Tutorial 1
Creating an Agent

In this tutorial, you'll learn how to create an agent, as well as an Actionary. The Actionary is the connection between the database and your code, and stores and manages all the agents, objects, and actions within your program.

Before We Begin:

It's expected that you have the code installed. If you do not, please do that first (see the installation Readme).

Let's Begin:

Create a new program. Since this is a basic example, a main.cpp is created. At the top of our code, we need to import a few libraries, specifically, the agent process and Actionary. Place this at the top of your code:

```
#include "agentproc.h"
```

This is the include files needed to get a basic agent up and running. agentproc.h holds the AgentProc class that builds the agent and itself includes all of the other files needed.

Next, we create some global variables to control the agent and Actionary.

```
extern Actionary *actionary; // global pointer to the Actionary
AgentProc* agent;  // global pointer to an agent
parTime *partime;  // global pointer to PAR time class (So we can manipulate time)
char* actionLocation=strdup("PATH TO FOLDER");  // Path to PAR folder
int PAR::dbg = 1; /*< Used to see all the debug information in the code. Set to 0 to disable */
FILE* PAR::file_name = stdout; /*! <The output stream of all debug information */
```

The Actionary and agent variables just point to the Actionary and agent. The Actionary pointer is defined in AgentProc.cpp and is required for any PAR application. Every entity in the virtual world that will be performing actions needs to be associated with an AgentProc. We would recommend creating a subclass of this class to store any additional information required for your application. This would include a pointer to the graphical representation of the agent in animated applications and specific AI. You can create lists of these agents and call their update methods systematically.

Partime allows us to set up timing information for the system. You can have simulations begin at different times of day and speed up the rate of time as well.

The variable actionLocation, points to the place PAR where is installed. It's used to give a starting location for the location of the Python action scripts (which is talked about in more detail in tutorial 2).

Finally, the last two variables are useful for debugging information. We have created a macro, called *par_debug*, which is useful for diagnosing issues with the system. dbg controls whether the macro is used, and file_name provides an output stream for all the information. For this tutorial, we use stdout so all data is displayed to the terminal screen.

Next we move onto the main function. Within this function, we'll set the environment for the agent, and create a basic agent. The main function appears as:

```
int main(void){
        partime = new parTime();  // setup the timing info for the simulation
        partime->setTimeOffset(8,30,30);  // hrs., min., sec. from midnight
        partime->setTimeRate(1);          // how fast should time change

//Creates an actionary
        actionary=new Actionary();
        actionary->init();

        agent=new AgentProc("Agent_0");
        agent->setCapability("ROOT");

        system("PAUSE"); // just holds the output window
}
```

First, we create and set the partime variable to 8:30 am, and have it run at a constant rate of one per time step. Next, we create an Actionary. Finally, an agent is created with the name Agent_0. This agent is given the base capability 'ROOT'. It is important to give agents capabilities; otherwise the system won't know what the agent can and cannot do. By giving the highest level capability to the agent, it can perform all actions that are children of 'ROOT' (see the action tree diagram in the general documentation).

Before compiling and running this code, you'll need to add some project settings:

Additional Include Directories should include paths (using the environmental variables) to:

```
PAR/database
PAR/agentProc
PAR/lwnets
${CONNECTOR_ROOT}/include
${PYTHON_ROOT}/include
${MYSQL_ROOT}/include
```

Additional Library Directories should include paths to:

`PAR/libs`
`${CONNECTOR_ROOT}/debug` (or the Release directory, as appropriate)

Additional Dependencies (under Linker) should include:

databased.lib
agentProcd.lib
lwnetd.lib
mysqlcppconn.lib
python27_d.lib
winmm.lib

Or the release versions, as appropriate

You should now be able to compile and run this first tutorial. Note that we've provided libs and dlls for mysqlpp and python. However, these may not be compatible with different operating systems. If this tutorial cannot be run because of a dll conflict, try compiling your own versions of these files. (Also, remember that the three PAR solutions need to be compiled to generate the .lib files before this tutorial can be run).

This is a very simple start to help make sure that your setup is correct. Additional tutorials will highlight the functionality of PAR.