Programming Assignment: Behavior Trees

Topics and references

- Behavior Trees

Task

You have to implement a few components that can be used for construction of a behavior tree. You can find the complete list of structures and functions to implement in form of pseudocode on the Moodle web page and in the assignment framework that is provided and accessible on the SVN server.

Complete instruction to the assignment and explanation of the method **will be given in the class**.

You implementation must passes all given tests in order to get the full mark for the assignment.

Submission details

Please read the following details carefully and adhere to all requirements to avoid unnecessary deductions. Since submission details will remain the same for all programming assessments, this portion will be skipped in future documents detailing labs and assignments.

Source files

You have to submit the header `functions.h` and source file `functions.cpp`.

```
#ifndef FUNCTIONS_H
#define FUNCTIONS_H

#include <sstream>
#include <string>
#include <list>

#include "data.h"

#define UNUSED(x) (void)x;

namespace AI
{

    // Check the state of a task comparing it with given by parameter
    class CheckState : public Task
    {
        Task checktask;
        State checkstate;

    public:
        CheckState(Task checktask = {}, State checkstate = State::Success)
```

```
            : Task{ "CheckState" }, checktask{ checktask }, checkstate{
checkstate }
        {
        }

        // Your code ...

    };

    // Selector composite
    //     Returns immediately with a success status code
    //     when one of its children runs successfully.
    class Selector : public Task
    {
        std::list<SMART> tasks;

    public:
        Selector(std::initializer_list<SMART> tasks = {})
            : Task{ "Selector" }, tasks{ tasks }
        {
        }

        // Your code ...

    };

    // Sequence composite
    //     Returns immediately with a failure status code
    //     when one of its children fails.
    class Sequence : public Task
    {
        std::list<SMART> tasks;

    public:
        Sequence(std::initializer_list<SMART> tasks = {})
            : Task{ "Sequence" }, tasks{ tasks }
        {
        }

        // Your code ...

    };

    // Random selector composite
    //     Tries a single child at random.
    class RandomSelector : public Task
    {
        std::list<SMART> tasks;

    public:
        RandomSelector(std::initializer_list<SMART> tasks = {})
            : Task{ "RandomSelector" }, tasks{ tasks }
        {
        }

        // Your code ...

    };
```

```
    // Decorators


    // Inverter
    //     Invert the value returned by a task
    class Inverter : public Task
    {
        SMART task;

    public:
        Inverter(SMART task = {})
            : Task{ "Inverter" }, task{ task }
        {
        }

        // Your code ...

    };

    // Succeeder
    //     Always return success, irrespective of what the child node actually
returned.
    //     These are useful in cases where you want to process a branch of a tree
    //     where a failure is expected or anticipated, but you don't want to
    //     abandon processing of a sequence that branch sits on.
    class Succeeder : public Task
    {
        SMART task;

    public:
        Succeeder(SMART task = {})
            : Task{ "Succeeder" }, task{ task }
        {
        }

        // Your code ...

    };

    // Repeater
    //     A repeater will reprocess its child node each time its
    //     child returns a result. These are often used at the very
    //     base of the tree, to make the tree to run continuously.
    //     Repeaters may optionally run their children a set number of
    //     times before returning to their parent.
    class Repeater : public Task
    {
        SMART task;
        int counter;

    public:
        Repeater(SMART task = {}, int counter = 0)
            : Task{ "Repeater" }, task{ task }, counter{ counter }
        {
        }
```

```
        // Your code ...

    };

    // Repeat_until_fail
    //      Like a repeater, these decorators will continue to
    //      reprocess their child. That is until the child finally
    //      returns a failure, at which point the repeater will
    //      return success to its parent.
    class Repeat_until_fail : public Task
    {
        SMART task;

    public:
        Repeat_until_fail(SMART task = {})
            : Task{ "Repeat_until_fail" }, task{ task }
        {
        }

        // Your code ...

    };

} // end namespace

#endif
```

```
#include "functions.h"

namespace AI
{

} // end namespace
```

Compiling, executing, and testing

Run `make` with the default rule to bring program executable `main.out` up to date:

```
$ make
```

Or, directly test your implementation by running `make` with target `test`:

```
$ make test
```

If the `diff` command in the `test` rule is not silent, then one or more of your function definitions is incorrect and will require further work.

