



# Connecting to External Environments

---

## An SGIO Tutorial



# Overview

---

- Introduction
  - What is SGIO and why is it useful
  - General structure of SGIO
- Using SGIO
  - Classes and methods
- Building Applications with SGIO
  - Headers, libs to include, etc.



# What is SGIO?

---

- Soar General Input Output
- Want to attach Soar to an external environment via the input-link and output-link
- SGIO enables the communication by mapping between objects the environment side and the io-link in Soar



# Benefits of SGIO

---

## ■ Alternatives

- Communicate with environment via Tcl
  - Slow, requires intimate knowledge of Tcl, referencing Tcl objects in your C code, or using a compile library like SWIG
- Have environment communicate directly with the Soar kernel
  - Requires lots of bookkeeping, i.e. tracking timetags
  - Mapping the io-link directly is tedious



# Benefits of SGIO

---

- Abstracts away from kernel details
- Mapping to io-link is easy
- Can use TSI for debugging
- Can avoid Tcl if not debugging (fast!)
- Can run on multiple machines
- Can easily switch between debugging and high-performance modes

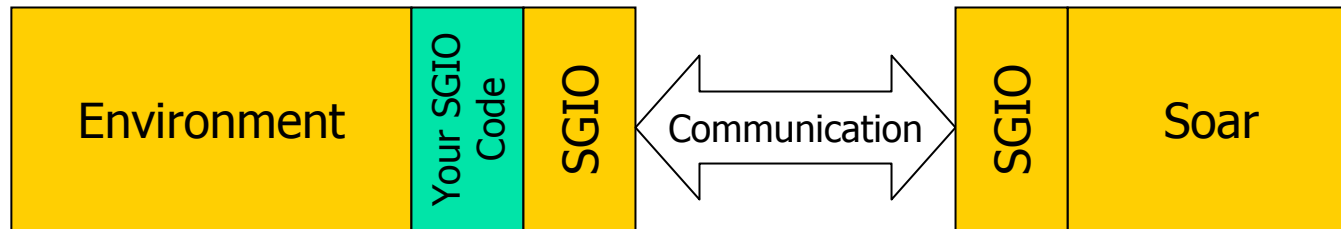


# SGIO Framework

---

- Environment (via SGIO) initiates all communication with Soar
  - Environment initializes stuff, puts things on input-link, reads things from output-link
  - Soar does not make calls to the environment; it simply responds to commands from it
    - The environment does all the pushing and pulling of io-link WMEs

# SGIO Framework





# SGIO Framework: Classes

---

- Three main classes implemented in C++:
  - Soar: represents connection to Soar
  - Agent: represents a particular agent running in a Soar connection
  - WorkingMemory: interface which handles some of the bookkeeping associated with the agent's working memory





# Using SGIO: Overview

---

- Initialization
- Manipulating the input-link
- Running an Agent
- Manipulating the output-link
- Shutting down
- Compiling an application



# Initialization

---

- Create a connection to Soar
- Create agents
- Load productions
- Creating working memory interface



# Initialization: Creating a connection to Soar

---

- Can create either API Soar or SIO Soar
  - API compiles the Soar kernel directly into the application
    - Very fast
    - No TSI window for debugging
    - Single machine
  - SIO communicates with Soar via sockets
    - Slower
    - Get TSI for debugging
    - Multiple machines
  - This choice does not affect later code

