Embedded Software

Inter-thread communication

IHA | ENGINEERING COLLEGE OF AARHUS

Agenda

- Communication design challenge
- The Message Queue - Conceptual
- Consequences

Message Queue

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.

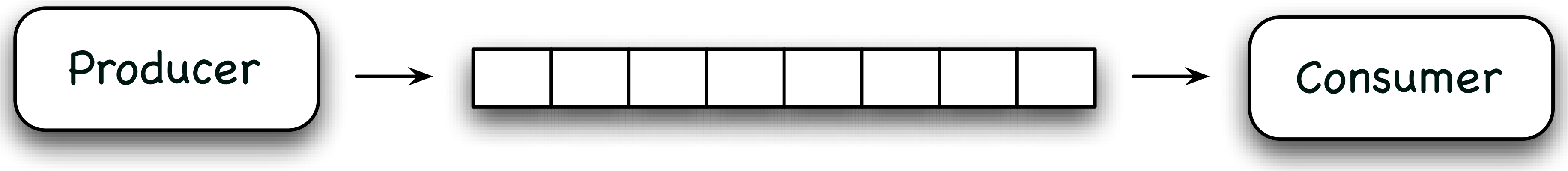
Communication design challenges

- 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.
- ***Consequence***
  - A design challenge ensuring that no deadlocks or timing issues exist
  - Readability easily becomes an issue too

The Message Queue - Conceptual

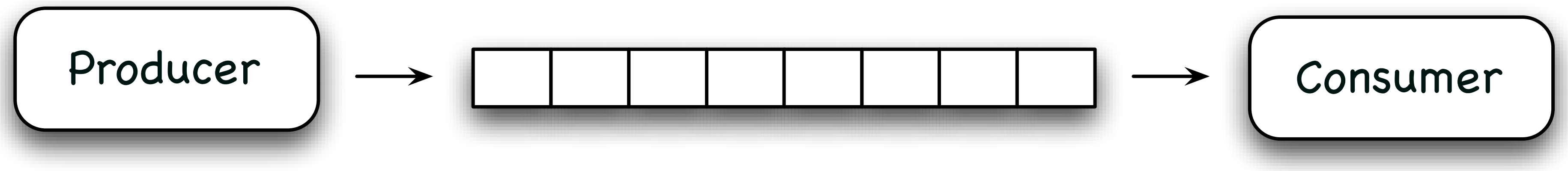
- We want an approach where
  - *all* processing within a thread must not require locking
  - however *other* threads must be able to pass control and/or data to a specific thread via some mechanism.
  - *multiple* threads may concurrently decide to pass such control and/or data

Resembles the “Producer & Consumer problem”

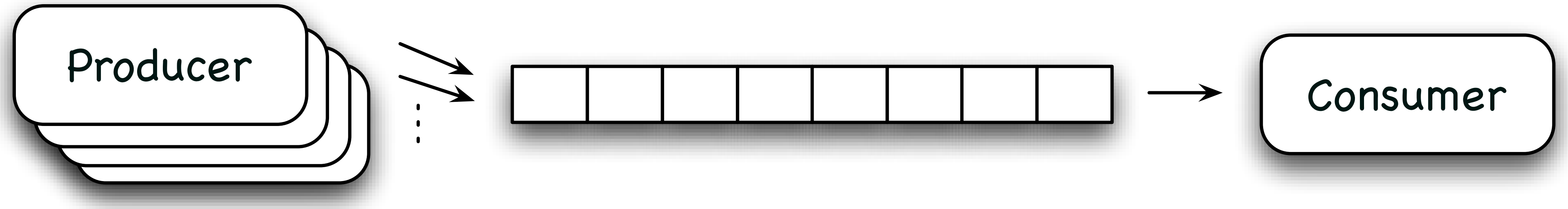


- The producer-consumer problem
  - A producer thread produces buffer items
  - A consumer thread consumes them
- Applied to our problem we get

Resembles the “Producer & Consumer problem”



- The producer-consumer problem
  - A producer thread produces buffer items
  - A consumer thread consumes them
- Applied to our problem we get



Further requirements for the Message Queue

Further requirements for the 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

Further requirements for the 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

Further requirements for the 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?*

Further requirements for the 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?

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

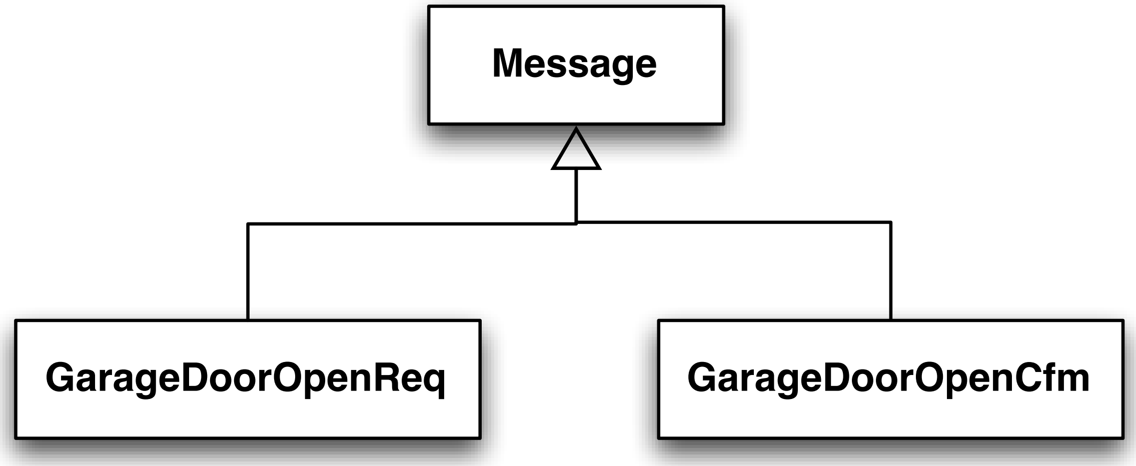
What is the structure of the information to pass around?

- void* or simple array of bytes
  - Can contain anything
  - No type information - No type-safety
- template based
  - Depends on the implementation, can be good solution
  - Type-safety

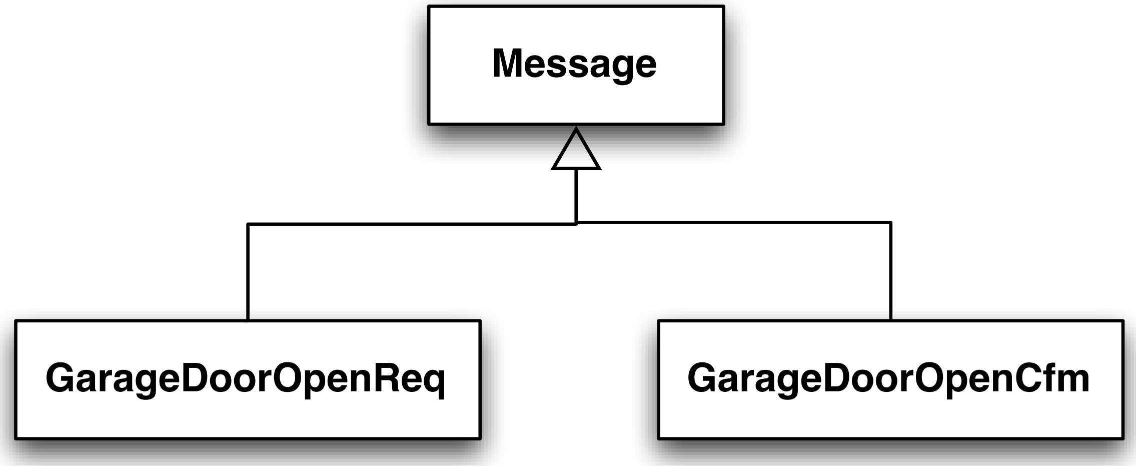
What is the structure of the information to pass around?