To make a memory leak test, run `make` with target `leak`:

```
$ make leak
```

Make sure that the output does not show any memory related issue.

File-level documentation

Every **edited by student** source and header file *must* begin with a *file-level* documentation block. This documentation serves the purpose of providing a reader the raison d'être of this source file at some later point of time (could be days or weeks or months or even years later). This module will use Doxygen to tag source and header files for generating html-based documentation. An introduction to Doxygen and a configuration file is provided on the module web page. Here is a sample for a C++ source file:

```
/*!*****************************************************************************
\file    functions.cpp
\author  Vadim Surov, <Your Name>
\par     DP email: vsurov\@digipen.edu, <Your Email>
\par     Course: CS380
\par     Section: A
\par     Programming Assignment 10
\date    04-30-2021

\brief
  This file has declarations and definitions that are required for submission
*******************************************************************************/
```

Function-level documentation

Every function that you declare and define and submit for assessment must contain *function-level documentation*. This documentation should consist of a description of the function, the inputs, and return value. In team-based projects, this information is crucial for every team member to quickly grasp the details necessary to efficiently use, maintain, and debug the function. Certain details that programmers find useful include: what does the function take as input, what is the output, a sample output for some example input data, how the function implements its task, and importantly any special considerations that the author has taken into account in implementing the function. Although beginner programmers might feel that these details are unnecessary and are an overkill for assignments, they have been shown to save considerable time and effort in both academic and professional settings. Humans are prone to quickly forget details and good function-level documentation provides continuity for developers by acting as a repository for information related to the function. Otherwise, the developer will have to unnecessarily invest time in recalling and remembering undocumented gotcha details and assumptions each time the function is debugged or extended to incorporate additional features. Here is a sample for function `substitute_char`:

```
/*!*****************************************************************************
\brief
   Replaces each instance of a given character in a string with
   other given characters.

\param string
  The string to walk through and replace characters in.

\param old_char
  The original character that will be replaced in the string.

\param new_char
  The character used to replace the old characters
```

```
\return
  The number of characters changed in the string.
*******************************************************************************/
```

Submission and automatic evaluation

1. In the course web page, click on the appropriate submission page to submit `functions.cpp` and `functions.h`.
2. Please read the following rubrics to maximize your grade. Your submission will receive:
    - `F` grade if your `functions.cpp` doesn't compile with the full suite of `g++` options.
    - `F` grade if your `functions.cpp` doesn't link to create an executable.
    - Your implementation's output doesn't match correct output of the grader (you can see the inputs and outputs of the auto grader's tests). The auto grader will provide a proportional grade based on how many incorrect results were generated by your submission. `A+` grade if output of function matches correct output of auto grader.
    - A deduction of one letter grade for each missing documentation block in `functions.cpp` and `functions.h`. Your submission `functions.*` (if it was edited by you) must have **one** file-level documentation block and **one** function-level documentation blocks. A teaching assistant will physically read submitted source files to ensure that these documentation blocks are authored correctly. Each missing or incomplete or copy-pasted (with irrelevant information from some previous assessment) block will result in a deduction of a letter grade. For example, if the automatic grader gave your submission an `A+` grade and one documentation block is missing, your grade will be later reduced from `A+` to `B+`. Another example: if the automatic grade gave your submission a `C` grade and the two documentation blocks are missing, your grade will be later reduced from `C` to `E`. Likewise, your submission `functions.h` must have **one** file-level documentation block and **one** function-level documentation block.