# Initialization: Creating a connection to Soar

- If want API Soar

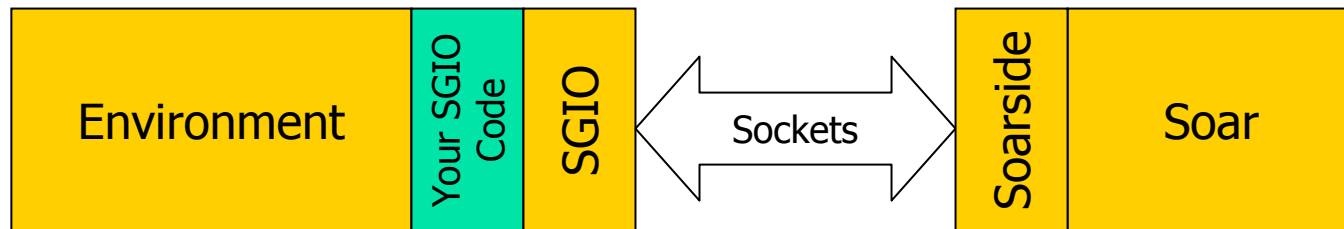
- `sgio::Soar* soar = new sgio::APISoar();`



# Initialization: Creating a connection to Soar

- If want SIO Soar

- `sgio::Soar* soar = new  
sgio::SIOSoar("127.0.0.1", 6969, true);`
- Specify IP address, port number, lockstep
- Lockstep is running Soar synchronously





# Initialization: SIOSoar and Soarside

---

- If using sockets, need something to connect to
- Soarside is a program which handles SGIO communications
- Must already be running on target machine



# Initialization: Creating agents

---

- An instance of Soar can run multiple agents (i.e. multiple eaters or multiple tanks)
- Create the agent from the soar connection
- `sgio::Agent* agent = soar->CreateAgent("my-agent");`
  - Specify the agent's name

# Initialization: Loading productions



- Assuming the agent is already created:
- `agent->LoadProductions("my-agent.soar");`
  - Specify file to load
  - Can only specify a file name, not a path
- If using API Soar, assumes file is located in a subdirectory of the cwd called "agents"
- If using SIO Soar, assumes file is located in the cwd
- If loading fails, can't actually detect that at this point





# Initialization: Creating Working Memory Interface

---

- Create a working memory interface for a particular agent
- `sgio::WorkingMemory* mem = new  
sgio::WorkingMemory(agent);`
  - Specify agent to create the interface for



# Manipulating the Input-Link

---

- Adding WME's
- Updating existing WME's
- Removing WME's
- Sending changes to Soar



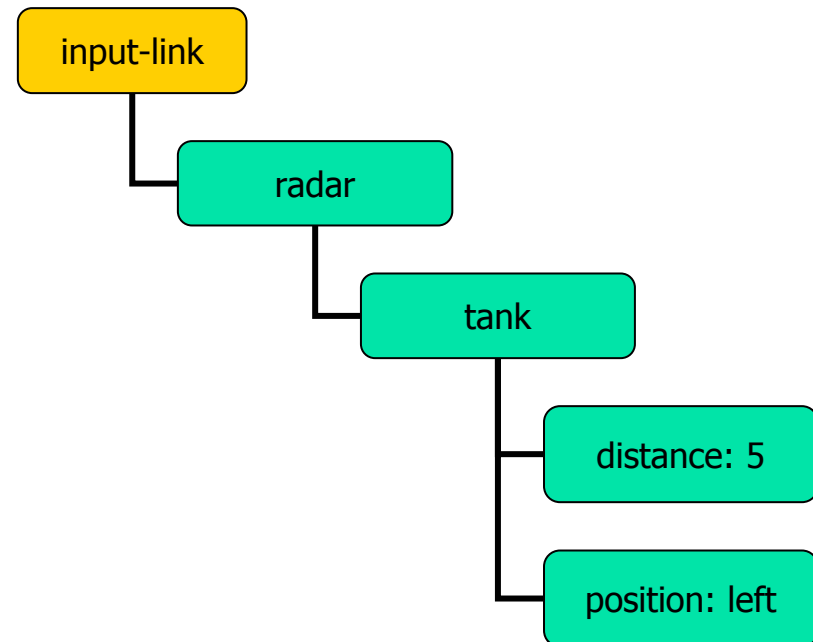
# I-Link: Adding WME's

---

- Four kinds of WME's
  - ID WME's: just a name, no value
  - Attribute-Value pairs: can specify data type
    - Integer WME's
    - Float WME's
    - String WME's
- Structure can be arbitrarily deep

