Wiselib School 2010

Session 2
Wiselib Programming Basics

Tobias Baumgartner

Institute of Operating Systems and Computer Networks
Handout on http://www.ibr.cs.tu-bs.de/alg/winterschool/

October 2010

Outline

Template Based Design

```
1 class iSenseRadioModel {
2   int enable_radio() {}
3 }
```

```
1 class ShawnRadioModel {
2   int enable_radio() {}
3 }
```

```
1 template<typename Radio_P>
2 class Algorithm
3 {
4   typedef Radio_P Radio;
5
6   int init( Radio& radio ) {
7     radio_ = &radio;
8     radio_->enable_radio();
9   }
10
11  Radio::self_pointer_t radio_;
12 }
```

```
1 Algorithm<iSenseRadioModel> algrithm_isense;
2 Algorithm<ShawnRadioModel> algrithm_shawn;
```

Template Based Design

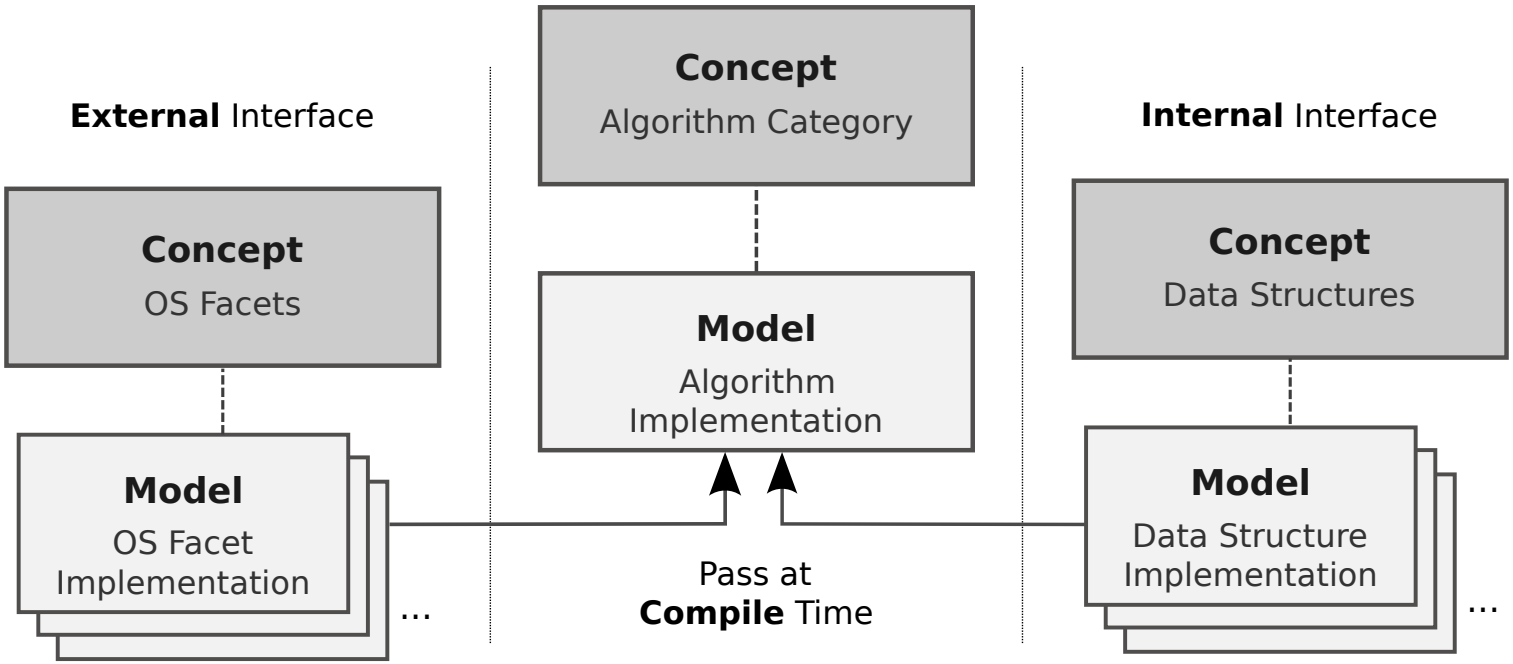
```
1 class iSenseRadioModel {
2   int enable_radio() {}
3 }
```

```
1 class ShawnRadioModel {
2   int enable_radio() {}
3 }
```

```
1 template<typename Radio_P>
2 class Algorithm
3 {
4   typedef Radio_P Radio;
5
6   int init( Radio& radio ) {
7     radio_ = &radio;
8     radio_->enable_radio();
9   }
10
11  Radio::self_pointer_t radio_;
12 }
```

```
1 Algorithm<iSenseRadioModel> algrithm_isense;
2 Algorithm<ShawnRadioModel> algrithm_shawn;
```

Architecture



Outline

Outline

What is a Facet?

