.send(ID_START_IND);
}
```

THE CAR THREAD IMPLEMENTATION

- Create thread
- Event loop - VERY IMPORTANT!!!
  - Idea is
    - Always waiting to receive an event!
    - E.g. thread is blocked waiting
- Receive message and id
  - Id denotes which message is received
- Handle message, also called dispatcher

```
enum { ID_START_IND, ID_GARAGE_DOOR_OPEN_CFM }

pthread_t carId;
MsgQueue carMq;

void carHandler(unsigned id, Message* msg) {
  switch(id) {
    case ID_START_IND:
      carHandleIdStartInd();
      break;
    case ID_GARAGE_DOOR_OPEN_CFM:
      carHandleIdGarageDoorOpenCfm(
        static_cast<GarageDoorOpenCfm*>(msg));
      break;
  }
}

void* car(void*) {
  for(;;) {
    unsigned long id;
    Message* msg = carMq.receive(id);
    carHandler(id, msg);
    delete(msg);
  }
}

void startCarThread() {
  pthread_create(&carId, nullptr, car, nullptr);
  carMq.send(ID_START_IND);
}
```

THE CAR THREAD IMPLEMENTATION

- Create thread
- Event loop - VERY IMPORTANT!!!
  - Idea is
    - Always waiting to receive an event!
    - E.g. thread is blocked waiting
- Receive message and id
  - Id denotes which message is received
- Handle message, also called dispatcher
- In handler - switch on `id`
  - Based on `id`, do stuff
  - Note that sometimes an `id` is enough
    - E.g. the id itself, is *the* information

```
enum { ID_START_IND, ID_GARAGE_DOOR_OPEN_CFM }

pthread_t carId;
MsgQueue carMq;

void carHandler(unsigned id, Message* msg) {
  switch(id) {
    case ID_START_IND:
      carHandleIdStartInd();
      break;
    case ID_GARAGE_DOOR_OPEN_CFM:
      carHandleIdGarageDoorOpenCfm(
        static_cast<GarageDoorOpenCfm*>(msg));
      break;
  }
}

void* car(void*) {
  for(;;) {
    unsigned long id;
    Message* msg = carMq.receive(id);
    carHandler(id, msg);
    delete(msg);
  }
}

void startCarThread() {
  pthread_create(&carId, nullptr, car, nullptr);
  carMq.send(ID_START_IND);
}
```

THE CAR THREAD IMPLEMENTATION

- Create thread
- Event loop - VERY IMPORTANT!!!
  - Idea is
    - Always waiting to receive an event!
    - E.g. thread is blocked waiting
- Receive message and id
  - Id denotes which message is received
- Handle message, also called dispatcher
- In handler - switch on `id`
  - Based on `id`, do stuff
  - Note that sometimes an `id` is enough
    - E.g. the id itself, is *the* information
- Call designated handlers
  - Do not inline the handler's code
    - Becomes messy
    - Loss of overview

```
enum { ID_START_IND, ID_GARAGE_DOOR_OPEN_CFM }

pthread_t carId;
MsgQueue carMq;

void carHandler(unsigned id, Message* msg) {
  switch(id) {
    case ID_START_IND:
      carHandleIdStartInd();
      break;
    case ID_GARAGE_DOOR_OPEN_CFM:
      carHandleIdGarageDoorOpenCfm(
        static_cast<GarageDoorOpenCfm*>(msg));
      break;
  }
}

void* car(void*) {
  for(;;) {
    unsigned long id;
    Message* msg = carMq.receive(id);
    carHandler(id, msg);
    delete(msg);
  }
}

void startCarThread() {
  pthread_create(&carId, nullptr, car, nullptr);
  carMq.send(ID_START_IND);
}
```

THE CAR THREAD IMPLEMENTATION

- Create thread
- Event loop - VERY IMPORTANT!!!
  - Idea is
    - Always waiting to receive an event!
    - E.g. thread is blocked waiting
- Receive message and id
  - Id denotes which message is received
- Handle message, also called dispatcher
- In handler - switch on `id`
  - Based on `id`, do stuff
  - Note that sometimes an `id` is enough
    - E.g. the id itself, is *the* information
- Call designated handlers
  - Do not inline the handler's code
    - Becomes messy
    - Loss of overview
- delete message
  - Remember *sender creates, allocates* and *receiver destroys, deletes*
  - Only receiver knows when processing has completed
  - E.g. implies - messages must be on heap!