- void* or simple array of bytes
  - Can contain anything
  - No type information - No type-safety
- template based
  - Depends on the implementation, can be good solution
  - Type-safety
- Inheritance
  - Simple and extended via sub-classing
  - Type-safety
  - Might incur overhead

Using Message as a base class



Using Message as a base class

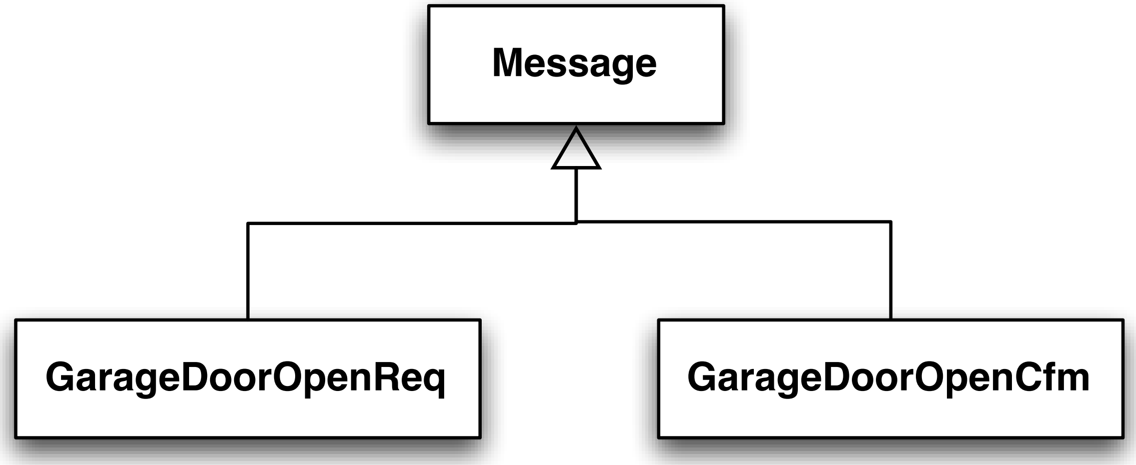


```
class Message
{
public:
   virtual ~Message(){}
};
```

```
void handler(Message* msg)
{
   // Which message???
   if(dynamic_cast<GarageDoorOpenReq*>(Msg) != NULL)
   {
         // This is the one, handle it
   }
   else if(... != NULL)
   {
         // Some other message, handle it
   }
   else if(... != NULL)
   {
         // Some other message, handle it
   }
   ...
}
```

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

Using Message as a base class



Message

GarageDoorOpenReq

GarageDoorOpenCfm

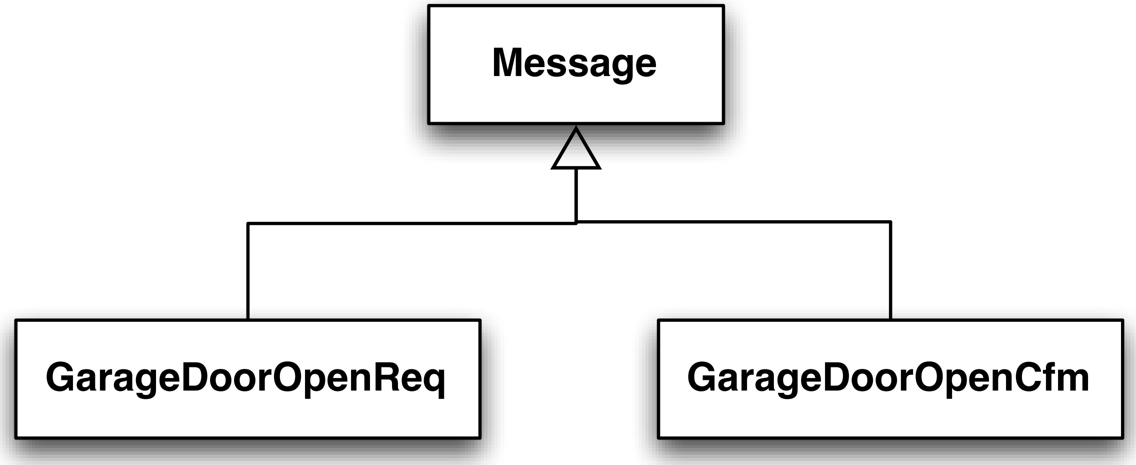
```
class Message
{
public:
   virtual ~Message(){}
};
```

```
void handler(Message* msg)
{
   // Which message???
   if(dynamic_cast<GarageDoorOpenReq*>(Msg) != NULL)
   {
         // This is the one, handle it
   }
   else if(... != NULL)
   {
         // Some other message, handle it
   }
   else if(... != NULL)
   {
         // Some other message, handle it
   }
   ...
}
```