- **Connection** between algorithms and OS
- **OS Facets** (Concepts)
  - OS Facet
  - Radio Facet
  - Timer Facet
  - ...
- For **each** supported OS at least **one model** per facet
  - iSenseOsModel
  - ContikiRadioModel
  - ShawnTimerModel
  - ...
- **Possible** to provide **muliple models** per facet
  - ContikiRimeRadioModel
  - Contiki6LowPanRadioModel
  - ...

OS Facet Overview

	OS	Radio (TX Power Ext.) (RSSI/LQI Ext.)	Timer	Logging	Clock (Set Clock Ext.)	Serial Comm.	Position	
WP2 OSA	⊕	○	○	⊕				**4/7**
Contiki	⊕	○	⊕	⊕		○		**5/7**
TinyOS	⊕			⊕				**2/7**
iSense	⊕	⊕ ● ●	⊕	⊕	⊕ ●	⊕	○	**7/7**
ScatterWeb	⊕	○	⊕	⊕				**4/7**
Shawn	⊕	⊕ ● ●	⊕	⊕	⊕		⊕	**6/7**

(⊕ = fully supported, ○ = works / proof of concept, ● = with extension)

Exchangeability with Algorithms

- Basic design issue: **Flexibility**
- Pass an **algorithm** where a **facet** is expected
- Examples
  - Pass routing algorithm where radio is expected
    $\Rightarrow$ Enable flexible multihop neighborhoods
  - Pass time-synchronization algorithm where clock is expected
    $\Rightarrow$ Enable system-wide time basis
  - Pass localization algorithm where position is expected
    $\Rightarrow$ Only some nodes in the network need to know their position
  - Pass routing-based debug model where debug facet is expected
    $\Rightarrow$ Debug nodes that are not connected to a gateway position
- Advantage: Totally **transparent** for algorithm

Outline

The OS Facet

```
1 concept OsFacet {
2   typedef ... size_t;
3   typedef ... block_data_t; // "byte"-like type for buffers
4   enum ReturnValues { SUCCESS, EUNSPEC, ... }; // Define constants for return values
5
6   typedef ... Radio; // Wireless communication facet
7   typedef ... Timer;
8   typedef ... Debug; // Send debug messages
9
10  static const Endianess endianess; // WISELIB_LITTLE_ENDIAN or WISELIB_BIG_ENDIAN
11 }
```

- **Only** facet which does **not need** to be **instantiated**
- Provide **type definitions** and **constants**
- Platform properties (endianess, size type, ...)
- Constants for return values
  - Include at least `SUCCESS` and `ERR_UNSPEC` (unspecified error)
  - May/will include more, similar to `errno`
- Default types for basic OS Facets

Radio Facet (1 of 4)

- Design issues
  - Abstraction to underlying hardware radio
  - Complex routing algorithms
  - *Virtual* radio providing *virtual* ids
- Send messages to other nodes
- Callback registration for received messsages
- Provide node id (and its **type**!)
  - Node id type is defined **per radio**
  - E.g., provide IP addresses, but run on 16-bit addresses
  - Only restriction: Be passed to `sizeof()`

Radio Facet (2 of 4)

```
1   concept RadioFacet {
2     typedef ... node_id_t;
3     typedef ... block_data_t;
4     typedef ... size_t;
5     typedef ... message_id_t;
```

- Ability to provide **arbitrary** node ID types
- Message ID type to identify received messages

```
9     enum SpecialNodeIds { BROADCAST_ADDRESS = ..., NULL_NODE_ID = ... };
10    enum Restrictions { MAX_MESSAGE_LENGTH = ... };
```

- Basic constants for broadcasting and unknown nodes
- Maximal message length defined **per radio**

```
11    int enable_radio();
12    int disable_radio();
```

- Turn on/off radio
- Return SUCCESS or error code if failed

Radio Facet (3 of 4)

```
13      int send(node_id_t receiver, size_t len, block_data_t *data );
```