```cpp
enum { ID_START_IND, ID_GARAGE_DOOR_OPEN_CFM }

pthread_t carId;
MsgQueue carMq;

void carHandler(unsigned id, Message* msg) {
  switch(id) {
    case ID_START_IND:
      carHandleIdStartInd();
      break;
    case ID_GARAGE_DOOR_OPEN_CFM:
      carHandleIdGarageDoorOpenCfm(
        static_cast<GarageDoorOpenCfm*>(msg));
      break;
  }
}

void* car(void*) {
  for(;;) {
    unsigned long id;
    Message* msg = carMq.receive(id);
    carHandler(id, msg);
    delete(msg);
  }
}

void startCarThread() {
  pthread_create(&carId, nullptr, car, nullptr);
  carMq.send(ID_START_IND);
}
```

THE CAR THREAD IMPLEMENTATION

- Create thread
- Event loop - VERY IMPORTANT!!!
  - Idea is
    - Always waiting to receive an event!
    - E.g. thread is blocked waiting
- Receive message and id
  - Id denotes which message is received
- Handle message, also called dispatcher
- In handler - switch on `id`
  - Based on `id`, do stuff
  - Note that sometimes an `id` is enough
    - E.g. the id itself, is *the* information
- Call designated handlers
  - Do not inline the handler's code
    - Becomes messy
    - Loss of overview
- delete message
  - Remember *sender creates, allocates* and *receiver destroys, deletes*
  - Only receiver knows when processing has completed
  - E.g. implies - messages must be on heap!
- Everything is ready, thread waiting to do stuff - ***lets start***

```
enum { ID_START_IND, ID_GARAGE_DOOR_OPEN_CFM }

pthread_t carId;
MsgQueue carMq;

void carHandler(unsigned id, Message* msg) {
  switch(id) {
    case ID_START_IND:
      carHandleIdStartInd();
      break;
    case ID_GARAGE_DOOR_OPEN_CFM:
      carHandleIdGarageDoorOpenCfm(
        static_cast<GarageDoorOpenCfm*>(msg));
      break;
  }
}

void* car(void*) {
  for(;;) {
    unsigned long id;
    Message* msg = carMq.receive(id);
    carHandler(id, msg);
    delete(msg);
  }
}

void startCarThread() {
  pthread_create(&carId, nullptr, car, nullptr);
  carMq.send(ID_START_IND);
}
```

THE CAR THREAD IMPLEMENTATION

HANDLE START INDICATION

```
struct GarageDoorOpenReq : public Message
{
    MsgQueue* whoIsAskingMq_;
};


void carHandleIdStartInd()
{
  GarageDoorOpenReq* req = new GarageDoorOpenReq;
  req->whoIsAskingMq_ = &carMq;

  garageDoorControllerMq.send(ID_GARAGE_DOOR_OPEN_REQ, req);
}
```

THE CAR THREAD IMPLEMENTATION

HANDLE START INDICATION

- Message GarageDoorOpenReq owned by service provider
- Id for Message GarageDoorOpenReq owned by receiver
  - But here used by `Car`

```
struct GarageDoorOpenReq : public Message
{
    MsgQueue* whoIsAskingMq_;
};


void carHandleIdStartInd()
{
  GarageDoorOpenReq* req = new GarageDoorOpenReq;
  req->whoIsAskingMq_ = &carMq;

  garageDoorControllerMq.send(ID_GARAGE_DOOR_OPEN_REQ, req);
}
```

THE CAR THREAD IMPLEMENTATION

HANDLE START INDICATION

- Message GarageDoorOpenReq owned by service provider
- Id for Message GarageDoorOpenReq owned by receiver
  - But here used by `Car`
- Idea
  - Upon receiving start indication in car thread
    - Send open request to garage door controller

```
struct GarageDoorOpenReq : public Message
{
    MsgQueue* whoIsAskingMq_;
};


void carHandleIdStartInd()
{
  GarageDoorOpenReq* req = new GarageDoorOpenReq;
  req->whoIsAskingMq_ = &carMq;

  garageDoorControllerMq.send(ID_GARAGE_DOOR_OPEN_REQ, req);
}
```

THE CAR THREAD IMPLEMENTATION

HANDLE START INDICATION

- Message GarageDoorOpenReq owned by service provider
- Id for Message GarageDoorOpenReq owned by receiver
  - But here used by `Car`
- Idea
  - Upon receiving start indication in car thread
    - Send open request to garage door controller
- Create request

