Embedded Software

Abstract Object-Oriented OS APIs

AARHUS UNIVERSITY

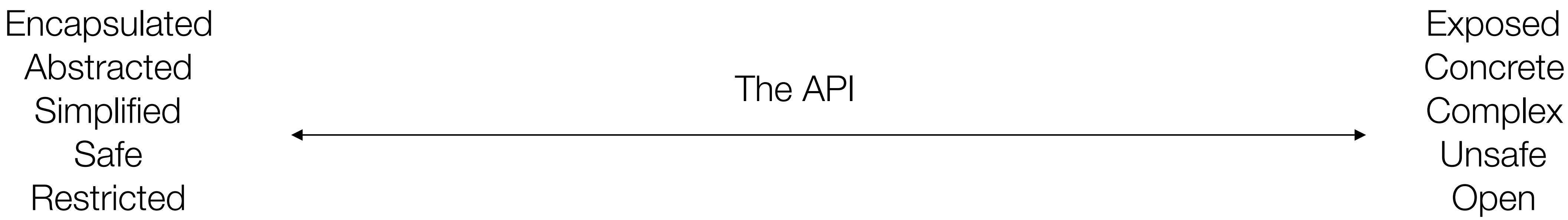
Agenda

- What is an API and why use it?
- What is an OS API?
- What should/could it cover?
- How is the wrapping of the real OS achieved (includes pimpl/cheshire idiom + defines)?
- Why an OO representation of OS API?
- From OO to code
    - Concrete examples using OS API
- Guidelines for writing event based thread oriented programs with the OO OS API

API and OS API - What and Why?

What is an API?

- Why use an API?
  - Encapsulation – the API may hide some of the system
  - Abstraction – only the system interface is revealed
  - Simplification – the API may restrict access to the system



Encapsulated
Abstracted
Simplified
Safe
Restricted

The API

Exposed
Concrete
Complex
Unsafe
Open

The OS API concept

- Operating systems have extensive APIs to access OS resources
  - Threads, mutexes, semaphores, timers, pipes…
  - Example: Thread creation

```
//win32
HANDLE CreateThread(…);
```

```
//POSIX - Linux
void* pthread_create(…);
```

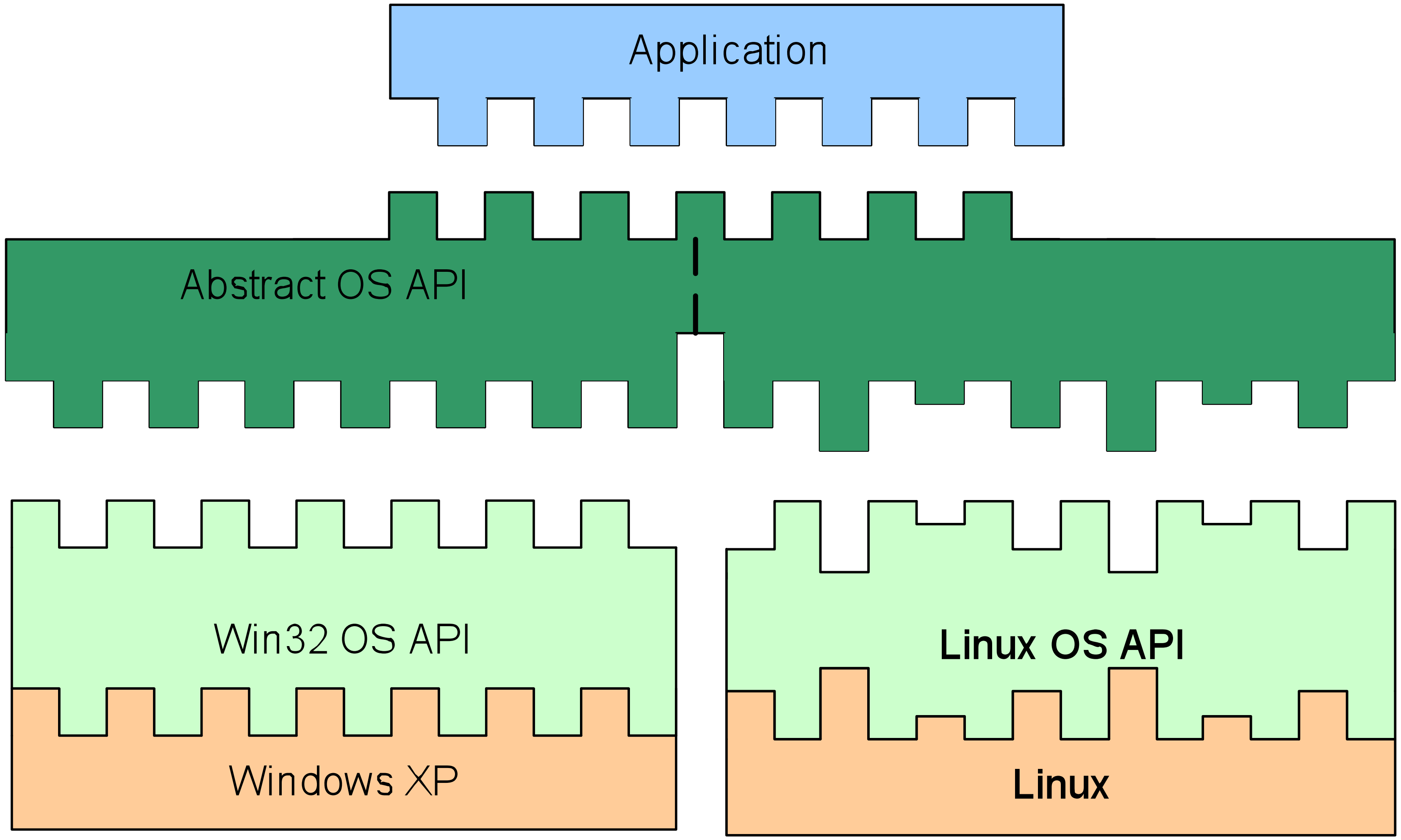
```
//VxWorks
void* pthread_create(…);
```

```
//FreeRTOS
portBASE_TYPE xTaskCrete(…);
```