- Send message to `receiver` (either unicast or broadcast)
- Return `SUCCESS` or error code if failed

```
14      node_id_t id();
```

- Return node id: Can be of **arbitrary** type

```
15    template<class T, void (T::*TMethod)(node_id_t, size_t, block_data_t*)>
16    int reg_recv_callback(T *obj_pnt);
17
18    int unreg_recv_callback(int cid);
19  }
```

- Callback registration: Return *callback id* (or -1 if failed)
- Pass *callback id* to unregister callback

Radio Facet (4 of 4) - Derived Concepts

- VariablePowerRadio

```
1       typedef ... TxPower;
2
3       int set_power(TxPower p);
4       TxPower power();
```

  - Set transmission power
  - Read out TX power to work on value (increment, decrement, ...)

- ExtendedDataRadio

```
1       typedef ... ExtendedData;
2
3       template<class T, void (T::*TMethod)(node_id_t, size_t, block_data_t*,
4                                            ExtendedData&)>
5       int reg_recv_callback( T *obj_pnt );
```

  - Register **receive** method with additional parameter
  - Extended data can be LQI, RSSI, ...
    $\rightarrow$ Again a concept with different `ExtendedData`-models

Timer Facet

- Event mechanism
- *Wait for given time, then call me back...*

```
1   concept TimerFacet {
2     typedef ... millis_t;
3
4     template<typename T, void (T::*TMethod)(void*)>
5     int set_timer(millis_t millis, T *obj_pnt, void *userdata);
6   }
```

- Time given in milliseconds
- Callback registration: Call passed method in given time
- `userdata` is passed on callback

Debug/Logging Facet

- Write out debug or logging data
- Equivalent to `printf()`

```
1    concept DebugFacet
2    {
3      void debug( const char *msg, ... );
4    }
```

- Only one method: `debug(...)`
- Usage as `printf()`
  ⇒ debug_->debug("print an int: %d", my_int);

Clock Facet

- Access to system time
- Type defined by model (platform dependent)

```
1   concept ClockFacet {
2     typedef ... time_t;
3
4     enum ClockSpecificData { CLOCKS_PER_SEC = ..., };
5
6     time_t time();
7   }
```

- Only one method: `time()`
- Number of clock tics per second (`CLOCKS_PER_SEC`):
  $\rightarrow$ Deal with platform independent time calculations

- Derived Concept: Settable Clock facet

```
1     int set_time( time_t time );
```

  - Set time (e.g., for time-synchronization)
  - Currently only implemented for iSense

Outline

Facet Structure

- **Construction** of facets **system dependent**
  - Shawn: A facet needs to know to which processor it belongs
  - iSense: Require access to `isense::Os`
  - Contiki: Only calls to C functions
- Each system with **own constructors**
- Generic Wiselib Application
  - **Construction** must be **hidden** for user
  - Solution: Template based **facet provider**
- Direct Integration
  - Facets are **known** to user
  - **Directly** initialize facets

Generic Wiselib Application (1 of 2)

- Template `FacetProvider`
  → Internals in Session 4

```
1       template<typename OsModel_P,
2                typename Facet_P>
3       class FacetProvider {
4         static Facet& get_facet( AppMainParameter& os );
5       }
```

- Template **specialization** for different platforms
- Method `get_facet()` returns **reference to facet**

```
1       void init( Os::AppMainParameter& value )
2       {
3          radio_ = &wiselib::FacetProvider<Os, Os::Radio>::get_facet( value );
4        }
5       ...
6       Os::Radio::self_pointer_t radio_;
```

- See `wiselib.svn/applications/*` for **examples**!

Generic Wiselib Application (2 of 2)

- Special Issue: The `self_pointer_t`

```
1       void init( Os::AppMainParameter& value )
2       {
3           radio_ = &wiselib::FacetProvider<Os, Os::Radio>::get_facet( value );
4        }
5       ...
6       Os::Radio::self_pointer_t radio_;
```

- Each facet/algorithm provides `self_pointer_t`
- Access via `radio_->enable_radio()`
- Usually, this is just a pointer
  → `typedef self_t* self_pointer_t`
- For C systems, this can be used to **optimize** code space

iSense Application

- iSense facets usually expect `isense::Os` in constructor

```
1   template<typename OsModel_P>
2   class iSenseRadioModel
3     : public isense::Receiver
4   {
5     iSenseRadioModel( isense::Os& os )
6       : os_(os)
7     {
8       os_.dispatcher().add_receiver( this );
9     }
10    ...
11  }
```

- Directly used as members

```
1   #include "external_interface/isense/isense_radio.h"
2   typedef wiselib::iSenseOsModel Os;
3
4   class iSenseDemoApplication {
5
6     iSenseDemoApplication( isense::Os& os )
7       : isense::Application(os),
8         radio_( os )
9     {}
10
11    Os::Radio radio_;
12  }
```

- See `wiselib.svn/iapps/*` for **examples**!