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

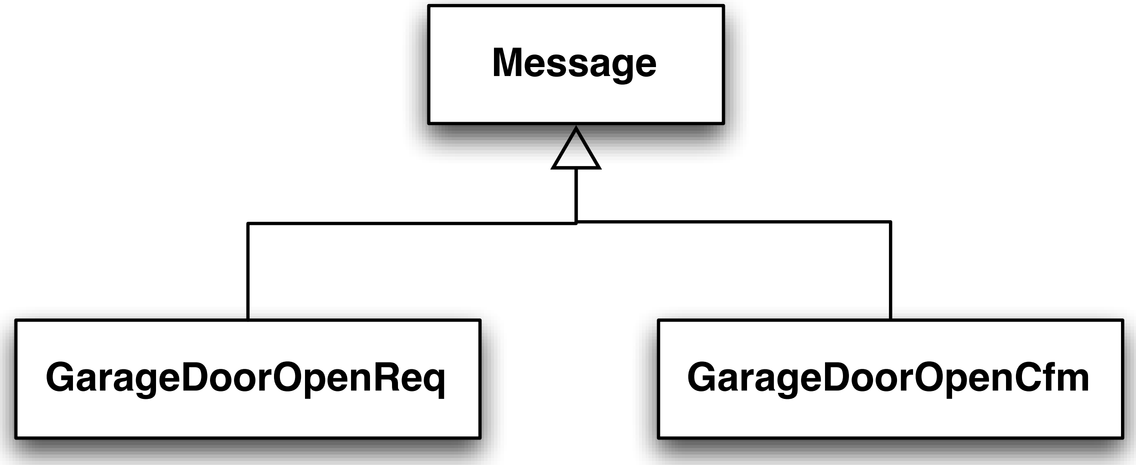
NOT GOOD ENOUGH

An identifier to designate which child it is



An identifier to designate which child it is

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

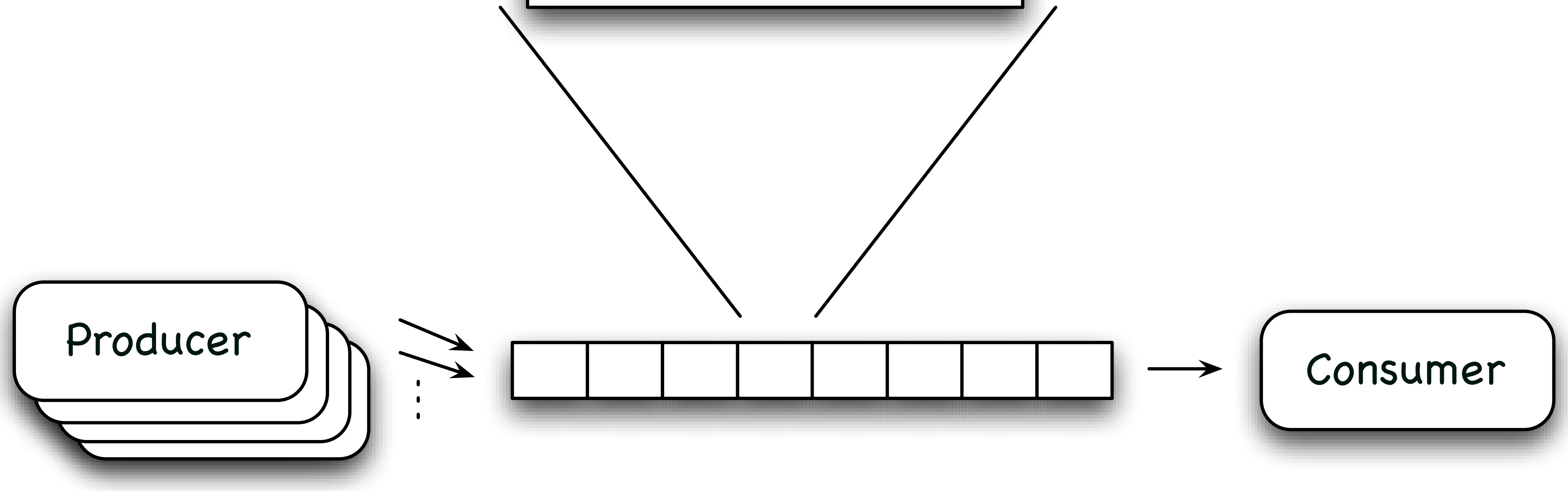


```
void handler(Message* msg, size_t id)
{
   switch(id)
   {
      case ID_GARAGE_DOOR_OPEN_REQ:
         GarageDoorOpenReq* gdor = dynamic_cast<GarageDoorOpenReq*>(Msg);
         // Do stuff - call handler
         break;

      case ID_XXX:
         // ...
         break;

      default:
         std::cout << "Argh, unknown identifier, what to do???" << std::endl;
   };
}
```

Choice of item in MsgQueue

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

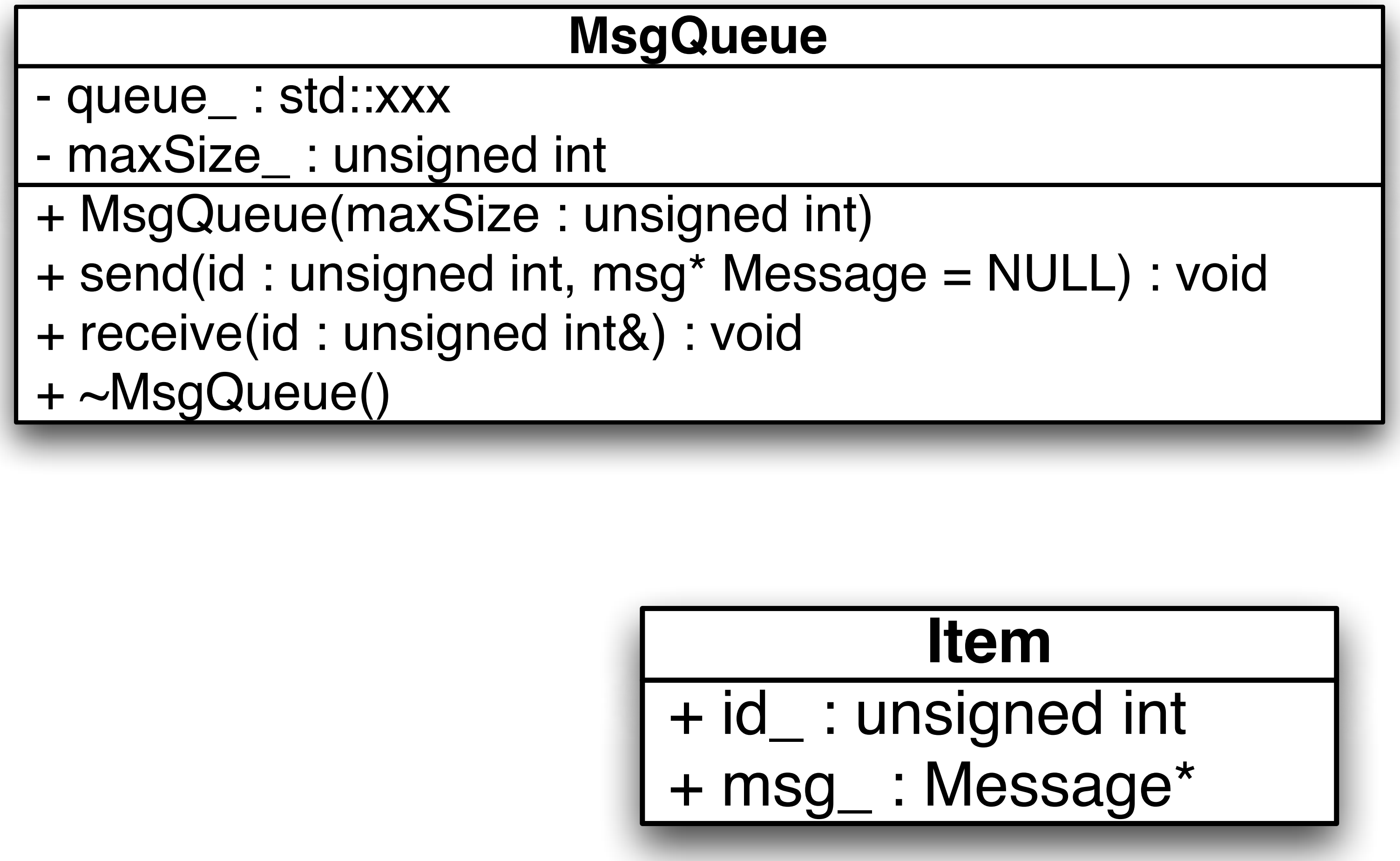
```
struct Item
{
    size_t id_;
    Message* msg_;
};
```



Example of what to pass around

- Combine an identifier with a class/structure
  - The compound signifies the control/data information to be send/received
  - The identifier is denoted by the receiving party NOT part of a globally defined enum; ***why not? Placed in a central place everyone knows; seems very good...?!***