Concrete example - Article on OSAL

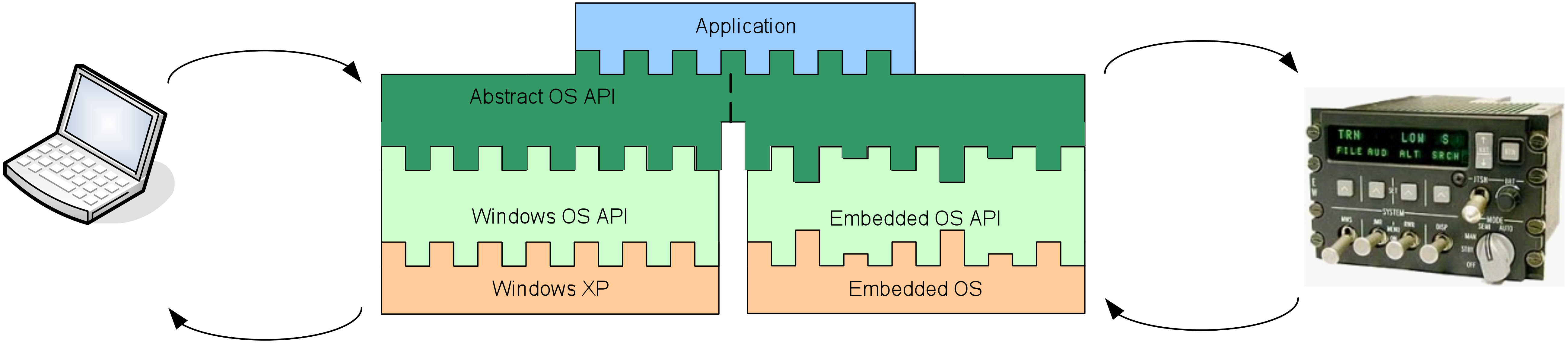
- *An Operating System Abstraction Layer for Portable Applications in Wireless Sensor Networks* (for the Mantis OS and FreeRTOS)
  - Why?
    - Faster development due to increase in portability
      - New platforms demand "*only*" implementation of OSAL (and drivers)
    - Support for different OS's deployed on different platforms
    - Same API used again and again - Only one API to learn
  - How?
    - Thin layer introduced between Application layer and OS layer

The OS API abstraction



Abstract OS API: Cross-development

- Develop the system for the host platform
  - Debug the system until no errors are left
  - Use stubs for real-life peripherals (GoF Strategy)
- Now develop the same system for target platform
  - Little or no change to application
  - Now debug target-specific problems (timing, real peripherals, etc.)



Application

Abstract OS API

Windows OS API

Embedded OS API

Windows XP

Embedded OS

Thin-layer example - Wrapping OS functionality

- Semaphore implementation in linux

```
// inc/osapi/linux/Semaphore.hpp
#include <semaphore.h>
#include <osapi/Utility.hpp>

namespace osapi
{
  class Semaphore : Notcopyable
  {
  public:
    Semaphore(unsigned int initCount);
    void wait();
    void signal();
    ~Semaphore();
  private:
   sem_t semId_;
  };
}
```

```
// linux/Semaphore.cpp
#include <osapi/Semaphore.hpp>

namespace osapi
{
  Semaphore::Semaphore(unsigned int initCount)
  {
    if(sem_init(&semId_, 1, initCount) != 0)
      throw SemaphoreError();
  }

  void Semaphore::wait()
  {
    if(sem_wait(&semId_) != 0) throw SemaphoreError();
  }

  void Semaphore::signal()
  {
    if(sem_post(&semId_) != 0) throw SemaphoreError();
  }

  Semaphore::~Semaphore()
  {
    sem_destroy(&semId_);
  }

}
```

What should/could it cover?

Things to cover - Example but not limited to

- Basic system functionality *considered* important in the uses deemed important (*Inspired* by FreeRTOS API)
  - Threads
  - Mutexes/Semaphores
  - Conditionals
  - Time functions
  - Message Queues
  - Timers
  - Input (keyboard)
  - External connection handling such as TCP/IP etc.
  - *Further requirements are more than feasible, this is but a mere start*
    - *Depends on the usage needs*

How is the wrapping of the real OS achieved

Multiple platforms via defines

- Class exercise
  - Inspect OS Api code and determine how its used
  - Find file to illustrate and explain to class
  - How do you use it?

In Grp - 10-15mins

Grp chosen at random

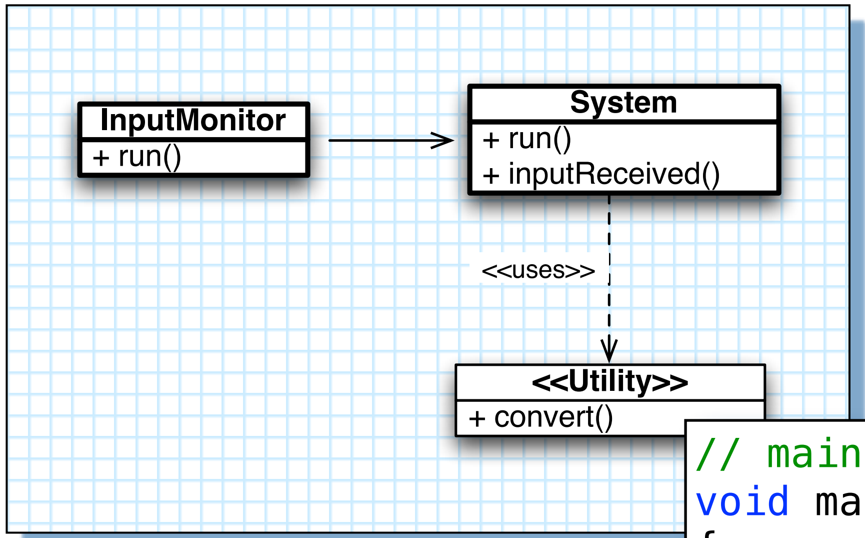
Why an OO representation of OS API?