```
struct GarageDoorOpenReq : public Message
{
    MsgQueue* whoIsAskingMq_;
};


void carHandleIdStartInd()
{
  GarageDoorOpenReq* req = new GarageDoorOpenReq;
  req->whoIsAskingMq_ = &carMq;

  garageDoorControllerMq.send(ID_GARAGE_DOOR_OPEN_REQ, req);
}
```

THE CAR THREAD IMPLEMENTATION

HANDLE START INDICATION

- Message GarageDoorOpenReq owned by service provider
- Id for Message GarageDoorOpenReq owned by receiver
  - But here used by `Car`
- Idea
  - Upon receiving start indication in car thread
    - Send open request to garage door controller
- Create request
- Tell who is asking
  - In this case its a specific `Car`
  - All communication is via *message queues*!!!

```
struct GarageDoorOpenReq : public Message
{
    MsgQueue* whoIsAskingMq_;
};


void carHandleIdStartInd()
{
  GarageDoorOpenReq* req = new GarageDoorOpenReq;
  req->whoIsAskingMq_ = &carMq;

  garageDoorControllerMq.send(ID_GARAGE_DOOR_OPEN_REQ, req);
}
```

THE CAR THREAD IMPLEMENTATION

HANDLE START INDICATION

- Message GarageDoorOpenReq owned by service provider
- Id for Message GarageDoorOpenReq owned by receiver
  - But here used by `Car`
- Idea
  - Upon receiving start indication in car thread
    - Send open request to garage door controller
- Create request
- Tell who is asking
  - In this case its a specific `Car`
  - All communication is via *message queues*!!!
- Send the message to the garage door controller

```
struct GarageDoorOpenReq : public Message
{
    MsgQueue* whoIsAskingMq_;
};


void carHandleIdStartInd()
{
  GarageDoorOpenReq* req = new GarageDoorOpenReq;
  req->whoIsAskingMq_ = &carMq;

  garageDoorControllerMq.send(ID_GARAGE_DOOR_OPEN_REQ, req);
}
```

THE CAR THREAD IMPLEMENTATION

HANDLE GARAGE DOOR OPEN CFM

```
struct GarageDoorOpenCfm : public Message
{
  bool result_;
};


void carHandleIdGarageDoorOpenCfm(GarageDoorOpenCfm* cfm)
{
  if(cfm->result_)
  {
    driveIntoParkingLot();
  }
}
```

THE CAR THREAD IMPLEMENTATION

HANDLE GARAGE DOOR OPEN CFM

- Message GarageDoorOpenCfm owned by service provider
- Id for Message GarageDoorOpenCfm owned by receiver
    - Receiver is `Car`

```
struct GarageDoorOpenCfm : public Message
{
  bool result_;
};


void carHandleIdGarageDoorOpenCfm(GarageDoorOpenCfm* cfm)
{
  if(cfm->result_)
  {
    driveIntoParkingLot();
  }
}
```

THE CAR THREAD IMPLEMENTATION

HANDLE GARAGE DOOR OPEN CFM

- Message GarageDoorOpenCfm owned by service provider
- Id for Message GarageDoorOpenCfm owned by receiver
  - Receiver is `Car`
- Idea
  - Upon receiving confirm check to see if its ok to enter

```
struct GarageDoorOpenCfm : public Message
{
  bool result_;
};


void carHandleIdGarageDoorOpenCfm(GarageDoorOpenCfm* cfm)
{
  if(cfm->result_)
  {
    driveIntoParkingLot();
  }
}
```

THE CAR THREAD IMPLEMENTATION

HANDLE GARAGE DOOR OPEN CFM

- Message GarageDoorOpenCfm owned by service provider
- Id for Message GarageDoorOpenCfm owned by receiver
  - Receiver is `Car`
- Idea
  - Upon receiving confirm check to see if its ok to enter
- Perform the check

```
struct GarageDoorOpenCfm : public Message
{
  bool result_;
};