The desired MsgQueue interface



MsgQueue

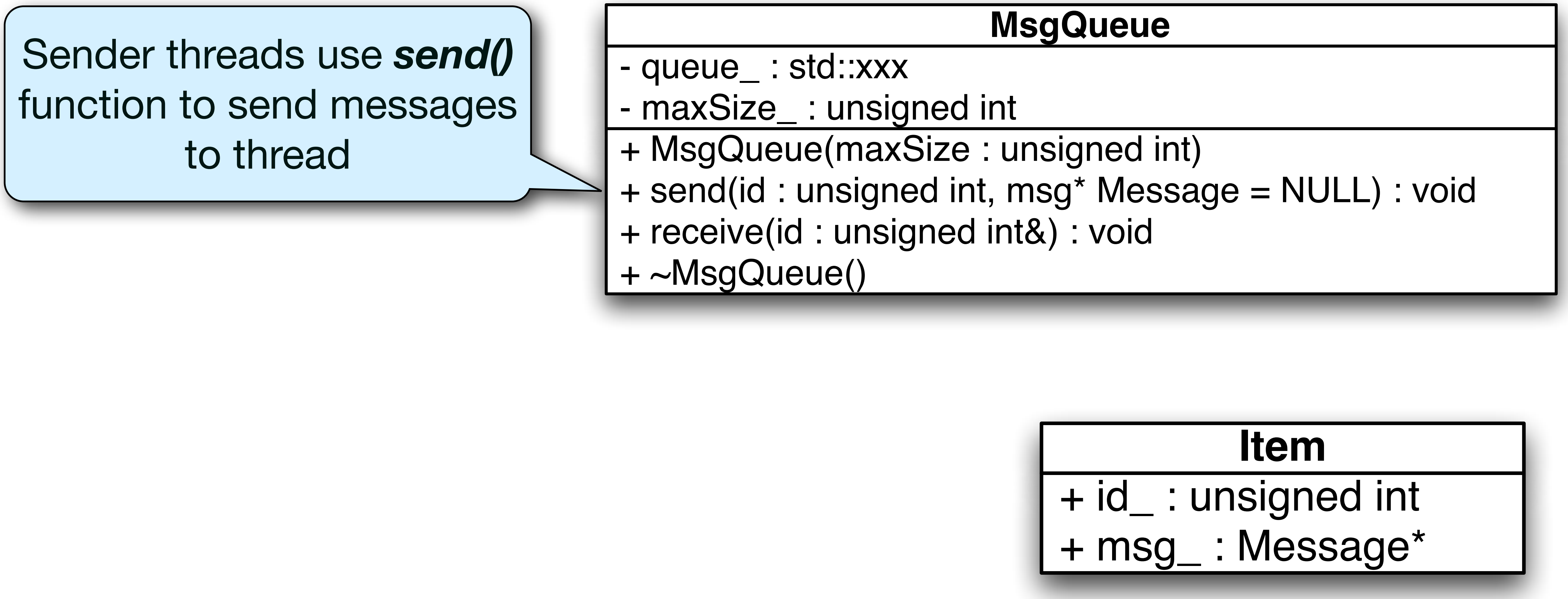
- queue_ : std::xxx
- maxSize_ : unsigned int

+ MsgQueue(maxSize : unsigned int)
+ send(id : unsigned int, msg* Message = NULL) : void
+ receive(id : unsigned int&) : void
+ ~MsgQueue()

Item

+ id_ : unsigned int
+ msg_ : Message*

The desired MsgQueue interface



Sender threads use ***send()*** function to send messages to thread

MsgQueue

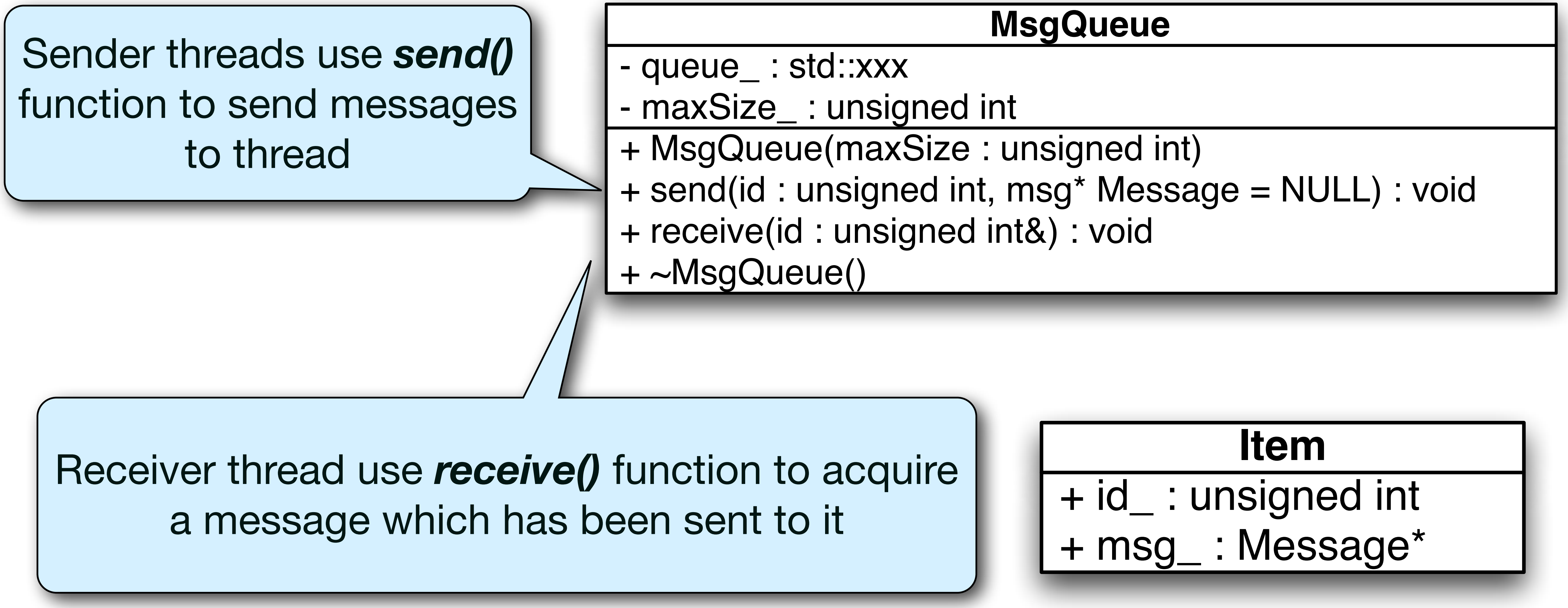
- queue_ : std::xxx
- maxSize_ : unsigned int

+ MsgQueue(maxSize : unsigned int)
+ send(id : unsigned int, msg* Message = NULL) : void
+ receive(id : unsigned int&) : void
+ ~MsgQueue()

Item

+ id_ : unsigned int
+ msg_ : Message*

The desired MsgQueue interface



Sender threads use ***send()*** function to send messages to thread

MsgQueue

- queue_ : std::xxx
- maxSize_ : unsigned int

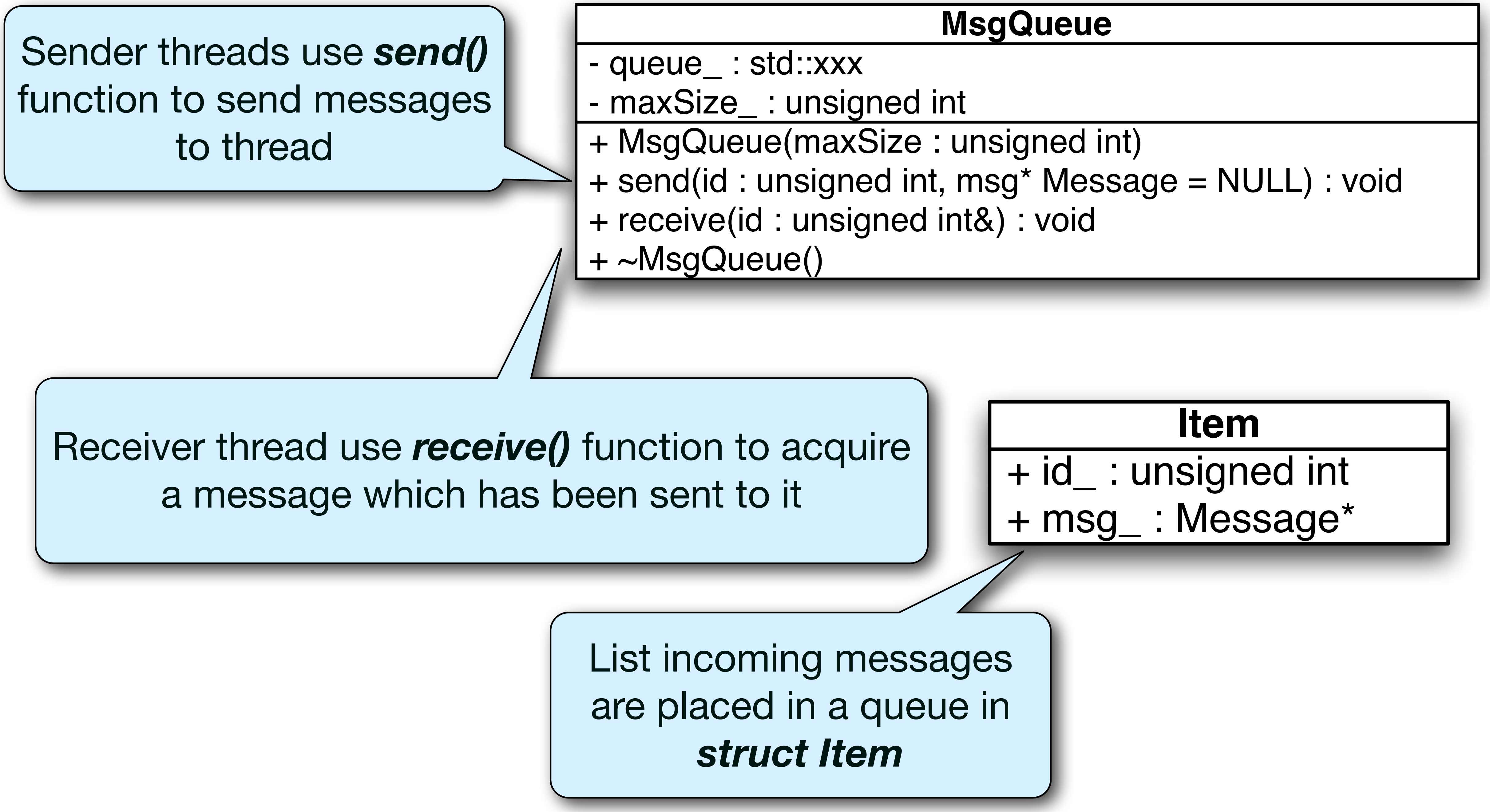
+ MsgQueue(maxSize : unsigned int)
+ send(id : unsigned int, msg* Message = NULL) : void
+ receive(id : unsigned int&) : void
+ ~MsgQueue()