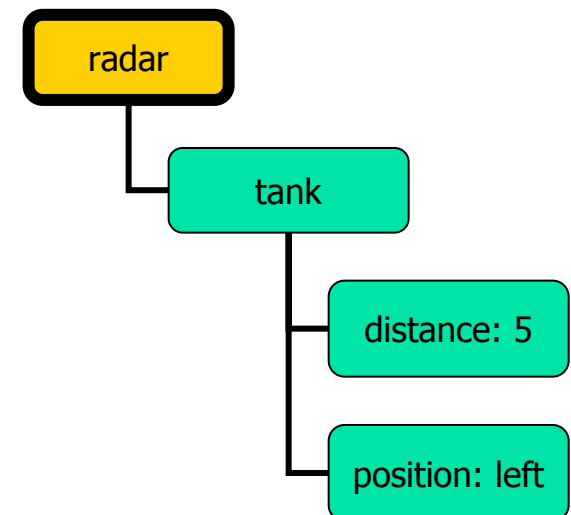
# I-Link: Adding WME's

- Say we want to add this structure (from TankSoar)
- Radar and tank are ID WME's
- Distance and position are attribute-value pairs



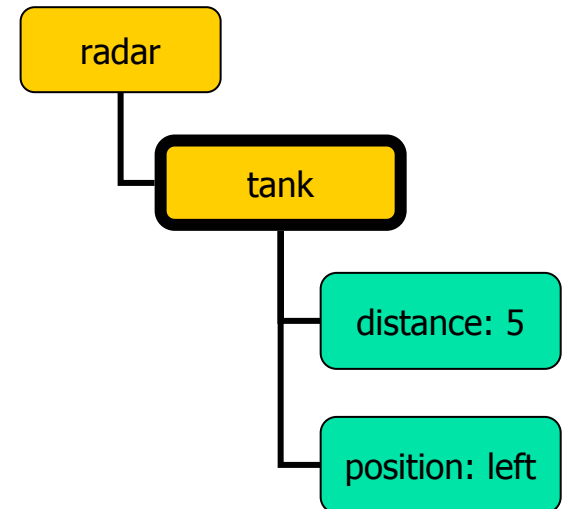
# I-Link: Adding WME's

- Create a SoarId object in the memory interface for radar
- `sgio::SoarId* radarId = mem->CreateIdWME(mem->GetILink(), "radar");`
  - Specify WME's parent and WME's name
  - Parent is the input-link



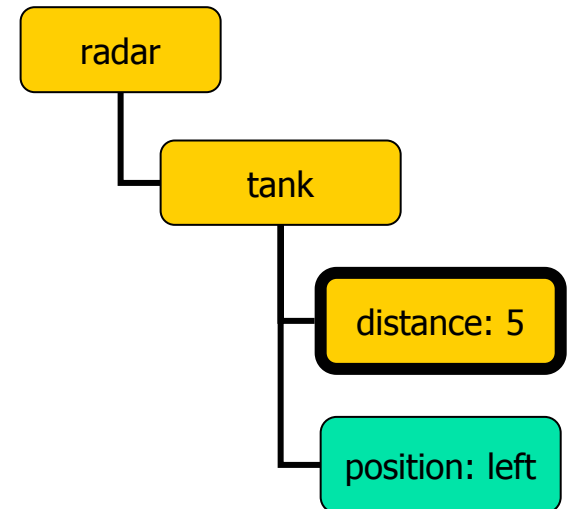
# I-Link: Adding WME's

- Add Tank ID WME in the same way
- `sgio::SoarId* tankId = mem->CreateIdWME (radarId, "tank");`
  - Parent is the radar WME this time