void carHandleIdGarageDoorOpenCfm(GarageDoorOpenCfm* cfm)
{
  if(cfm->result_)
  {
    driveIntoParkingLot();
  }
}
```

THE CAR THREAD IMPLEMENTATION

HANDLE GARAGE DOOR OPEN CFM

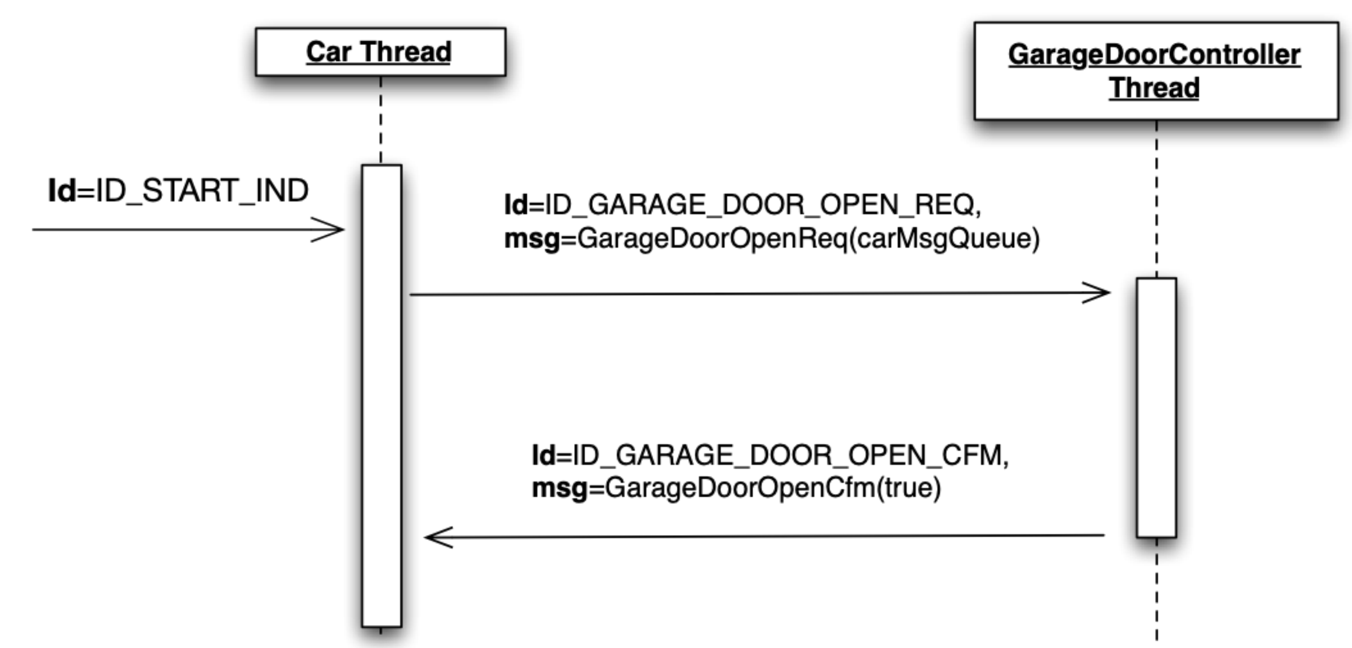
- Message GarageDoorOpenCfm owned by service provider
- Id for Message GarageDoorOpenCfm owned by receiver
  - Receiver is `Car`
- Idea
  - Upon receiving confirm check to see if its ok to enter
- Perform the check
- Enter car park...

```
struct GarageDoorOpenCfm : public Message
{
  bool result_;
};


void carHandleIdGarageDoorOpenCfm(GarageDoorOpenCfm* cfm)
{
  if(cfm->result_)
  {
    driveIntoParkingLot();
  }
}
```

THE GARAGE DOOR CONTROLLER THREAD

- Inspect diagram on the receiver part
  - Which message does it receive / need to handle?
  - Incoming messages are, that need to be *handled*
    - `ID_GARAGE_DOOR_OPEN_REQ`
  - Messages sent
    - `ID_GARAGE_DOOR_OPEN_CFM`



THE GARAGE DOOR CONTROLLER THREAD IMPLEMENTATION

```
enum { ID_GARAGE_DOOR_OPEN_REQ }

pthread_t garageDoorControllerId;
MsgQueue garageDoorControllerMq;

void garageDoorControllerHandler(unsigned id, Message* msg) {
  switch(id) {
    case ID_GARAGE_DOOR_OPEN_REQ:
      garageDoorControllerHandleIdGarageDoorOpenReq(
        static_cast<GarageDoorOpenReq*>(msg));
      break;
  }
}

void* garageDoorController(void*) {
  for(;;) {
    unsigned long id;
    Message* msg = garageDoorControllerMq.receive(id);
    garageDoorControllerHandler(id, msg);
    delete(msg);
  }
}

void startGarageDoorControllerThread() {
  pthread_create(&garageDoorControllerId, nullptr,
                 garageDoorController, nullptr);
}
```

THE GARAGE DOOR CONTROLLER THREAD IMPLEMENTATION

- Create thread