An abstract object-oriented OS API

- Why should the abstract OS API be object oriented?
    - Easier to work with (if you're used to objects)
    - Cleaner code
    - Decreases the representational gap between design and implementation

- The representational gap
    - The ”distance in representation” between the design and implementation of your application

The representational gap



```
// system.h
class System : public ThreadFunctor
{
public:
      enum { ID_INPUT };
      System() : mq_(MAX_QUEUE_SIZE) {}
      inputReceived(InputMsg* msg);
     MsgQueue mq_;
```

```
// main.cpp
void main()
{
     System sys;
     InputMonitor inputMonitor(sys.getMsgQueue());
     sys.start();
     inputMonitor.start();
     Thread ths(&sys);
     Thread thi(&inputMonitor);

     while(true) sleep(1000);
}
```

```
// inputmonitor.h
class InputMonitor : public Th
{
public:
      InputMonitor(MsgQueue* s
      ...
};


// inputmonitor.cpp
...
InputMonitor::inputReady()
{
      value = gpio_read_16bit(0x22);
      ...
      sysMsgQueue_->send(System::ID_INPUT, inputMsg);
}
```

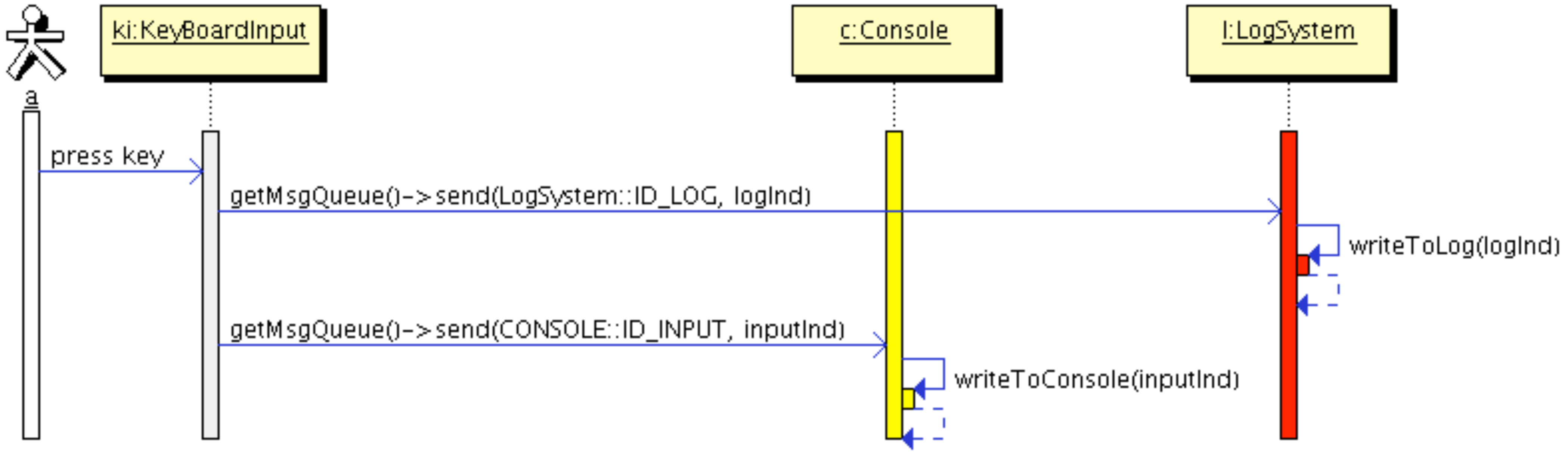
```
ve(id);

tic_cast<InputMsg*> (msg));

      }
      delete msg;
}

void System::inputReceived(InputMsg* msg)
{
      convInput = convert(msg->value_);
}
```

The challenge

- I want a simple program that can (and is based on the OS Api)
  - *Read keyboard input from stdin (thread)*
  - *Write it out to a log file (thread)*
  - *Print it out to console (thread) **
    - *Used in design, but not implemented*

Design - UML Sequence diagram

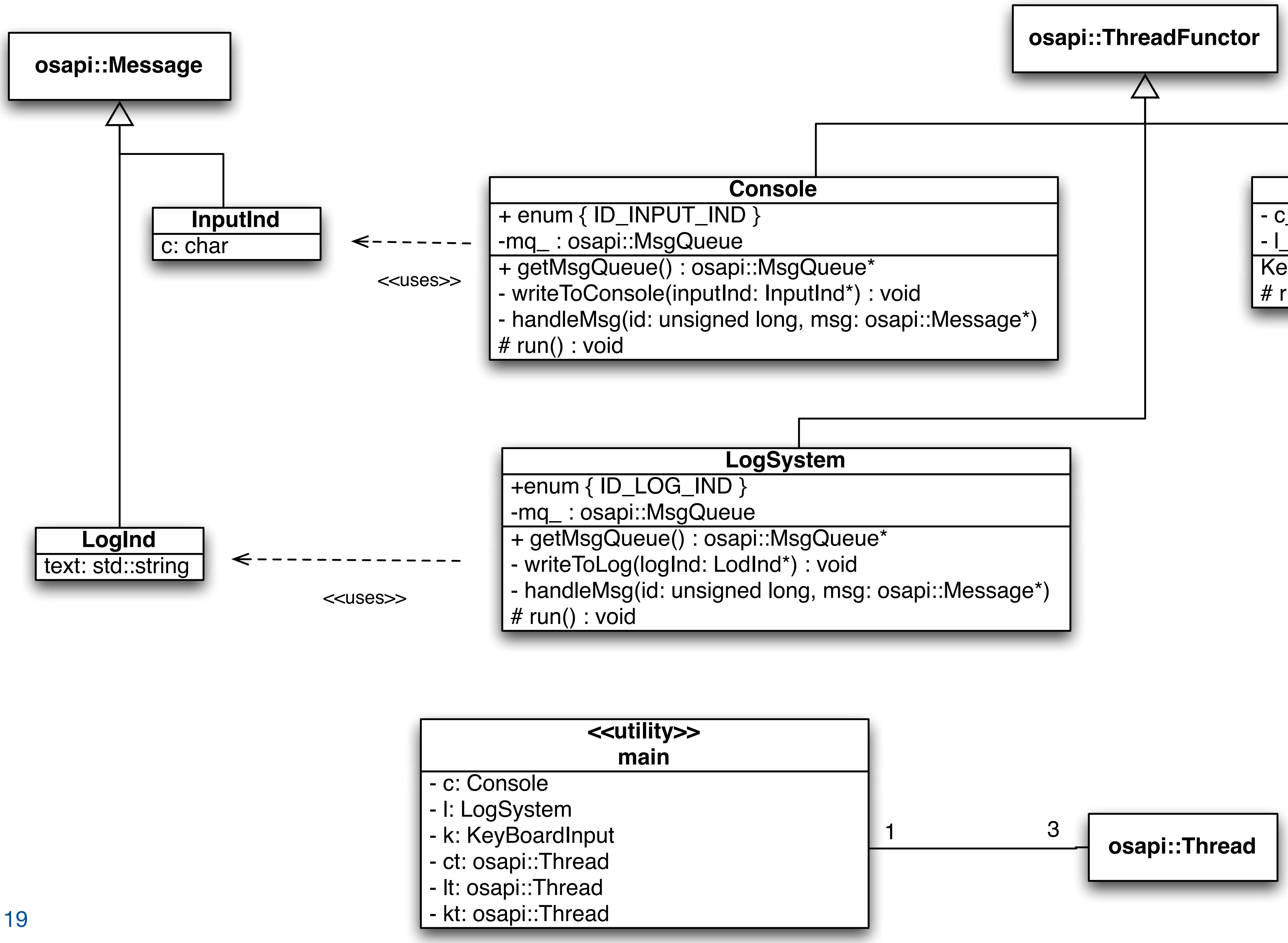
- Sequence diagram showing main Use Case