Shawn Application

- Shawn facets usually expect `ShawnOs` in constructor
  → Defined in `external_interface/shawn/shawn_types.h`

```
1       template<typename OsModel_P>
2       class ShawnRadioModel {
3         ShawnRadioModel( ShawnOs& os )
4           : os_(os)
5         {}
6         ...
7         ShawnOs& os_;
```

- Directly used as members

```
1       #include "external_interface/shawn/shawn_radio.h"
2       typedef wiselib::ShawnOsModel Os;
3
4       class WiselibExampleProcessor
5         : public virtual ExtIfaceProcessor {
6
7         WiselibExampleProcessor()
8           : wiselib_radio_ ( os_ )
9         {}
10
11        void boot() { os_.proc = this; }
12
13        ShawnOs os_;
14        Os::Radio wiselib_radio_;
```

- See `wiselib.svn/shawnapps/*` for **examples**!

Outline

Outline

Radio Facet

Concept

```
concept RadioFacet {
  // ...

  typedef ... block_data_t;
  typedef ... size_t;
  typedef ... message_id_t;

  enum SpecialNodeIds {
    BROADCAST_ADDRESS = ...,
    NULL_NODE_ID = ...
  };
  enum Restrictions {
    MAX_MESSAGE_LENGTH = ...
  };

  int enable_radio();
  int disable_radio();
  int send(
      node_id_t receiver,
      size_t len, block_data_t *data
  );
  node_id_t id();
  int reg_recv_callback(...);
  int unreg_recv_callback(int idx);
}
```

Implementations

- Radio models for all platforms
- Virtual and encrypting radio
- Routing algorithms

Usage example

```
class MyApp {
  void init(Os::AppMainParameter& amp) {
    radio_ = &wiselib::FacetProvider<...>(amp);

    radio_->enable_radio();
    radio_->reg_recv_callback<MyApp, &MyApp::
        recv>(this);

    char *m = "Hello World";
    radio_->send(
      Os::Radio::BROADCAST_ADDRESS,
      strlen(m), (Os::Radio::block_data_t*)(m)
    );
  }

  void recv(Os::Radio::node_id_t from,
    Os::Radio::size_t size,
    Os::Radio::block_data_t *buf) {
    debug_->debug("got %s from %d", from, buf);
  }
}
```

Problems when Sending Messages

- How to pass a message when network is heterogeneous?
  - Different bit widths
  - Alignment issues
  - Byte order
- How to identify own messages?
  - Radio may be used by multiple algorithms
- How to flexibly register a callback?
  - No virtual inheritance
  - No C function pointers

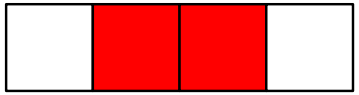
Outline

Heterogeneity

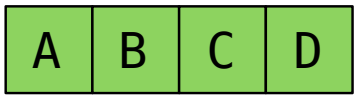
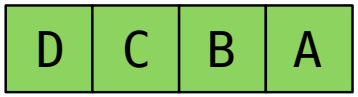
Bit Width

Write `int`	
Jennic	
MSP430	

Alignment

`uint16_t` at odd address	
Shawn (Desktop)	
MSP430	

Byte Order

Write an `uint32_t`	
Big Endian	
Little Endian	

The "Double Problem"

- IEEE Standard for Floating-Point Arithmetic (IEEE 754)
  - Single precision: 4 bytes
  - Double precision: **8 bytes**
- E.g., msp430-g++ (default)
  - Single precision: 4 bytes
  - Double precision: **4 bytes**

Solution

- Make use of **fixed size** data types
  - Include header `stdint.h`
  - Use data types `uint16_t`, `int32_t`, ...
- Provide "clever" read/write methods
  - Take care of platform differences
  - Do the right thing for all datatype/platform combinations
- Template specialization
  - Only needed conversions will be compiled
  - Easy to add new conversion rules for new platforms/datatypes

$\Rightarrow$ Developer does not have to worry about platform details!