```
enum { ID_GARAGE_DOOR_OPEN_REQ }

pthread_t garageDoorControllerId;
MsgQueue garageDoorControllerMq;

void garageDoorControllerHandler(unsigned id, Message* msg) {
  switch(id) {
    case ID_GARAGE_DOOR_OPEN_REQ:
      garageDoorControllerHandleIdGarageDoorOpenReq(
        static_cast<GarageDoorOpenReq*>(msg));
      break;
  }
}

void* garageDoorController(void*) {
  for(;;) {
    unsigned long id;
    Message* msg = garageDoorControllerMq.receive(id);
    garageDoorControllerHandler(id, msg);
    delete(msg);
  }
}

void startGarageDoorControllerThread() {
  pthread_create(&garageDoorControllerId, nullptr,
                 garageDoorController, nullptr);
}
```

THE GARAGE DOOR CONTROLLER THREAD IMPLEMENTATION

- Create thread
- Event loop - VERY IMPORTANT!!!
  - Idea is
    - Always waiting to receive an event!
    - E.g. thread is blocked waiting

```
enum { ID_GARAGE_DOOR_OPEN_REQ }

pthread_t garageDoorControllerId;
MsgQueue garageDoorControllerMq;

void garageDoorControllerHandler(unsigned id, Message* msg) {
  switch(id) {
    case ID_GARAGE_DOOR_OPEN_REQ:
      garageDoorControllerHandleIdGarageDoorOpenReq(
        static_cast<GarageDoorOpenReq*>(msg));
      break;
  }
}

void* garageDoorController(void*) {
  for(;;) {
    unsigned long id;
    Message* msg = garageDoorControllerMq.receive(id);
    garageDoorControllerHandler(id, msg);
    delete(msg);
  }
}

void startGarageDoorControllerThread() {
  pthread_create(&garageDoorControllerId, nullptr,
                 garageDoorController, nullptr);
}
```

THE GARAGE DOOR CONTROLLER THREAD IMPLEMENTATION

- Create thread
- Event loop - VERY IMPORTANT!!!
  - Idea is
    - Always waiting to receive an event!
    - E.g. thread is blocked waiting
- Receive message and id
  - Id denotes which message is received

```
enum { ID_GARAGE_DOOR_OPEN_REQ }

pthread_t garageDoorControllerId;
MsgQueue garageDoorControllerMq;

void garageDoorControllerHandler(unsigned id, Message* msg) {
  switch(id) {
    case ID_GARAGE_DOOR_OPEN_REQ:
      garageDoorControllerHandleIdGarageDoorOpenReq(
        static_cast<GarageDoorOpenReq*>(msg));
      break;
  }
}

void* garageDoorController(void*) {
  for(;;) {
    unsigned long id;
    Message* msg = garageDoorControllerMq.receive(id);
    garageDoorControllerHandler(id, msg);
    delete(msg);
  }
}

void startGarageDoorControllerThread() {
  pthread_create(&garageDoorControllerId, nullptr,
                 garageDoorController, nullptr);
}
```

THE GARAGE DOOR CONTROLLER THREAD IMPLEMENTATION

- Create thread
- Event loop - VERY IMPORTANT!!!
  - Idea is
    - Always waiting to receive an event!
    - E.g. thread is blocked waiting
- Receive message and id
  - Id denotes which message is received
- Handle message, also called dispatcher

```
enum { ID_GARAGE_DOOR_OPEN_REQ }

pthread_t garageDoorControllerId;
MsgQueue garageDoorControllerMq;

void garageDoorControllerHandler(unsigned id, Message* msg) {
  switch(id) {
    case ID_GARAGE_DOOR_OPEN_REQ:
      garageDoorControllerHandleIdGarageDoorOpenReq(
        static_cast<GarageDoorOpenReq*>(msg));
      break;
  }
}

void* garageDoorController(void*) {
  for(;;) {
    unsigned long id;
    Message* msg = garageDoorControllerMq.receive(id);
    garageDoorControllerHandler(id, msg);
    delete(msg);
  }
}

void startGarageDoorControllerThread() {
  pthread_create(&garageDoorControllerId, nullptr,
                 garageDoorController, nullptr);
}
```

THE GARAGE DOOR CONTROLLER THREAD IMPLEMENTATION

- Create thread
- Event loop - VERY IMPORTANT!!!
  - Idea is
    - Always waiting to receive an event!
    - E.g. thread is blocked waiting
- Receive message and id
  - Id denotes which message is received
- Handle message, also called dispatcher
- In handler - switch on `id`
  - Based on `id`, do stuff
  - Note that sometimes an `id` is enough
    - E.g. the id itself, is *the* information