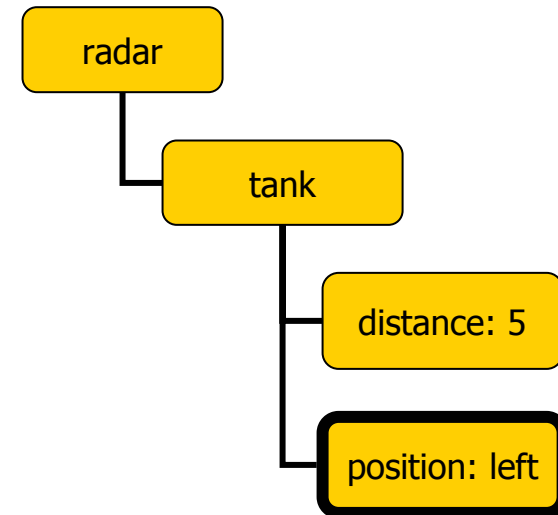
# I-Link: Adding WME's

- Create an integer WME for distance off of the tank WME
- `sgio::IntElement* distance = mem->`  
`CreateIntWME (tankId, "distance", 5);`
  - Specify parent, attribute name, attribute value



# I-Link: Adding WME's

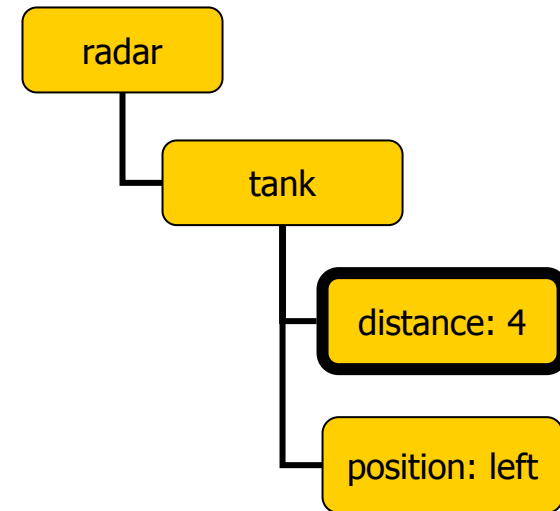
- Create a string WME for position off of the tank WME
- `sgio::StringElement* position = mem->`  
    `CreateStringWME (tankId, "position", "left");`
  - Specify parent, attribute name, attribute value





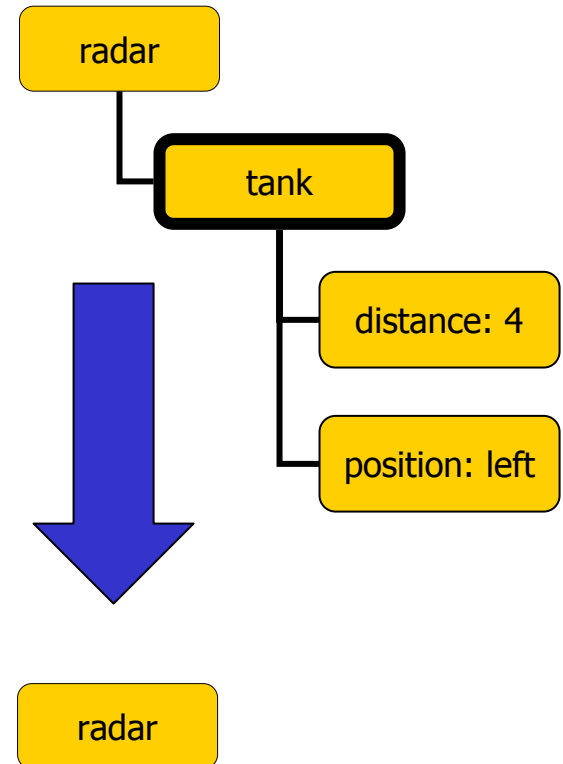
# I-Link: Modifying WME's

- If a WME already exists, we can modify the value
- Say we want to update the distance value to 4
- `mem->Update (distance, 4) ;`
  - Specify SGIO element to update and new value
  - Can do this for any of the attribute-value types