a
ki:KeyBoardInput
c:Console
l:LogSystem
press key
getMsgQueue()->send(LogSystem::ID_LOG, logInd)
writeToLog(logInd)
getMsgQueue()->send(CONSOLE::ID_INPUT, inputInd)
writeToConsole(inputInd)

Design - UML Class diagrams

- Class model for whole system



osapi::Message

InputInd
c: char

LogInd
text: std::string

<<uses>>

<<uses>>

osapi::ThreadFunctor

Console
+ enum { ID_INPUT_IND }
-mq_ : osapi::MsgQueue
+ getMsgQueue() : osapi::MsgQueue*
- writeToConsole(inputInd: InputInd*) : void
- handleMsg(id: unsigned long, msg: osapi::Message*)
# run() : void

KeyBoardInput
- c_ : Console*
- l_ : LogSystem*
KeyBoardInput(c: Console*, l: LogSystem*)
# run() : void

LogSystem
+enum { ID_LOG_IND }
-mq_ : osapi::MsgQueue
+ getMsgQueue() : osapi::MsgQueue*
- writeToLog(logInd: LodInd*) : void
- handleMsg(id: unsigned long, msg: osapi::Message*)
# run() : void

<<utility>>
main
- c: Console
- l: LogSystem
- k: KeyBoardInput
- ct: osapi::Thread
- lt: osapi::Thread
- kt: osapi::Thread

osapi::Thread

Implementation using the OO OS Api

- The main.cpp file

```cpp
// main.cpp
#include <osapi/Thread.hpp>
//#include <Console.hpp>
#include <osapi/example/LogSystem.hpp>
#include <osapi/example/KeyBoardInput.hpp>

int main()
{
    //Console c;
    LogSystem l;
    KeyBoardInput k(&l);

    //osapi::Thread ct(&c);
    osapi::Thread lt(&l);
    lt.start();
    osapi::Thread kt(&k);
    kt.start();

    //ct.join();
    lt.join();
    kt.join();

}
```

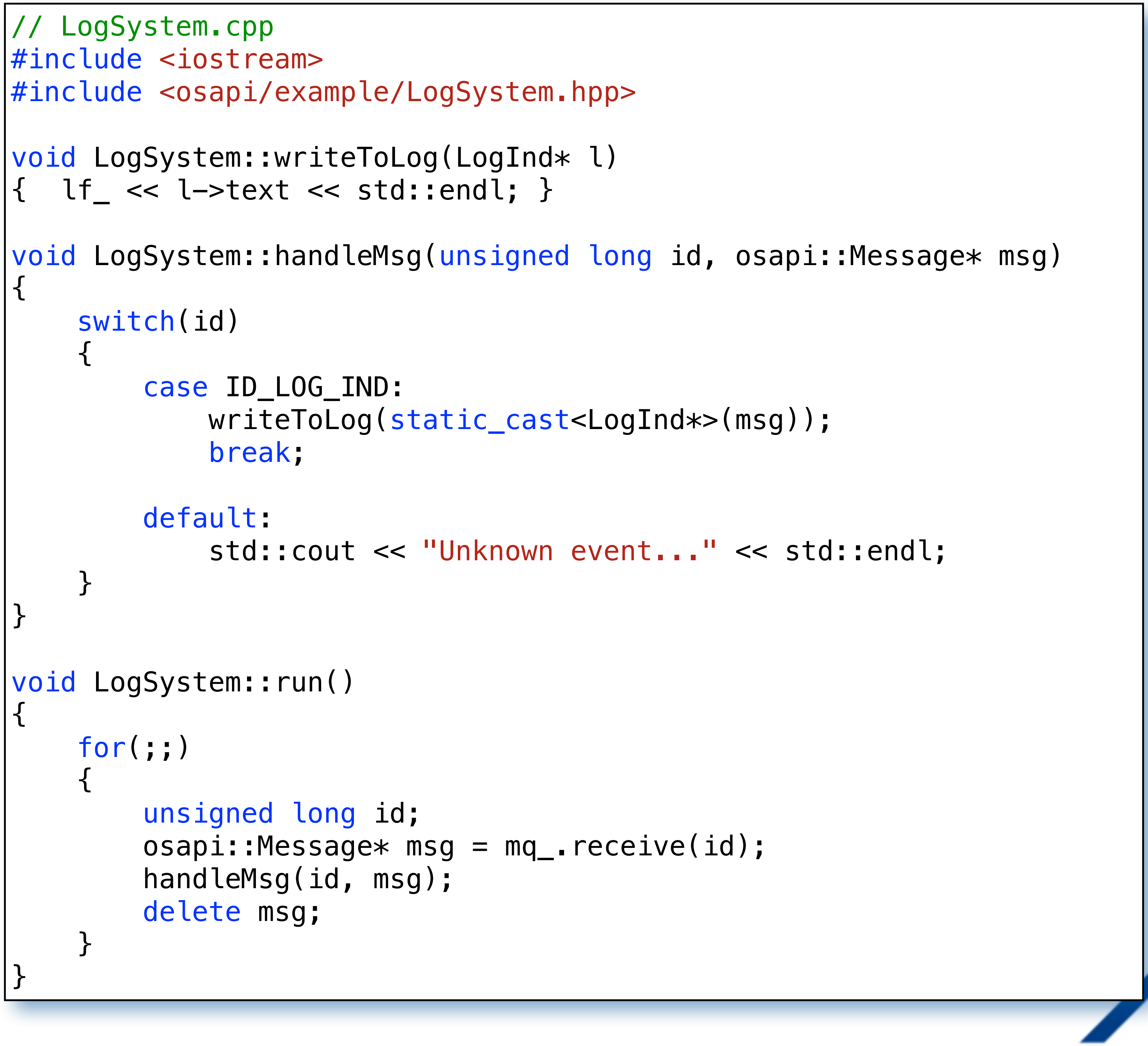
Implementation using the OO OS Api

```
#ifndef KEYBOARD_INPUT_H_
#define KEYBOARD_INPUT_H_
#include <string>
#include <osapi/MsgQueue.hpp>
#include <osapi/ThreadFunctor.hpp>
#include <osapi/example/LogSystem.hpp>

class KeyBoardInput : public osapi::ThreadFunctor
{
public:
    KeyBoardInput(LogSystem* l)
    : l_(l) {}
private:
    void run();
    LogSystem*      l_;
};

#endif
```

```
// KeyBoardInput.cpp
#include <iostream>
#include <osapi/example/KeyBoardInput.hpp>

void KeyBoardInput::run()
{
    for(;;)
    {
        std::string s;
        std::cin >> s;
        LogInd* logInd = new LogInd;
        logInd->text = s;
        l_->getMsgQueue()->send(LogSystem::ID_LOG_IND, logInd);
    }
}
```

```
#ifndef LOG_SYSTEM_H_
#define LOG_SYSTEM_H_

#include <string>
#include <fstream>
#include <osapi/MsgQueue.hpp>
#include <osapi/ThreadFunctor.hpp>

struct LogInd : public osapi::Message
{ std::string text; };

class LogSystem : public osapi::ThreadFunctor
{
public:
    enum { ID_LOG_IND };
    static const int MAX_QUEUE_SIZE = 10;
    LogSystem()
    : mq_(MAX_QUEUE_SIZE), lf_("log.txt") { }

    osapi::MsgQueue* getMsgQueue() { return &mq_; }

private:
    void writeToLog(LogInd* l);
    void handleMsg(unsigned long id, osapi::Message* msg);
    void run();

    osapi::MsgQueue mq_;
    std::ofstream   lf_;
};

#endif
```