```
enum { ID_GARAGE_DOOR_OPEN_REQ }

pthread_t garageDoorControllerId;
MsgQueue garageDoorControllerMq;

void garageDoorControllerHandler(unsigned id, Message* msg) {
  switch(id) {
    case ID_GARAGE_DOOR_OPEN_REQ:
      garageDoorControllerHandleIdGarageDoorOpenReq(
        static_cast<GarageDoorOpenReq*>(msg));
      break;
  }
}

void* garageDoorController(void*) {
  for(;;) {
    unsigned long id;
    Message* msg = garageDoorControllerMq.receive(id);
    garageDoorControllerHandler(id, msg);
    delete(msg);
  }
}

void startGarageDoorControllerThread() {
  pthread_create(&garageDoorControllerId, nullptr,
                 garageDoorController, nullptr);
}
```

THE GARAGE DOOR CONTROLLER THREAD IMPLEMENTATION

- Create thread
- Event loop - VERY IMPORTANT!!!
  - Idea is
    - Always waiting to receive an event!
    - E.g. thread is blocked waiting
- Receive message and id
  - Id denotes which message is received
- Handle message, also called dispatcher
- In handler - switch on `id`
  - Based on `id`, do stuff
  - Note that sometimes an `id` is enough
    - E.g. the id itself, is *the* information
- Call designated handlers
  - Do not inline the handler's code
    - Becomes messy
    - Loss of overview

```
enum { ID_GARAGE_DOOR_OPEN_REQ }

pthread_t garageDoorControllerId;
MsgQueue garageDoorControllerMq;

void garageDoorControllerHandler(unsigned id, Message* msg) {
  switch(id) {
    case ID_GARAGE_DOOR_OPEN_REQ:
      garageDoorControllerHandleIdGarageDoorOpenReq(
        static_cast<GarageDoorOpenReq*>(msg));
      break;
  }
}

void* garageDoorController(void*) {
  for(;;) {
    unsigned long id;
    Message* msg = garageDoorControllerMq.receive(id);
    garageDoorControllerHandler(id, msg);
    delete(msg);
  }
}

void startGarageDoorControllerThread() {
  pthread_create(&garageDoorControllerId, nullptr,
                 garageDoorController, nullptr);
}
```

THE GARAGE DOOR CONTROLLER THREAD IMPLEMENTATION

- Create thread
- Event loop - VERY IMPORTANT!!!
  - Idea is
    - Always waiting to receive an event!
    - E.g. thread is blocked waiting
- Receive message and id
  - Id denotes which message is received
- Handle message, also called dispatcher
- In handler - switch on `id`
  - Based on `id`, do stuff
  - Note that sometimes an `id` is enough
    - E.g. the id itself, is *the* information
- Call designated handlers
  - Do not inline the handler's code
    - Becomes messy
    - Loss of overview
- delete message
  - Remember sender creates / allocates receiver deletes
  - Only receiver knows when processing has completed
  - E.g. implies - messages must be on heap!

```
enum { ID_GARAGE_DOOR_OPEN_REQ }

pthread_t garageDoorControllerId;
MsgQueue garageDoorControllerMq;

void garageDoorControllerHandler(unsigned id, Message* msg) {
  switch(id) {
    case ID_GARAGE_DOOR_OPEN_REQ:
      garageDoorControllerHandleIdGarageDoorOpenReq(
        static_cast<GarageDoorOpenReq*>(msg));
      break;
  }
}

void* garageDoorController(void*) {
  for(;;) {
    unsigned long id;
    Message* msg = garageDoorControllerMq.receive(id);
    garageDoorControllerHandler(id, msg);
    delete(msg);
  }
}

void startGarageDoorControllerThread() {
  pthread_create(&garageDoorControllerId, nullptr,
                 garageDoorController, nullptr);
}
```

THE GARAGE DOOR CONTROLLER THREAD IMPLEMENTATION

HANDLE GARAGE DOOR OPEN REQ

```cpp
struct GarageDoorOpenReq : public Message
{
    MsgQueue* whoIsAskingMq_;
};

struct GarageDoorOpenCfm : public Message
{
  bool result_;
};


void garageDoorControllerHandleIdGarageDoorOpenReq
                              (GarageDoorOpenReq* req)
{
  GarageDoorOpenCfm* cfm = new GarageDoorOpenCfm;
  cfm->result_ = openGarageDoor();

  req->whoIsAskingMq_->send(ID_GARAGE_DOOR_OPEN_CFM, cfm);