# I-Link: Removing WME's

- We can also remove existing WME's
  - Children of specified WME will also be removed
- `mem->DestroyWME (tankId) ;`
  - Specify SGIO element to remove
- Memory cleanup is handled internally (so any pointers we have to child elements are invalid)





# I-Link: Sending changes to Soar

---

- Once all of the manipulations have been done to the working memory interface, we need to commit those changes so the Soar agent can see them
- `mem->Commit ();`
- Sending all the changes at once avoids repeated overhead of multiple trips to Soar



# Running an agent

---

- Assuming a connection to Soar already exists:
- `soar->RunTilOutput () ;`
  - Runs all agents on this connection for a max of 15 decision cycles
- Can also run a single agent:
- `agent->RunTilOutput () ;`
  - Runs this agent for a max of 15 decision cycles
- “Bug”: always runs 15 decision cycles the first time, even if output is generated sooner



# Manipulating the Output-Link

---

- Checking for waiting commands
- Reading commands
- Getting parameters from commands
- Marking commands as processed



# O-Link: Checking for waiting commands

---

- To see if an agent has any commands waiting on the output-link:
- `bool waiting = agent->Commands ();`



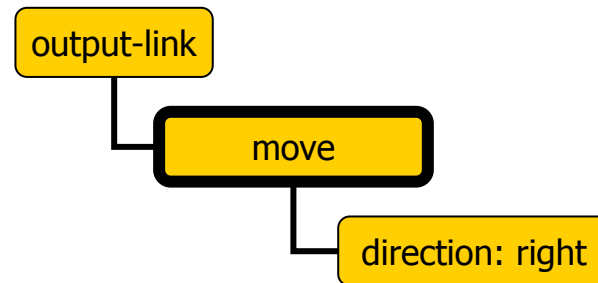
# O-Link: Reading a command

---

- To get a waiting command:
- `std::auto_ptr<sgio::Command> cmd = agent->GetCommand();`
  - Returns a `std::auto_ptr`
  - `std::auto_ptr` is nice because it takes care of its own memory management (i.e. we won't have to delete the command object ourselves)
- If there are multiple commands waiting, will need to loop to get them all

# O-Link: Getting command name

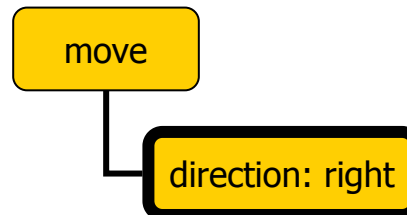
- `std::string name = cmd->GetCommandName();`
- In this case, `name = "move"`
- Note: Structures can only be two-level





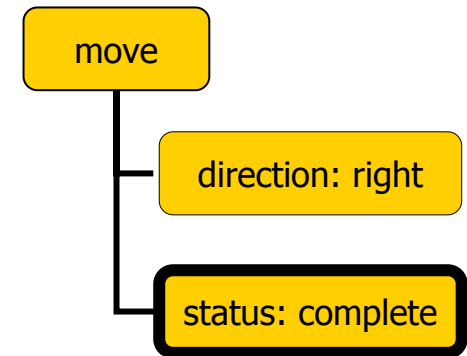
# O-Link: Reading command attribute-values

- We assume that the environment knows the structure of commands
  - Once we know which command we have, can ask for the values of specific attributes
- `std::string value = cmd->GetParameterValue("direction");`
- In this case, `value = "right"`
- May have to convert value to another datatype