Implementation using the OO OS Api



```
// LogSystem.cpp
#include <iostream>
#include <osapi/example/LogSystem.hpp>

void LogSystem::writeToLog(LogInd* l)
{  lf_ << l->text << std::endl; }

void LogSystem::handleMsg(unsigned long id, osapi::Message* msg)
{
    switch(id)
    {
        case ID_LOG_IND:
            writeToLog(static_cast<LogInd*>(msg));
            break;

        default:
            std::cout << "Unknown event..." << std::endl;
    }
}

void LogSystem::run()
{
    for(;;)
    {
        unsigned long id;
        osapi::Message* msg = mq_.receive(id);
        handleMsg(id, msg);
        delete msg;
    }
}
```

Inspect implementation

- Class exercise
  - Inspect code and compare it to the design presented
  - How is the OS Api used

In Grp - 10-15mins

Questions???

From pThread to OO OS Api thread

From pThread to OO OS Api thread

```
class Thread
{
public:
    //...
    Thread(ThreadFunctor* tf,
           ThreadPriority p = PRIORITY_NORMAL,
           const std::string& name="",
           bool autoStart = false);
    //...
    void start();
private:
    //...
    ThreadFunctor*  tf_;
};
```

- Inheriting from ThreadFunctor and implementing run()
  - ThreadFunctor ***is*** the thread
  - class ***Thread*** is the ***controlling*** entity and handles start, priority, wait/join etc.
  - class Thread is passed pointer to thread upon creation

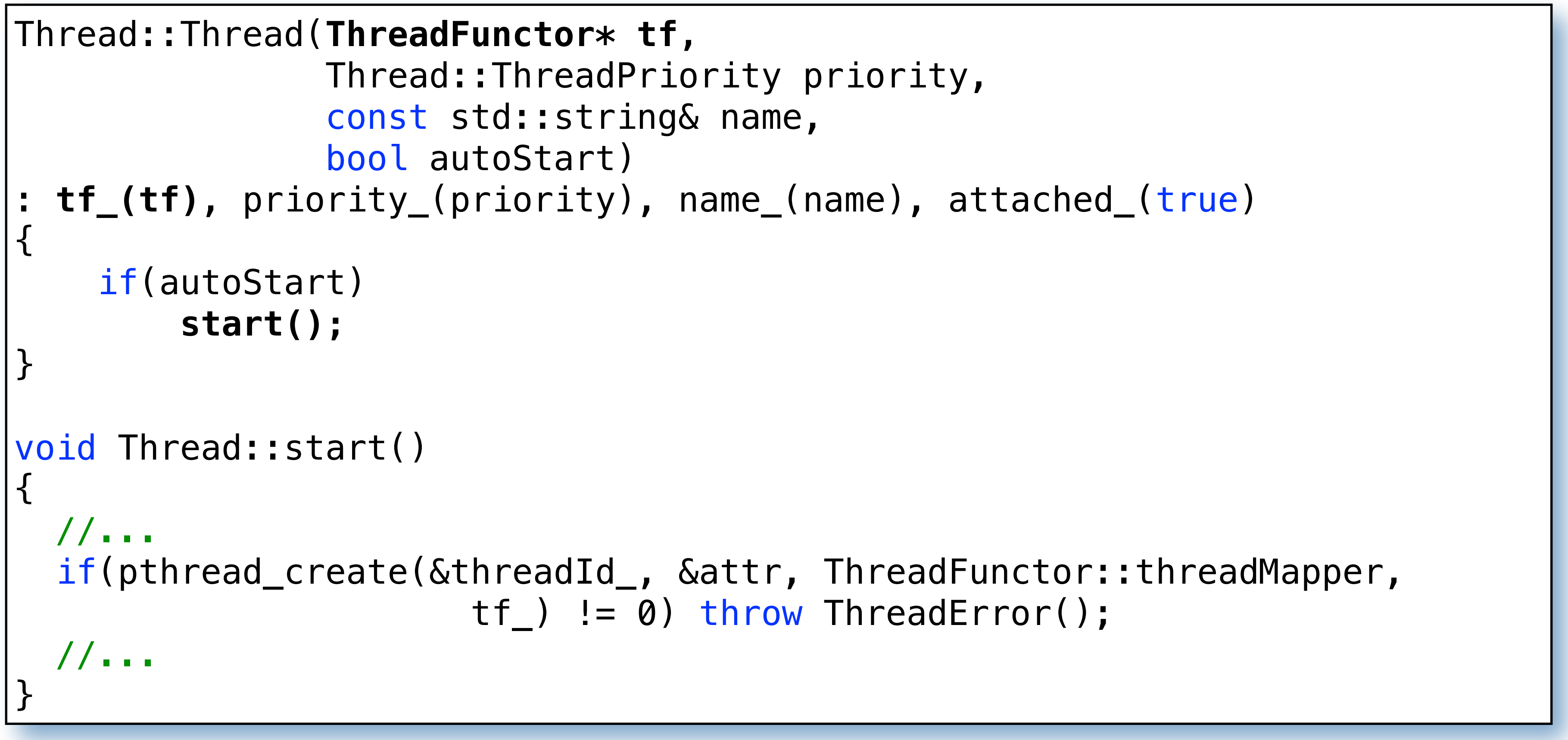
```
class ThreadFunctor
{
public:

protected:
    virtual void run() = 0;
    ~ThreadFunctor(){}
private:
    //...
    static void* threadMapper(void* p);
};
```

