NAME

Class TaggedValues − Constructs a map of parameter name and value pairs from an ASCII input file and provides an interface to query for the value of a given parameter.

SYNOPSIS

#include <TaggedValues.h>

TaggedValues(const std::string *inputFile***);**
TaggedValues(const std::vector<std::string> *virtualInputFile***);**
int getParamValue(std::string *tag***, int** *defaultValue***);**
double getParamValue(std::string *tag***, double** *defaultValue***);**
std::vector<std::string> getVirtualInputFile();
void printParamValues();
void printVirtualInputFile();

DESCRIPTION

Allows the user to construct an object built from an input file that lists parameter names and their corresponding values. Rather than opening and looking to the input file each time the user would like to get a parameter value, this object will open the file once, read each line of the file, and insert the parameters and values into a map so that efficient querying can be carried out.

There are two constructors for the class, each taking one parameter. The first constructs the class object by providing the path of an input file. The second constructs the class object from a vector of strings where each vector indexed string represents a line of an input file. This second constructor is provided for versatility in the storage of an input file in case the user would like to embed it with other information in a single file. As long as the user can extract and create a vector with the input file lines, the user will be able to construct a TaggedValues object and query the data.

The **getParamValue()** method takes two parameters, the name of the parameter to look for and a default value, which will be returned if the provided parameter name is not found in the map. There are two polymorphically equivalent **getParamValue()** methods allowing for the caller to specify a defaultValue type of int or double. The defaultValue type used by the caller will determine the type returned by the method. E.g. if the caller specifies a defaultValue of 1.0 then a value found in the map will be returned as a double, otherwise 1.0 would be returned. Similarly, if the default value specified was 1 then the returned value would be an integer. It is assumed that the user will know what type each returned parameter's value is going to be and will provide a defaultValue with the appropriate type. Otherwise some undesired results may be experienced.

The **getVirtualInputFile()** method returns a vector of strings that contains the contents of the input file (or vector) initially given to the constructor when creating the class object.

The **printParamValues()** method prints to standard output all of the parameter name and value pairs contained in the class's map.

The **printVirtualInputFile()** method prints to standard output the input file provided when the class object was constructed.

INPUT FILE FORMAT

Each parameter and value must be separated by a single '=' or ':' character.

Each parameter and value must be on its own line (or equivalently its own indexed string in a vector<string>).

Whitespace is removed from each line in an input file as well as from all parameter queries to try and achieve a uniform format. As far as the parser is concerned, the line "p a ra m3 = 4" is equivalent to "param3=4". Finding the parameter name and value involves iterating throught the string until the separator character is reached. The characters on the left hand side are considered the key and those on the right hand side are the value. These two strings are then entered as a pair into the class's map.

Comment usage: Users may use the '#' character to denote a comment. The parsing of the parameter-value pairs will effectively ignore any '#' and characters that follow it on the line.

```
#This entire line will be ignored.
p a ram( 4) = 9.0 #This is a partial line comment.
```

The process for removing whitespace also handles comment removal. The example above is functionally equivalent to the following:

```
param(4)=9.0
```

The commenting used here is similar to many programming languages and is not a difficult concept. Adding comments and notes to an input file can greatly increase the readability. The following is an example of a correctly formatted input file.

```
# Number of Classes
classes = 2

# Number of Machines
servers = 2

# Machine to which each class belongs
sigma(1) = 1
sigma(2) = 2

# Arrival rates into each class
lambda = 1.0

# Holding costs
c(1) = 1.0
c(2) = 7.0

# Service rates of each class
mu(1) = 1.2
mu(2) = 1.4

# Destination Class (0 to exit system)
s(1) = 2
s(2) = 0

epsilon = .00001
iterMax = 10000
N = 40
```

AUTHOR

The idea and methods for this class came from code written by Nathan Walker. The class documented here was written by Taylor Carr, who also wrote the manual page.