}
```

THE GARAGE DOOR CONTROLLER THREAD IMPLEMENTATION

HANDLE GARAGE DOOR OPEN REQ

- Message GarageDoorOpenReq owned by service provider
  - Id for Message GarageDoorOpenReq owned by receiver

```
struct GarageDoorOpenReq : public Message
{
    MsgQueue* whoIsAskingMq_;
};

struct GarageDoorOpenCfm : public Message
{
  bool result_;
};


void garageDoorControllerHandleIdGarageDoorOpenReq
                             (GarageDoorOpenReq* req)
{
  GarageDoorOpenCfm* cfm = new GarageDoorOpenCfm;
  cfm->result_ = openGarageDoor();

  req->whoIsAskingMq_->send(ID_GARAGE_DOOR_OPEN_CFM, cfm);

}
```

THE GARAGE DOOR CONTROLLER THREAD IMPLEMENTATION

HANDLE GARAGE DOOR OPEN REQ

- Message GarageDoorOpenReq owned by service provider
  - Id for Message GarageDoorOpenReq owned by receiver
- Message GarageDoorOpenCfm owned by service provider
  - Id for Message GarageDoorOpenCfm owned by receiver
    - In this case it's `Car`

```
struct GarageDoorOpenReq : public Message
{
    MsgQueue* whoIsAskingMq_;
};

struct GarageDoorOpenCfm : public Message
{
  bool result_;
};


void garageDoorControllerHandleIdGarageDoorOpenReq
                              (GarageDoorOpenReq* req)
{
  GarageDoorOpenCfm* cfm = new GarageDoorOpenCfm;
  cfm->result_ = openGarageDoor();

  req->whoIsAskingMq_->send(ID_GARAGE_DOOR_OPEN_CFM, cfm);

}
```

THE GARAGE DOOR CONTROLLER THREAD IMPLEMENTATION

HANDLE GARAGE DOOR OPEN REQ

- Message GarageDoorOpenReq owned by service provider
  - Id for Message GarageDoorOpenReq owned by receiver
- Message GarageDoorOpenCfm owned by service provider
  - Id for Message GarageDoorOpenCfm owned by receiver
    - In this case it's `Car`
- Idea
  - Upon receiving a request open door and send confirm

```
struct GarageDoorOpenReq : public Message
{
    MsgQueue* whoIsAskingMq_;
};

struct GarageDoorOpenCfm : public Message
{
  bool result_;
};


void garageDoorControllerHandleIdGarageDoorOpenReq
                             (GarageDoorOpenReq* req)
{
  GarageDoorOpenCfm* cfm = new GarageDoorOpenCfm;
  cfm->result_ = openGarageDoor();

  req->whoIsAskingMq_->send(ID_GARAGE_DOOR_OPEN_CFM, cfm);

}
```

THE GARAGE DOOR CONTROLLER THREAD IMPLEMENTATION

HANDLE GARAGE DOOR OPEN REQ

- Message GarageDoorOpenReq owned by service provider
  - Id for Message GarageDoorOpenReq owned by receiver
- Message GarageDoorOpenCfm owned by service provider
  - Id for Message GarageDoorOpenCfm owned by receiver
    - In this case it's `Car`
- Idea
  - Upon receiving a request open door and send confirm
- Allocate confirm

```
struct GarageDoorOpenReq : public Message
{
    MsgQueue* whoIsAskingMq_;
};

struct GarageDoorOpenCfm : public Message
{
  bool result_;
};


void garageDoorControllerHandleIdGarageDoorOpenReq
                            (GarageDoorOpenReq* req)
{
  GarageDoorOpenCfm* cfm = new GarageDoorOpenCfm;
  cfm->result_ = openGarageDoor();

  req->whoIsAskingMq_->send(ID_GARAGE_DOOR_OPEN_CFM, cfm);

}
```

THE GARAGE DOOR CONTROLLER THREAD IMPLEMENTATION

HANDLE GARAGE DOOR OPEN REQ

- Message GarageDoorOpenReq owned by service provider
  - Id for Message GarageDoorOpenReq owned by receiver
- Message GarageDoorOpenCfm owned by service provider
  - Id for Message GarageDoorOpenCfm owned by receiver
    - In this case it's `Car`
- Idea
  - Upon receiving a request open door and send confirm
- Allocate confirm
- Open door and save responds in confirm