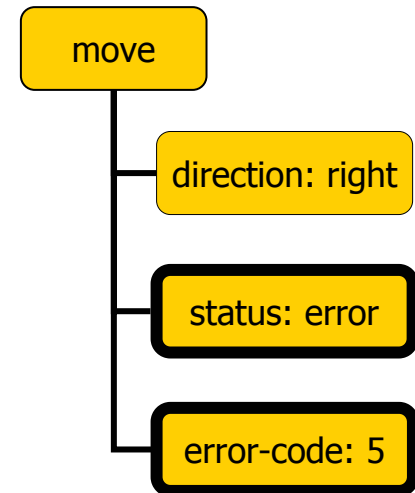
# O-Link: Marking commands as processed

- Once we have read in all the info associated with a command and responded appropriately, we can mark the command as processed
- `cmd->AddStatusComplete();`
- This change must be committed as well, but we usually just commit it with next i-link update



# O-Link: Marking commands as processed

- Or, if there was an error in processing the command (i.e. it was missing some attributes)
- `cmd->AddStatusError();`
- Once a command has been marked as an error, we can add an error code (an integer)
- `cmd->AddErrorCode(5);`

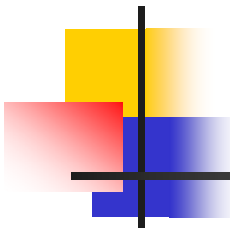




# Shutting down

---

- Simply delete the memory interface and the Soar connection:
  - `delete mem;`
  - `delete soar;`
- Memory elements are cleaned up by the memory interface destructor
- Agents are cleaned up in the Soar connection destructor
- Commands are cleaned up by `std::auto_ptr`



# Building an application: Binaries

---

- `simside.lib`
  - Defines environment-side hooks for communicating with Soar
- `soarside.exe`
  - Connection point if Soar is not embedded
- `sgio_shared.lib`
  - Some shared classes, like messages



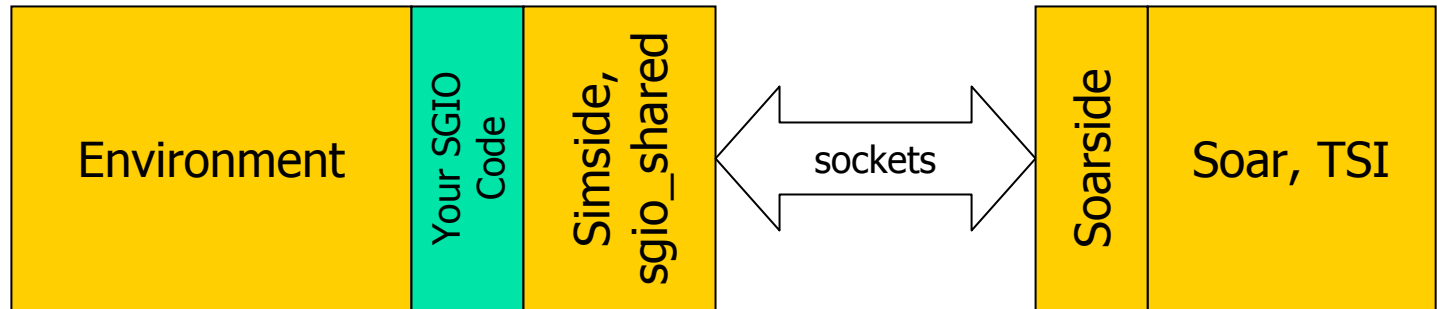
# Building an application

---

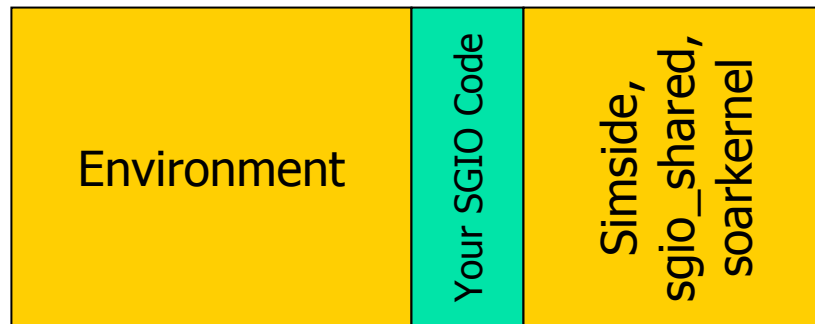
- Application must be compiled multithreaded
- Need to include `simside` and `shared` directories for various headers
- Need to link `simside.lib` and `sgio_shared.lib`
- If using API Soar, need to link `soarkernel.lib` (version 8.4.5)