Serialization

Read/write an `uint16_t`

```
1 block_data_t buffer[...];
2
3 uint16_t read_value() {
4   return read<OsModel, block_data_t, uint16_t>(buffer);
5 }
6
7 OsModel::size_t write_value(uint16_t value) {
8   return write<OsModel, block_data_t, uint16_t>(buffer, value);
9 }
```

- `read` and `write` care for heterogeneity
- Template specialization for each specific platforms (**possible**)
- Where not specialized, use default implementation

Templated Serialization provided by the Wiselib

```
1  template<typename OsModel_P, typename BlockData_P, typename Type_P>
2  inline Type_P read( BlockData_P *target )
3  {
4     return Serialization<OsModel_P, OsModel_P::endianness, BlockData_P, Type_P>
5               ::read( target );
6  }
7
8  template<typename OsModel_P, typename BlockData_P, typename Type_P>
9  inline void read( BlockData_P *target, Type_P& value )
10 {
11    value = Serialization<OsModel_P, OsModel_P::endianness, BlockData_P, Type_P>
12              ::read( target );
13 }
14
15 template<typename OsModel_P, typename BlockData_P, typename Type_P>
16 inline typename OsModel_P::size_t write( BlockData_P *target, Type_P& value )
17 {
18    return Serialization<OsModel_P, OsModel_P::endianness, BlockData_P, Type_P>
19              ::write( target, value );
20 }
```

- Basic functions for `read()` and `write()`
- Use Serialization class: Passing OS, endianness, block data, type
- Template specialization: Automatically generate platform dependent code

Example: Generic Implementation and Specialization

- Generic implementation: Used by default

```
1  template <typename OsModel_P, Endianness, typename BlockData_P, typename Type_P>
2  struct Serialization
3  {
4    static inline size_t write( BlockData *target, Type& value )
5    {
6      for ( unsigned int i = 0; i < sizeof(Type); i++ )
7        target[sizeof(Type) - 1 - i] = *((BlockData*)&value + i);
8      return sizeof(Type);
9    }
10   ...
```

- Specialization for big endian (default for all data types)

```
1  template <typename OsModel_P, typename BlockData_P, typename Type_P>
2  struct Serialization<OsModel_P, WISELIB_BIG_ENDIAN, BlockData_P, Type_P>
3  {
4    static inline size_t write( BlockData *target, Type& value )
5    {
6      for ( unsigned int i = 0; i < sizeof(Type); i++ )
7        target[i] = *((BlockData*)&value + i);
8      return sizeof(Type);
9    }
10   ...
```

Example: Floating Point Specialization

- Template specialization for `double` values

```
1  template <typename OsModel_P,
2            typename BlockData_P>
3  struct Serialization<OsModel_P, WISELIB_LITTLE_ENDIAN, BlockData_P, double>
4  {
5  public:
6      static inline double read( BlockData *target )
7      {
8          return FpSerialization<OsModel, WISELIB_LITTLE_ENDIAN, BlockData, double,
9            sizeof(double)>::read( target );
10     }
11
12     static inline size_t write( BlockData *target, double& value )
13     {
14         return FpSerialization<OsModel, WISELIB_LITTLE_ENDIAN, BlockData, double,
15           sizeof(double)>::write( target, value );
16     }
17 };
```

- Automatically adapt to platform via

  `sizeof(double)`

  as template argument
- FpSerialization: Same principle as Serialization class

Outline

Message Identification

- Very simple concept for messages:

Each message has an identifier in the first byte(s)

- Message id type is defined in radio

Always use `Radio::message_id_t`

- All radio facets are adjusted for same `message_id_t`

Currently `uint8_t`, may change to `uint16_t` soon

- See
  www.wiselib.org/wiki/design/messages/id_allocation

Accessing Message IDs

- Make use of **serialization**, also in algorithms!

```
1 #include "util/serialization/simple_types.h"
2
3 void
4 MyAlgorithm<OsModel_P, Radio_P, Debug_P>::
5 receive( node_id_t from, size_t len, block_data_t *data )
6 {
7   message_id_t msg_id = read<OsModel, block_data_t, message_id_t>( data );
8   if ( msg_id == MyMessageId )
9   { ... }
```