↓

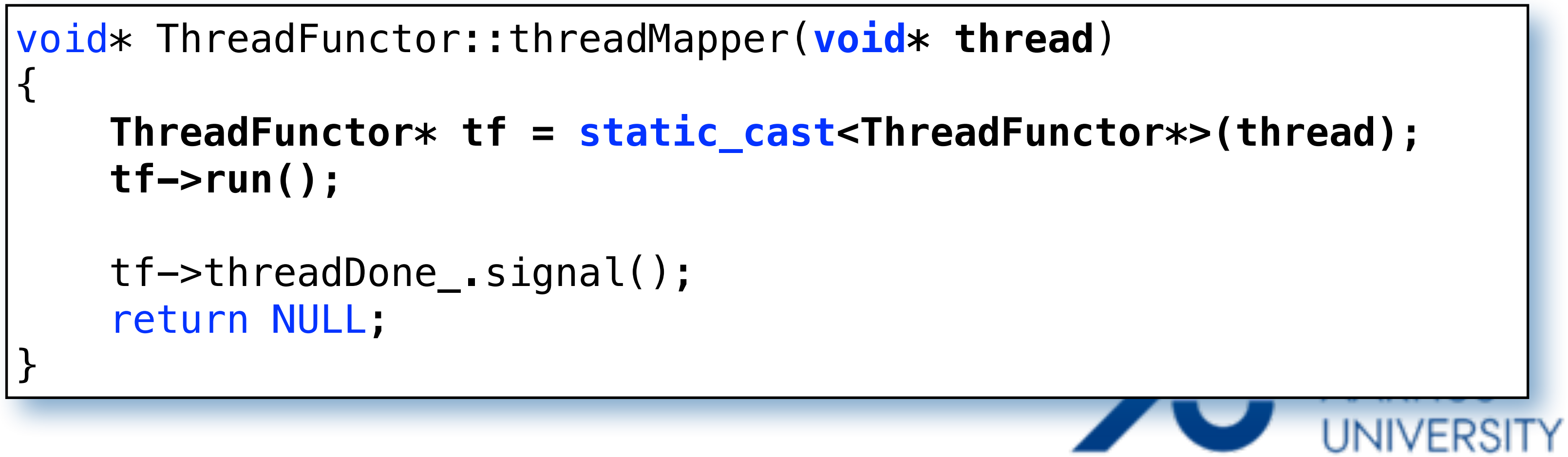
```
class KeyBoardInput : public
osapi::ThreadFunctor
{
public:
    KeyBoardInput(LogSystem* l)
    : l_(l) {}
protected:
    virtual void run();
private:
    LogSystem*      l_;
};
```

From pThread to OO OS Api thread

- class Thread creates thread using pthread_create()
- threadMapper is a static function with the signature required by pthread_create()
- run() is in effect the real thread function



```
Thread::Thread(ThreadFunctor* tf,
               Thread::ThreadPriority priority,
               const std::string& name,
               bool autoStart)
: tf_(tf), priority_(priority), name_(name), attached_(true)
{
    if(autoStart)
        start();
}

void Thread::start()
{
  //...
  if(pthread_create(&threadId_, &attr, ThreadFunctor::threadMapper,
                      tf_) != 0) throw ThreadError();
  //...
}
```



```
void* ThreadFunctor::threadMapper(void* thread)
{
    ThreadFunctor* tf = static_cast<ThreadFunctor*>(thread);
    tf->run();

    tf->threadDone_.signal();
    return NULL;
}
```

Inspect implementation

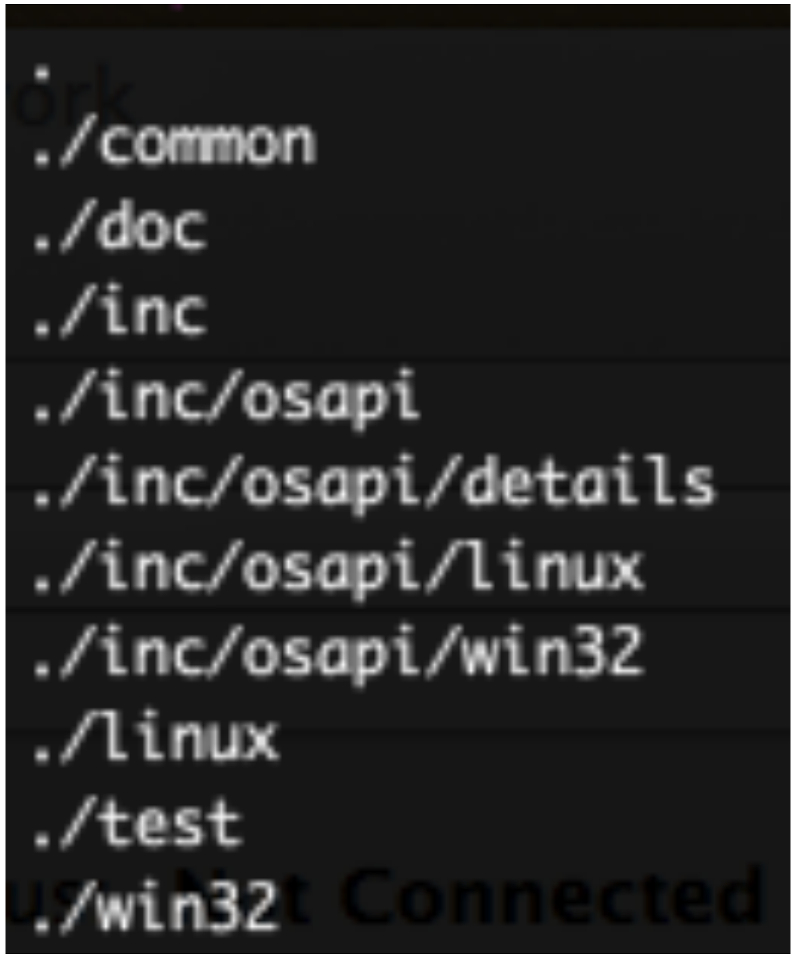
- Class exercise
  - Determine how the OO OS API go about the transition from pthread function to class method thread function...