Receiver thread use ***receive()*** function to acquire a message which has been sent to it

Item

+ id_ : unsigned int
+ msg_ : Message*

The desired MsgQueue interface



Sender threads use ***send()*** function to send messages to thread

MsgQueue

- queue_ : std::xxx
- maxSize_ : unsigned int

+ MsgQueue(maxSize : unsigned int)
+ send(id : unsigned int, msg* Message = NULL) : void
+ receive(id : unsigned int&) : void
+ ~MsgQueue()

Receiver thread use ***receive()*** function to acquire a message which has been sent to it

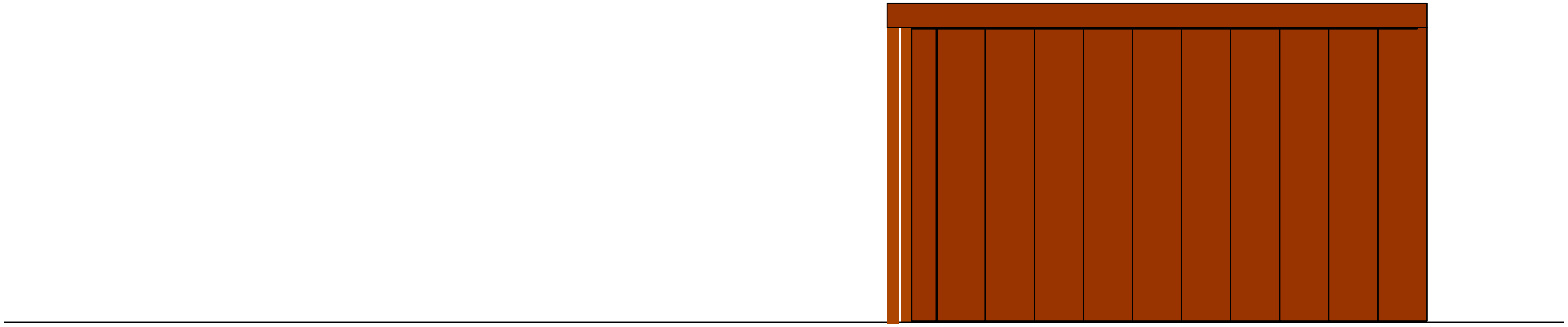
Item

+ id_ : unsigned int
+ msg_ : Message*

List incoming messages are placed in a queue in ***struct Item***

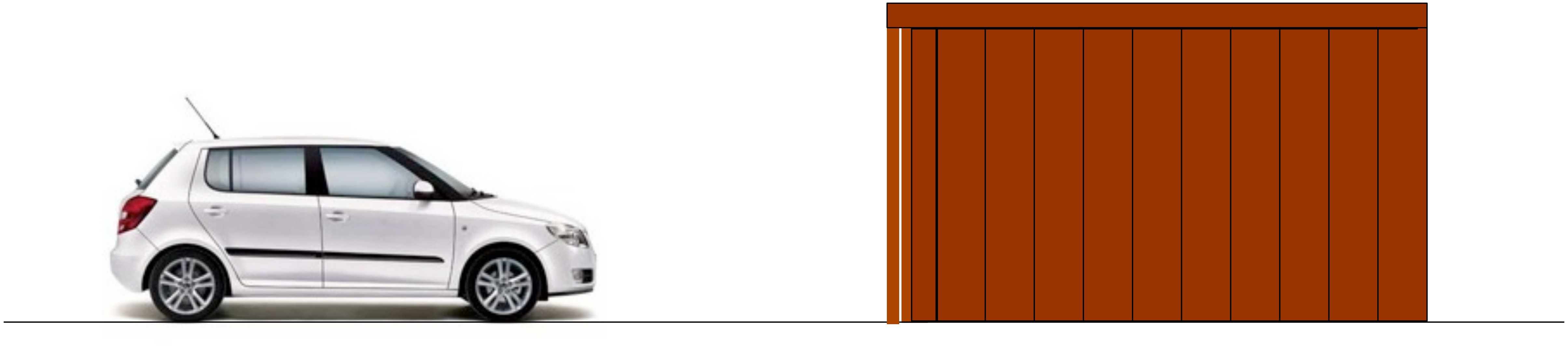
Case - Park-a-lot 2000

- Example: Park-a-lot 2000: An automated car parking system
  - One thread steers the car
  - Another thread steers the garage door opener



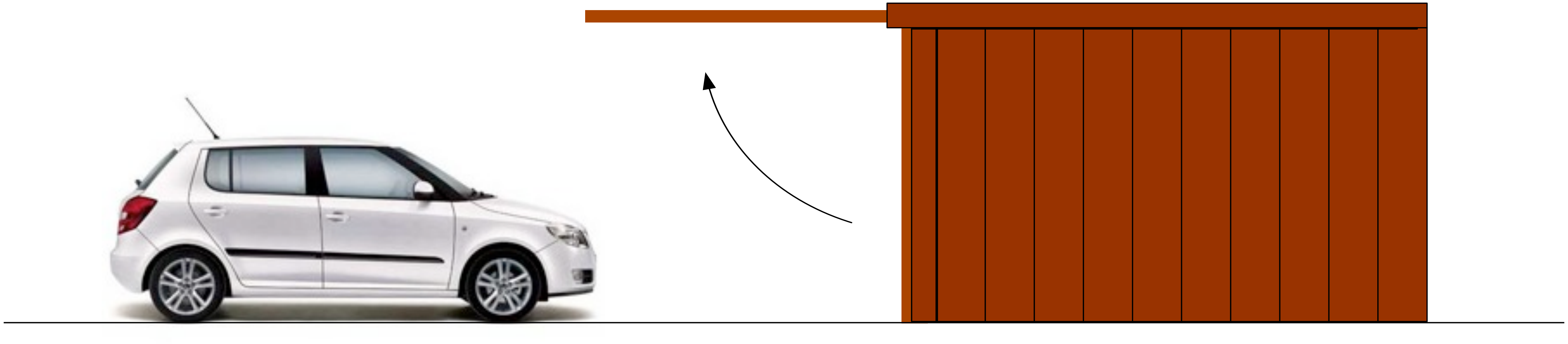
Case - Park-a-lot 2000

- Example: Park-a-lot 2000: An automated car parking system
  - One thread steers the car
  - Another thread steers the garage door opener