Generic Message Composition

- In own messages, make use of **serialization** and **type definitions**
- Define buffer array as **only** data member

```
1 template<typename OsModel_P, typename Radio_P>
2 class MyMessage {
3   message_id_t msg_id()
4   { return read<OsModel, block_data_t, message_id_t>( buffer ); };
5
6   void set_msg_id( message_id_t id )
7   { write<OsModel, block_data_t, message_id_t>( buffer, id ); }
8
9   node_id_t source()
10  { return read<OsModel, block_data_t, node_id_t>(buffer + SOURCE_POS); }
11  ...
12  size_t buffer_size() { return /* size of message */; }
13  ...
14  enum data_positions {
15    MSG_ID_POS  = 0,
16    SOURCE_POS  = sizeof(message_id_t),
17    NEXT_POS    = SOURCE_POS + sizeof(node_id_t),
18    ... };
19
20  block_data_t buffer[MAX_MESSAGE_LENGTH];
```

- When sending message over radio, **cast message to block data**:

```
1 radio().send( destination, message.buffer_size(), (block_data_t*)&message );
```

- On reception, **cast block data to message**:

```
1 MyMessage *message = (MyMessage *)buffer;
```

Outline

Design Issues for Callback Structure

- Ability to register callback (method pointer) in another class
- Requirements
  - Flexible
  - Fast
  - Small (in code space)
- **Our Solution:** Delegates
- Advantages
  - Register arbitrary methods (only signature must fit)
    $\Rightarrow$ Flexible
  - Mostly resolved at compile time
    $\Rightarrow$ Fast and small in code space
- Delegate internals $\rightarrow$ Session 4

Example: Register callback method at timer

- Timer Facet

```
1 concept TimerFacet {
2   template<typename T, void (T::*TMethod)(void*)>
3   int set_timer(millis_t millis, T *obj_pnt, void *userdata);
4 }
```

- Method signature: `void METHOD(void*)`
- Callback registration at Timer

```
1 template <..., typename Timer, ...>
2 class MyClass {
3   typedef MyClass<...,Timer,...> self_t;
4
5   int init
6   {
7     timer_->template set_timer<self_t, &self_t::timer_elapsed1>( 1000, this, 0 );
8     timer_->template set_timer<self_t, &self_t::timer_elapsed2>( 2000, this, 0 );
9   }
10
11  void timer_elapsed1(void* data);
12  void timer_elapsed2(void* data);
13
14  Timer::self_pointer_t timer_;
15 }
```

Outline

Problems with STL

- STL uses all kinds of C++ features like...
  - `new`/`delete`
  - RTTI
  - Exceptions

  $\Rightarrow$ Bad, some platforms do not support those!
- That's even true for other "slim" versions like the uSTL (`http://ustl.sourceforge.net`)

Solution: pSTL

- "pico" version of the STL
- implements a subset of the STL
- but usable on limited platforms
  - does not require `new`/`delete`, exceptions, RTTI
  - not even a dynamic memory
  - resource-efficient implementation
- replace pSTL with "normal" STL any time if you need something pSTL doesn't provide

→ Almost full STL power even on limited nodes

→ Code can be easily ported between STL and pSTL

pSTL Design

- STL-Code works with small modifications
- But don't use `const`, `new`, etc...
  (code size and portability)
- Only `++iter` supported, not `iter++`
  (easier to maintain)
- Currently only statically sized containers available (will change soon)
  (do not require dynamic memory)
- Some details (like allocator passing) are different
  (because allocation is different)

Implementations

Containers

- `vector_static`
- `list_static`
- `map_static_vector`
- `priority_queue`
- `queue_static`
- `set_static`
- `pair`

Algorithms

- `for_each`
- `find` / `search`
- `min` / `max`
- `copy`
- heap operations
- sorting
- etc...

Example: MapStaticVector

```
1 #include <iostream>
2
3 #include "util/pstl/map_static_vector.h"
4 #include "external_interface/pc/pc_os_model.h"
5
6 typedef wiselib::PCOsModel Os;
7 typedef wiselib::MapStaticVector<Os, int, const char*, 5> map_t;
8
9 int main(int argc, char** argv) {
10   map_t map;
11
12   map[1] = "first";
13   map[9876] = "over 9000";
14   map[42] = "the answer";
15   map.erase(9876);
16   map[815] = "flight no.";
17
18   map_t::iterator iter;
19   for(iter = map.begin(); iter != map.end(); ++iter) {
20     std::cout << iter->first << " => " << iter->second << "\n";
21   }
22   return 0;
23 }
```