In Grp - 10mins

Questions???

Library Layout - Directory

- Challenges creating an OS API
  - Handling architectures & OS
  - Macros, defines, directories
  - Common “denominator” or lack of...
    - Consequences



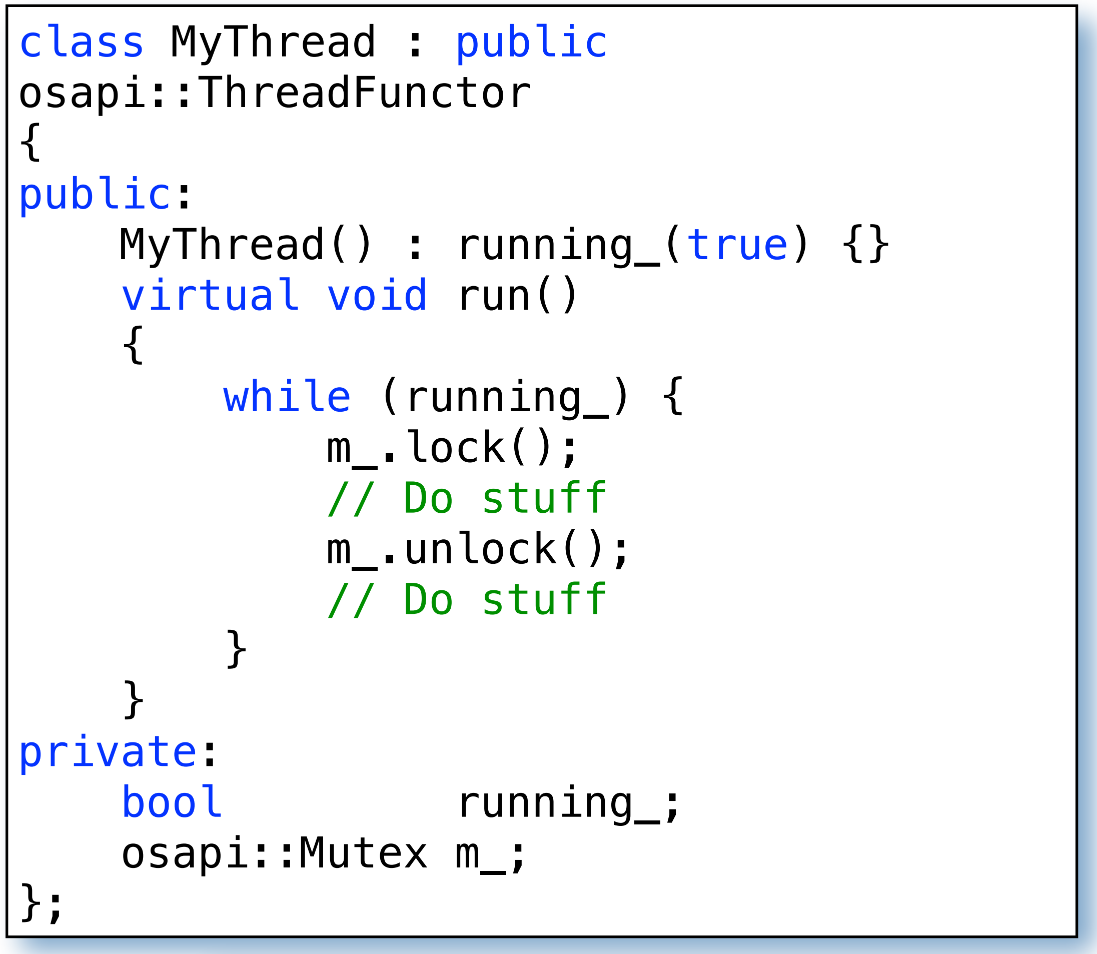
```
.
./common
./doc
./inc
./inc/osapi
./inc/osapi/details
./inc/osapi/linux
./inc/osapi/win32
./linux
./test
./win32
```

Grp 2 & 2 - 3mins

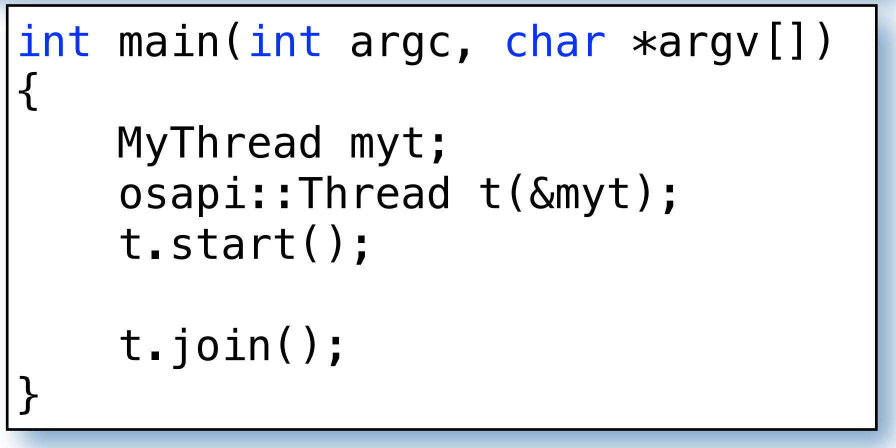
Usage & Guidelines

OO OS Api - Example

- Simple example
  - MyThread inherits and implements method *run* from ThreadFunctor
  - osapi::Mutex is part of MyThread and is default appropriately initialized
  - MyThread is created on the stack in function main()
    - Started via call to start()
    - Waited upon via join()