# Building an application

SIO Soar



API Soar





# Compiling: Headers to include

---

- All explicitly included headers are from Simside:
  - `sgio_siosoar.h`: SIOSoar class
  - `sgio_apisoar.h`: APISoar class
  - `sgio_wmemem.h`: WorkingMemory, Element classes
  - `sgio_command.h`: Command class
  - `sgio_agent.h`: Agent class
- Need shared include directory for things included by these headers, i.e. `thread.h`





# Running an application

---

- SIO Soar
  - Soarside needs Tcl-8.3.x to run Soar, TSI
    - tcl83.dll and tk83.dll must be accessible
  - Doesn't work with Tcl-8.4 (bug in Tcl)
- API Soar
  - Everything is integrated, should just run



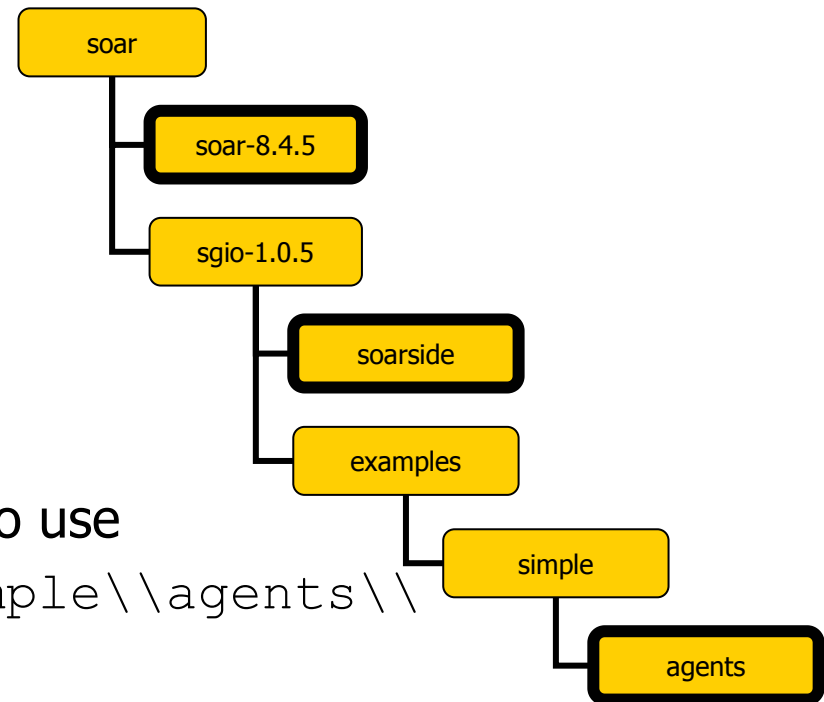
# Running an Application: Soarside.exe

---

- Command line parameters
  - Port number: default 6969
  - Init file: default soarside-init.tcl
  - Example:
    - `soarside.exe 7605 my-init.tcl`
  - Note: arguments must be specified in order
- Init file contents
  - Switch to directory containing Soar
  - source `start-soar.tcl`
  - Switch to directory containing Agents

# Running an Application: Soarside init file example

```
# switch to the location of the version of
# Soar we want to use
cd ..\..\..\soar-8.4.5\
# start Soar
source start-soar.tcl
# switch to the Agents directory we want to use
cd ..\sgio-1.0.5\examples\simple\agents\
```





# Wrapping it up

---

- SGIO Quick Reference sheet contains most of the info from this talk
- For the latest SGIO releases and news and source, or to submit a bug, visit our SourceForge site at:

<http://sourceforge.net/projects/sgio/>



# Questions?

---