```
struct GarageDoorOpenReq : public Message
{
    MsgQueue* whoIsAskingMq_;
};

struct GarageDoorOpenCfm : public Message
{
  bool result_;
};


void garageDoorControllerHandleIdGarageDoorOpenReq
                              (GarageDoorOpenReq* req)
{
  GarageDoorOpenCfm* cfm = new GarageDoorOpenCfm;
  cfm->result_ = openGarageDoor();

  req->whoIsAskingMq_->send(ID_GARAGE_DOOR_OPEN_CFM, cfm);

}
```

THE GARAGE DOOR CONTROLLER THREAD IMPLEMENTATION

HANDLE GARAGE DOOR OPEN REQ

- Message GarageDoorOpenReq owned by service provider
  - Id for Message GarageDoorOpenReq owned by receiver
- Message GarageDoorOpenCfm owned by service provider
  - Id for Message GarageDoorOpenCfm owned by receiver
    - In this case it's `Car`
- Idea
  - Upon receiving a request open door and send confirm
- Allocate confirm
- Open door and save responds in confirm
- Send the confirm to the requester
  - Note that the `mq_` contains a pointer to the requester's MsgQueue

```
struct GarageDoorOpenReq : public Message
{
    MsgQueue* whoIsAskingMq_;
};

struct GarageDoorOpenCfm : public Message
{
  bool result_;
};


void garageDoorControllerHandleIdGarageDoorOpenReq
                              (GarageDoorOpenReq* req)
{
  GarageDoorOpenCfm* cfm = new GarageDoorOpenCfm;
  cfm->result_ = openGarageDoor();

  req->whoIsAskingMq_->send(ID_GARAGE_DOOR_OPEN_CFM, cfm);

}
```

DESIGN & IMPLEMENTATION POINTS

- Event loop
  - Only one call to receive() and one call to handle() in the loop
    - Multiple calls - you got the concept all wrong!!!
- Message parsing
  - Sender allocates
  - Receiver deletes
    - Is the only part that knows when the message is no more needed
- In this example common variables are global
  - Real world - not necessarily
    - Extra parameter in all functions with a pointer to a common struct
      - E.g. Object Oriented C

DESIGN & IMPLEMENTATION POINTS

- In the previous example we only switch on id
  - States can be handled just as well

```
void* thread(void* gateData)
{
  Gate* gate =
    static_cast<Gate*>(gateData);

  while(gate->running)
  {
    unsigned long id;
    Message* msg =
      gate->mq_->receive(id);

    gateHandler(gate, id, msg);
    delete(msg);

  }
}
```

```
struct Gate { /* ... */ };

void gateHandleStClosedIdOpenReq(Gate* gate, OpenReq* req) {}

void gateHandleStClosed(Gate* gate, unsigned long id, Message* msg) {
  switch(gate->state) {
    case ID_OPEN_REQ:
      gateHandleStClosedIdOpenReq(gate, static_cast<OpenReq*>(req));
      break;
    /* ... */
  }
}

void gateHandler(Gate* gate, unsigned long id, Message* msg) {
  switch(gate->state) {
    case ST_CLOSED:
      gateHandleStClosed(gate, id, msg);
      break;

    case ST_OPENED:
      gateHandleStOpened(gate, id, msg);
      break;
  }
}
```

CONSEQUENCES

CONSEQUENCES

NEGATIVE

- No silver bullet by far.
- In a performance perspective not necessarily the best solution.
- Mostly to do with a-synchronicity, meaning that you are not guaranteed an answer but have to have some form of timeout.

CONSEQUENCES

NEGATIVE

- No silver bullet by far.
- In a performance perspective not necessarily the best solution.
- Mostly to do with a-synchronicity, meaning that you are not guaranteed an answer but have to have some form of timeout.

POSITIVE

- Does not inhibit misuse, but signifies a route that makes it “more” clear, as to what is to happen when.
- Reduces the need for critical sections e.g. mutexes and semaphores.
- Not blocked on a conditional/mutex while waiting

SUMMARY

SUMMARY

- What is it we in fact have done?
  - Entered the Event Driven Programming (EDP) paradigm

SUMMARY

- What is it we in fact have done?
  - Entered the Event Driven Programming (EDP) paradigm
- What is EDP?
  - Reaction based programming
    - Interrupts from sensors, key input, controller directives etc.
  - Multiple correct paths through the code
  - For more complex code structure where the code is not stateless state machines are the solution - Finite State Machine (FSM)