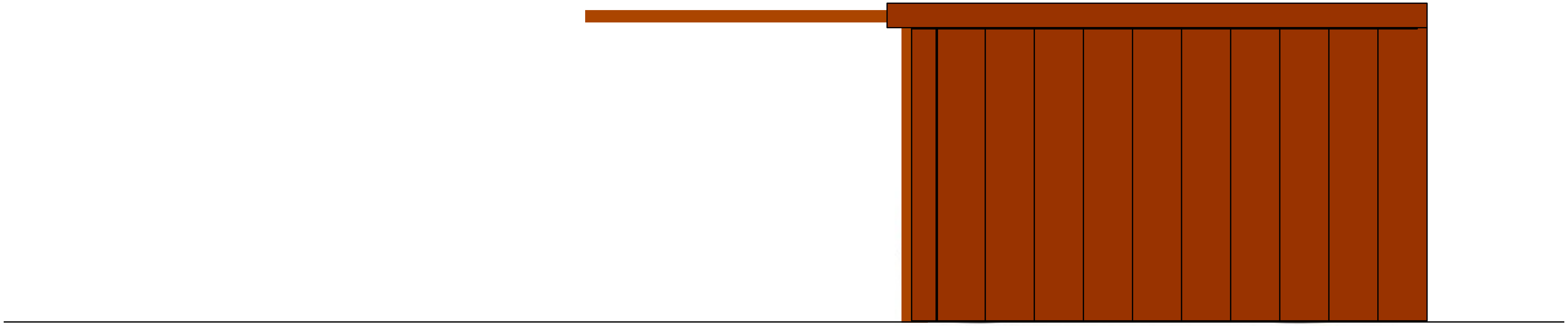
Case - Park-a-lot 2000

- Example: Park-a-lot 2000: An automated car parking system
  - One thread steers the car
  - Another thread steers the garage door opener



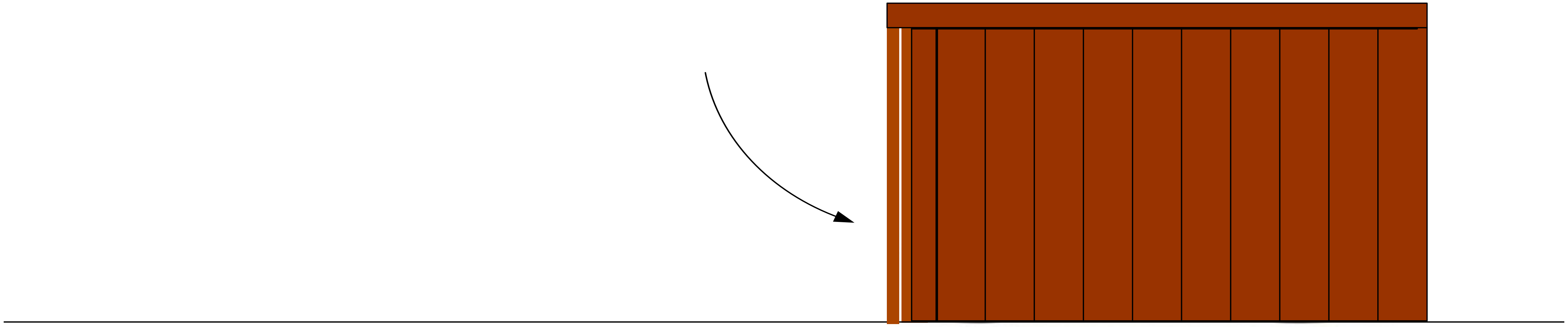
Case - Park-a-lot 2000

- Example: Park-a-lot 2000: An automated car parking system
    - One thread steers the car
    - Another thread steers the garage door opener

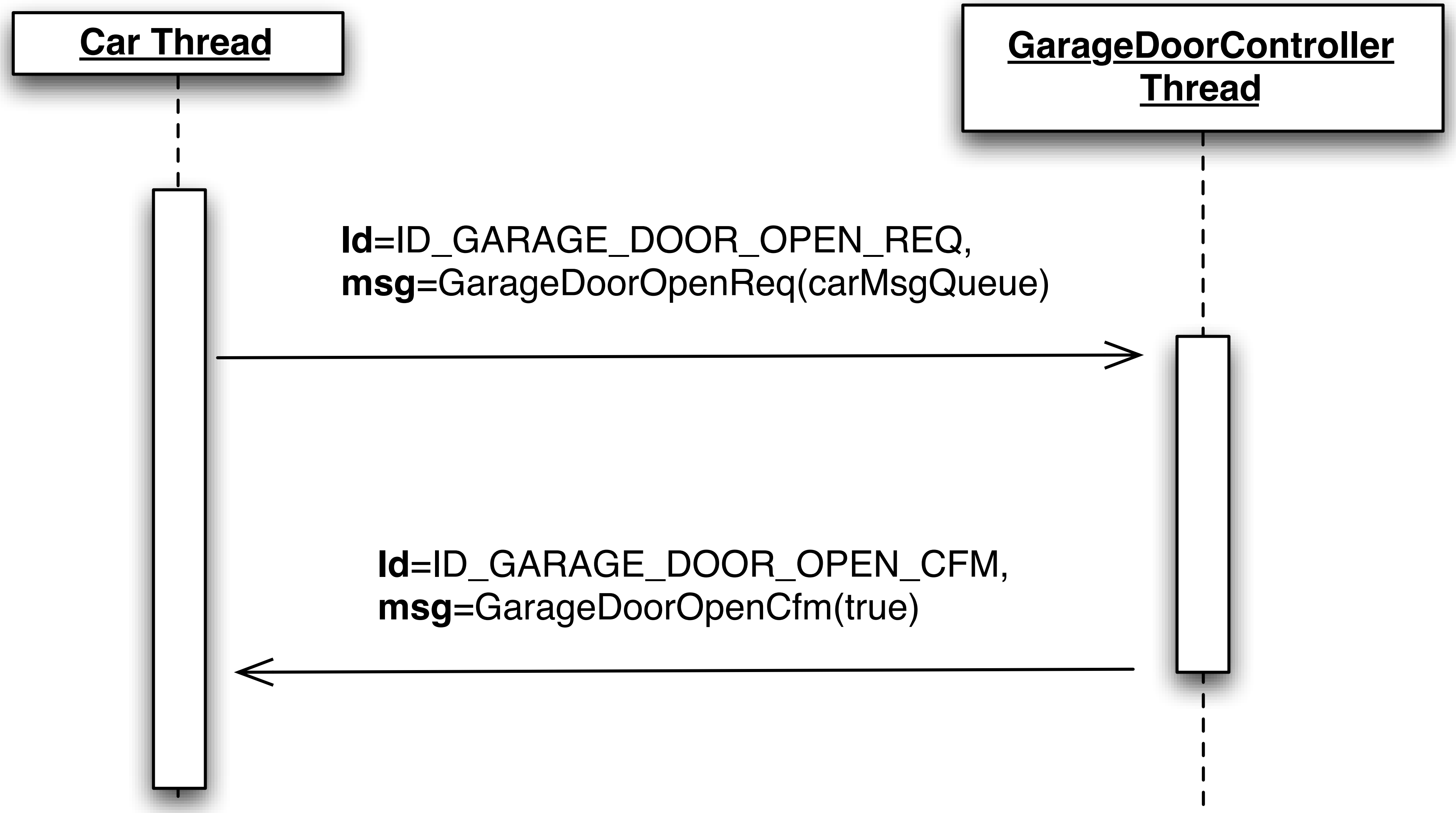


Case - Park-a-lot 2000

- Example: Park-a-lot 2000: An automated car parking system
  - One thread steers the car
  - Another thread steers the garage door opener