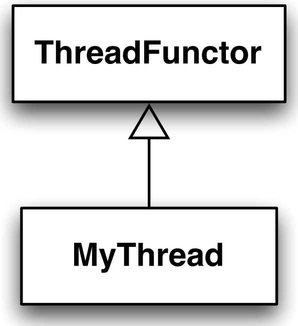


```
class MyThread : public
osapi::ThreadFunctor
{
public:
    MyThread() : running_(true) {}
    virtual void run()
    {
        while (running_) {
            m_.lock();
            // Do stuff
            m_.unlock();
            // Do stuff
        }
    }
private:
    bool         running_;
    osapi::Mutex m_;
};
```



```
int main(int argc, char *argv[])
{
    MyThread myt;
    osapi::Thread t(&myt);
    t.start();

    t.join();
}
```

Typical task structure in event-based system



```
void MyThread::run()
{
   while(running_)
   {
    // get message from message queue
    switch (on state) {
       case ST_IDLE:

          switch (on event) {
              case ID_MSG:
                 // Handle event.
                 break;
              default:
                 break;
          }

          break;
       default:
          break;
    }
   }

}
```

Perform setup here that does not belong in constructor

Get a message, e.g. msgQueue->receive()

Switch on state, each state has separate handlers

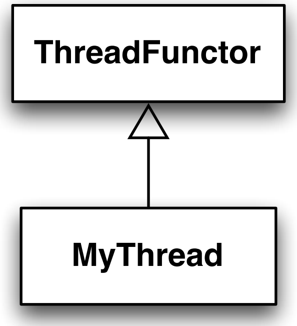
Handle each ”type” of message separately

Signal design error: Thread received something it did not expect

Execution should never reach this point

OS Api used with MsgQueues

- Thread class using the MsgQueue concept



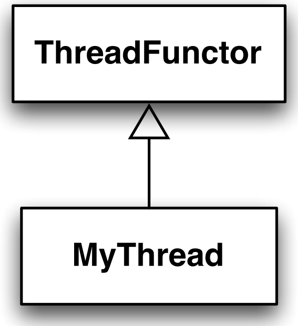
Messages that can be received = Class Interface

Thread function and message handler

Thread states

```
class MyThread : public osapi::ThreadFunctor
{
public:
    enum MsgID {
        ID_MSG,
        ID_TERMINATE
    }
    // Other functions
    MyThread();
protected:
    virtual void run();
private:
    void handleMsg(unsigned long id, osapi::Message* msg);

    enum State {
        ST_IDLE,
        ST_RUNNING
    };

    osapi::MsgQueue mq_;
  State           state_;
};
```

Typical task structure in event-based system



```
void MyThread::run()
{
    // Initial one-time setup
    while (running_) {
        {
            // Wait for message (e.g. mail)
            unsigned long id;
            osapi::Message* msg = mq_.receive(id);
            handleMsg(id, msg);
            delete msg;
        }
    }
}
```

Thread loop

```
void MyThread::handleMsg(unsigned long id,
                        osapi::Message* msg)
{
   switch (state_) {
        case ST_IDLE:
            handleMsgStateIdle(id, msg);
            break;
        default:
            break;
   }
}
```

Switch on state and handle it

Switch on event and handle it

```
void MyThread::handleMsgStIdle(unsigned int
long,
                               osapi::Message*
msg)
{
   switch(id)
   {
        case ID_MSG:
            // handle firstType-messages
            handleStIdleIdMsg(msg);
            break;

        case ID_TERMINATE:
            // handle secondType-messages
            break;

            ...

        default:
            // signal an error
            break;
    }

}
```

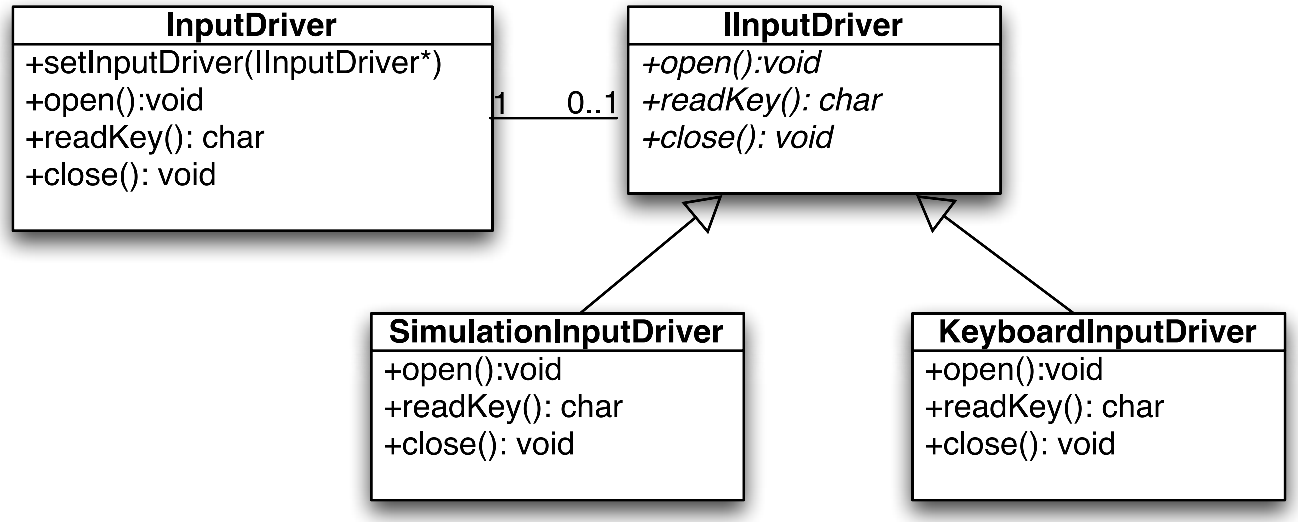
The abstract OO OS API

- The OS API includes the following resources:
  - An abstract ThreadFunctor & Thread class
  - sleep
  - A Timer class (for timeouts)
  - A Time class (simple time arithmetic)
  - Semaphore class (counting)
  - Mutex class
  - Conditional class
  - A ScopedLock class
  - A Completion class
  - A Log System
  - A Message Queue class
- Use (or extend) this to build generic, object-oriented applications

Design/Implementation hint

The Strategy pattern - An example for use in development

- Change strategy
  - KeyboardInputDriver
    - Real thing - requires target
  - SimulationInputDriver
    - Emulate condition on target



```
int main()
{
    InputDriver id;
    if(simul) // Specified somewhere
        id.setInputDriver(new SimulationInputDriver);
    else
        id.setInputDriver(new KeyboardInputDriver);

    // Simple example to illustrate usage
    id.open();
    while(...)
    {
        char c = id.readKey();
        processInput(c); // Process the incoming key
input
    }
    id.close();
}
```