```
/*
External documentation and recommendations on the use of this code is
available at http://www.cs.umass.edu/~rich/tiles.html.

This is an implementation of grid-style tile codings, based originally on
the UNH CMAC code (see http://www.ece.unh.edu/robots/cmac.htm).
Here we provide a procedure, "GetTiles", that maps real
variables to a list of tiles. This function is memoryless and requires no
setup. We assume that hashing colisions are to be ignored. There may be
duplicates in the list of tiles, but this is unlikely if memory-size is
large.

The input variables will be gridded at unit intervals, so generalization
will be by 1 in each direction, and any scaling will have
to be done externally before calling tiles.

It is recommended by the UNH folks that num-tiles be a power of 2, e.g., 16.
*/

#include <iostream>
#include "tiles.h"
#include "stdlib.h"
#include "math.h"

int hash_coordinates(int *coordinates, int num_indices, int memory_size);

void GetTiles(
        int tiles[],               // provided array contains returned tiles (tile
indices)
        int num_tilings,           // number of tile indices to be returned in tiles
        double variables[],         // array of variables
    int num_variables,         // number of variables
    int memory_size,           // total number of possible tiles (memory size)
    int hash1,                 // change these from -1 to get a different hashing
    int hash2,
    int hash3)
{
        int i,j;
        int qstate[MAX_NUM_VARS];
        int base[MAX_NUM_VARS];
        int coordinates[MAX_NUM_VARS + 4];   /* one interval number per rel dimension */
        int num_coordinates;

        if (hash1 == -1)
                num_coordinates = num_variables + 1;       // no additional hashing
corrdinates
        else if (hash2 == -1) {
                num_coordinates = num_variables + 2;       // one additional hashing
coordinates
                coordinates[num_variables+1] = hash1;
        }
        else if (hash3 == -1) {
                num_coordinates = num_variables + 3;       // two additional hashing
coordinates
                coordinates[num_variables+1] = hash1;
                coordinates[num_variables+2] = hash2;
    }
    else {
                num_coordinates = num_variables + 4;       // three additional hashing
```

```
coordinates
                coordinates[num_variables+1] = hash1;
                coordinates[num_variables+2] = hash2;
                coordinates[num_variables+3] = hash3;
    }

        /* quantize state to integers (henceforth, tile widths == num_tilings) */
    for (i = 0; i < num_variables; i++) {
        qstate[i] = (int) floor(variables[i] * num_tilings);
        base[i] = 0;
    }

    /*compute the tile numbers */
    for (j = 0; j < num_tilings; j++) {

                /* loop over each relevant dimension */
                for (i = 0; i < num_variables; i++) {

                /* find coordinates of activated tile in tiling space */
                        if (qstate[i] >= base[i])
                                coordinates[i] = qstate[i] - ((qstate[i] - base[i]) %
num_tilings);
                        else
                                coordinates[i] = qstate[i]+1 + ((base[i] - qstate[i] - 1)
% num_tilings) - num_tilings;

                        /* compute displacement of next tiling in quantized space */
                        base[i] += 1 + (2 * i);
                }
                /* add additional indices for tiling and hashing_set so they hash
differently */
                coordinates[i++] = j;

                tiles[j] = hash_coordinates(coordinates, num_coordinates, memory_size);
        }
        return;
}


/* hash_coordinates
   Takes an array of integer coordinates and returns the corresponding tile after hashing
*/
int hash_coordinates(int *coordinates, int num_indices, int memory_size)
{
        static int first_call = 1;
        static unsigned int rndseq[2048];
        int i,k;
        long index;
        long sum = 0;

        /* if first call to hashing, initialize table of random numbers */
    if (first_call) {
                for (k = 0; k < 2048; k++) {
                        rndseq[k] = 0;
                        for (i=0; i < sizeof(int); ++i)
                        rndseq[k] = (rndseq[k] << 8) | (rand() & 0xff);
                }
        first_call = 0;
    }
```

```
        for (i = 0; i < num_indices; i++) {
                /* add random table offset for this dimension and wrap around */
                index = coordinates[i];
                index += (449 * i);
                index %= 2048;
                while (index < 0) index += 2048;

                /* add selected random number to sum */
                sum += (long)rndseq[(int)index];
        }
        index = (int)(sum % memory_size);
        while (index < 0) index += memory_size;

        return(index);
}
```