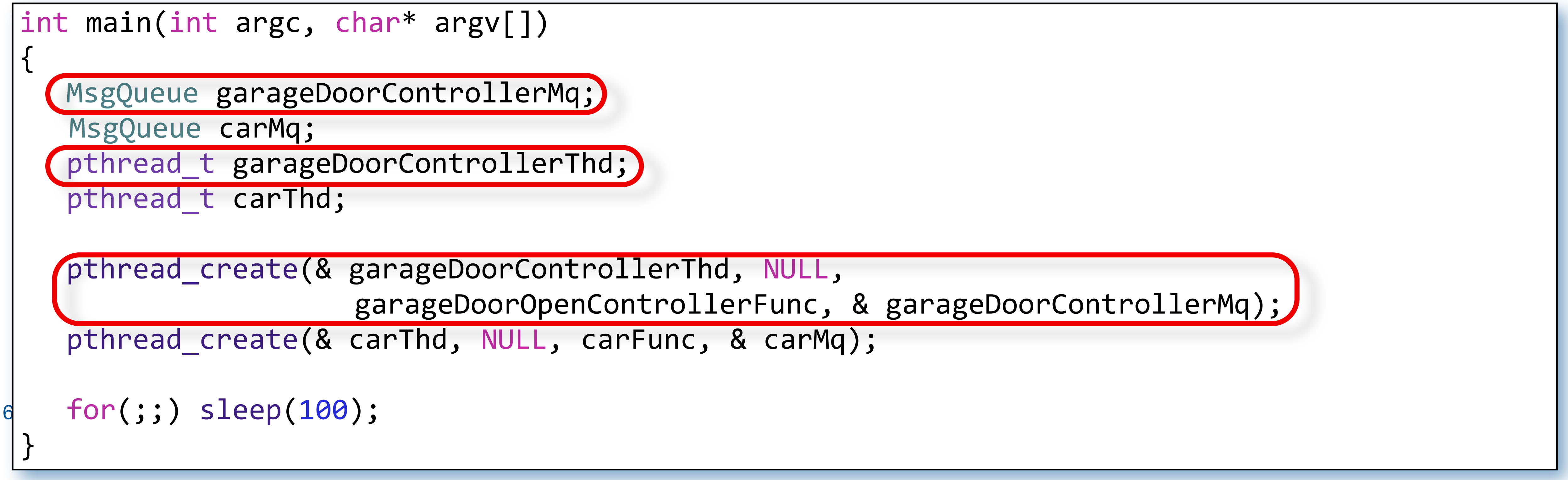


Sequence Diagram


Car Thread
GarageDoorController Thread
Id=ID_GARAGE_DOOR_OPEN_REQ,
msg=GarageDoorOpenReq(carMsgQueue)
Id=ID_GARAGE_DOOR_OPEN_CFM,
msg=GarageDoorOpenCfm(true)

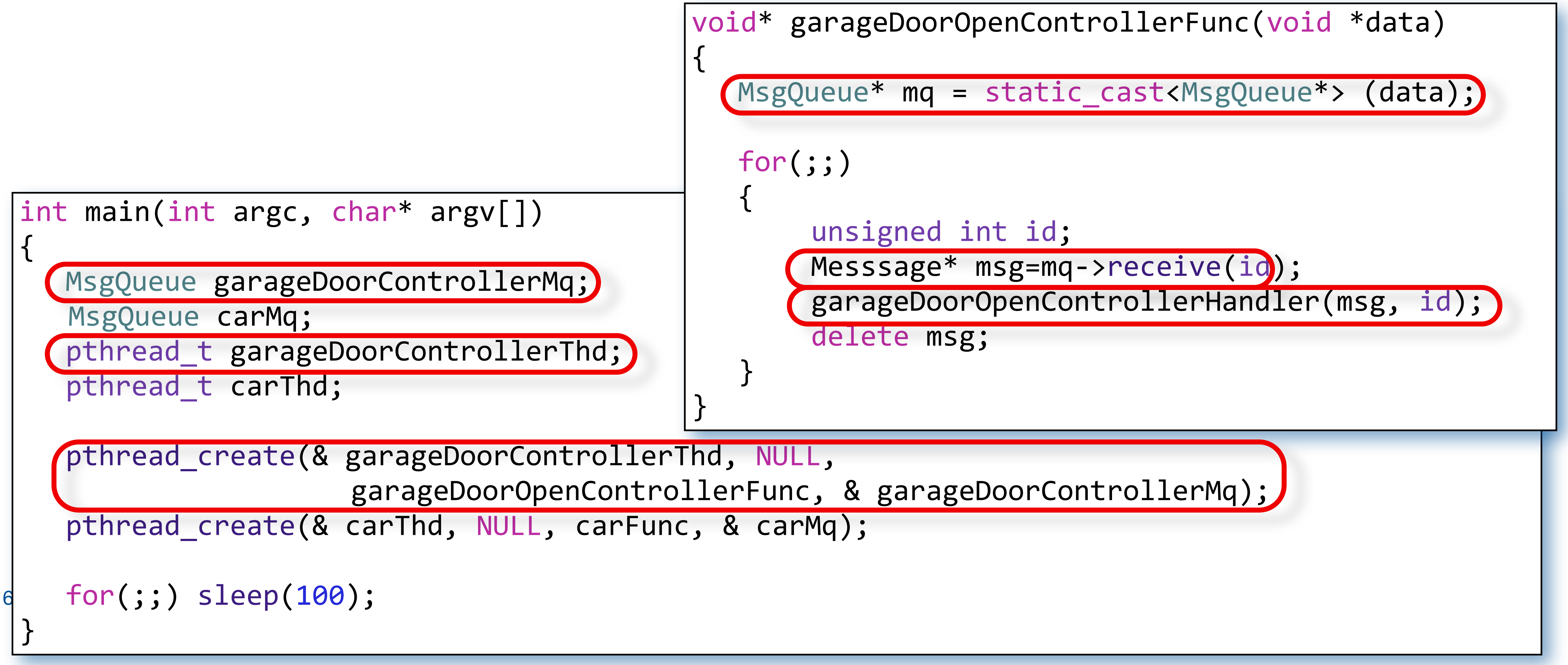
More complete example

More complete example



```
int main(int argc, char* argv[])
{
  MsgQueue garageDoorControllerMq;
  MsgQueue carMq;
  pthread_t garageDoorControllerThd;
  pthread_t carThd;

  pthread_create(& garageDoorControllerThd, NULL,
                 garageDoorOpenControllerFunc, & garageDoorControllerMq);
  pthread_create(& carThd, NULL, carFunc, & carMq);

  for(;;) sleep(100);
}
```

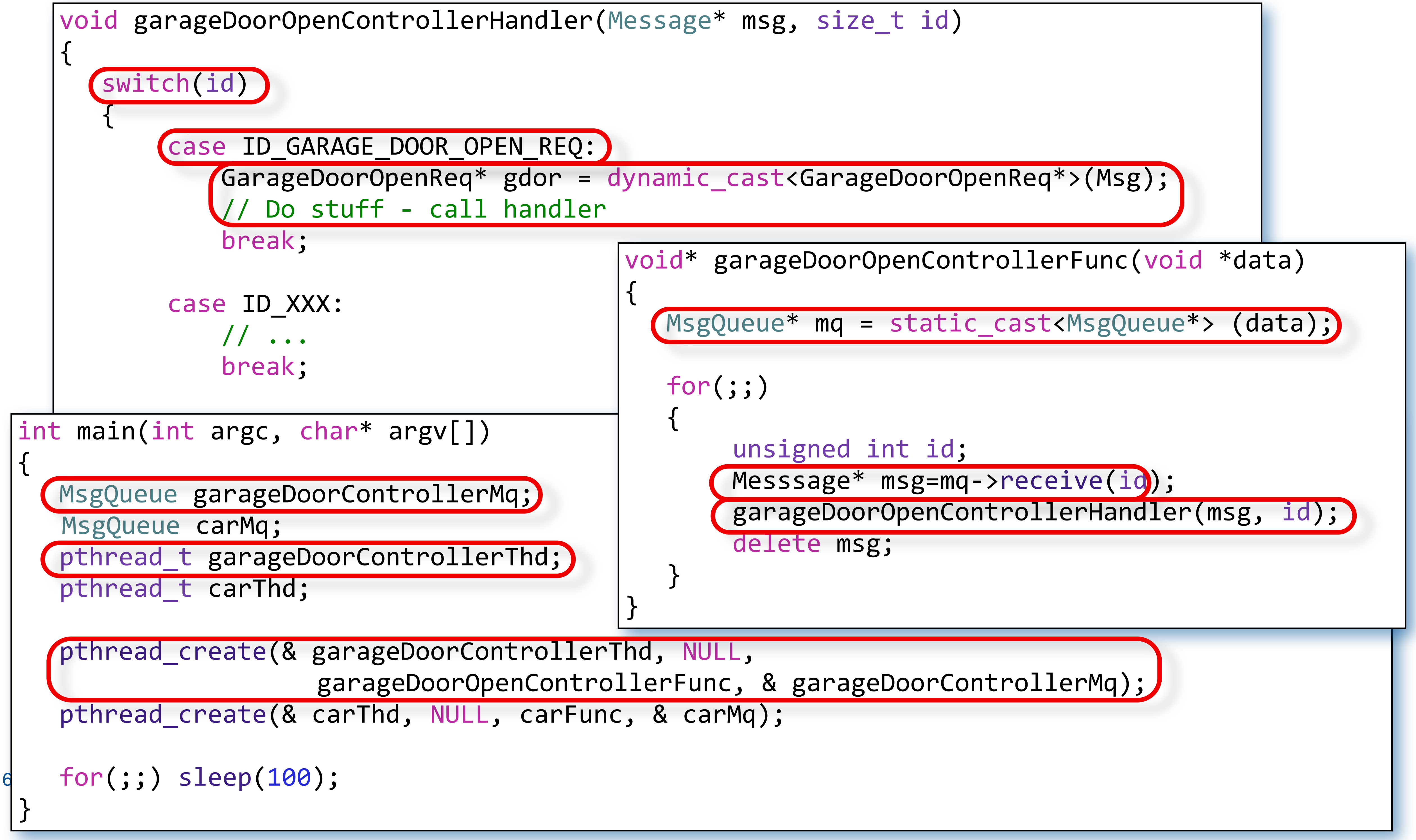
More complete example



```
int main(int argc, char* argv[])
{
  MsgQueue garageDoorControllerMq;
  MsgQueue carMq;
  pthread_t garageDoorControllerThd;
  pthread_t carThd;

  pthread_create(& garageDoorControllerThd, NULL,
                   garageDoorOpenControllerFunc, & garageDoorControllerMq);
  pthread_create(& carThd, NULL, carFunc, & carMq);

  for(;;) sleep(100);
}
```

```
void* garageDoorOpenControllerFunc(void *data)
{
  MsgQueue* mq = static_cast<MsgQueue*> (data);

  for(;;)
  {
      unsigned int id;
      Messsage* msg=mq->receive(id);
      garageDoorOpenControllerHandler(msg, id);
      delete msg;
  }
}
```

More complete example



```
void garageDoorOpenControllerHandler(Message* msg, size_t id)
{
  switch(id)
   {
      case ID_GARAGE_DOOR_OPEN_REQ:
         GarageDoorOpenReq* gdor = dynamic_cast<GarageDoorOpenReq*>(Msg);
         // Do stuff - call handler
         break;

      case ID_XXX:
         // ...
         break;
```

```
void* garageDoorOpenControllerFunc(void *data)
{
   MsgQueue* mq = static_cast<MsgQueue*> (data);

   for(;;)
   {
      unsigned int id;
      Messsage* msg=mq->receive(id);
      garageDoorOpenControllerHandler(msg, id);
      delete msg;
   }
}
```

```
int main(int argc, char* argv[])
{
   MsgQueue garageDoorControllerMq;
   MsgQueue carMq;
   pthread_t garageDoorControllerThd;
   pthread_t carThd;

   pthread_create(& garageDoorControllerThd, NULL,
                  garageDoorOpenControllerFunc, & garageDoorControllerMq);
   pthread_create(& carThd, NULL, carFunc, & carMq);

   for(;;) sleep(100);
}
```

Park-a-lot 2000 Communication

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

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

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

Park-a-lot 2000 Communication

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

Car Thread

```
void carSendingOpenReq()
{
   // Create request
   GarageDoorOpenReq* req = new GarageDoorOpenReq;
   req->mq_ = &carMq; // Who the requester is

   // Send it
   garageDoorControllerMq.send(ID_GARAGE_DOOR_OPEN_REQ, req);
}
```

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

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

Park-a-lot 2000 Communication

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

Car Thread

```
void carSendingOpenReq()
{
   // Create request
   GarageDoorOpenReq* req = new GarageDoorOpenReq;
   req->mq_ = &carMq; // Who the requester is

   // Send it
   garageDoorControllerMq.send(ID_GARAGE_DOOR_OPEN_REQ, req);
}
```

```
struct GarageDoorOpenReq : publ
{
   MsgQueue* mq_;
};
```

GDC Thread

```
void handleGarageOpenDoorReg(GarageDoorOpenReq* req)
{
   // Create responds
   GarageDoorOpenCfm* cfm = new GarageDoorOpenCfm;
   cfm->result_ = openGarageDoor(); // The door is open

   // Send responds to requester...
   req->mq_->send(ID_GARAGE_DOOR_OPEN_CFM, cfm);
}
```

```
struct GarageDoorOpenCfm : publi
{
   bool result_;
};
```

Park-a-lot 2000 Communication

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

```
struct GarageDoorOpenReq : publ
{
   MsgQueue* mq_;
};
```

```
struct GarageDoorOpenCfm : publi
{
   bool result_;
};
```

Car Thread

```
void carSendingOpenReq()
{
   // Create request
   GarageDoorOpenReq* req = new GarageDoorOpenReq;
   req->mq_ = &carMq; // Who the requester is

   // Send it
   garageDoorControllerMq.send(ID_GARAGE_DOOR_OPEN_REQ, req);
}
```

GDC Thread

```
void handleGarageOpenDoorReg(GarageDoorOpenReq* req)
{
   // Create responds
   GarageDoorOpenCfm* cfm = new GarageDoorOpenCfm;
   cfm->result_ = openGarageDoor(); // The door is open

   // Send responds to requester...
   req->mq_->send(ID_GARAGE_DOOR_OPEN_CFM, cfm);
}
```

Car Thread

```
void handleGarageOpenDoorCfm(GarageDoorOpenCfm* cfm)
{
   // Check responds
   if(cfm->result_)
   {
        driveIntoParkingLot();
   }
}
```

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