NAME

Class MultiIndex − Allows an arbitrary number of nested *for*-loops to be implemented with a single loop.

SYNOPSIS

#include <MultiIndex.h>

MultiIndex(int *dim***, int*** *start***, int*** *end***, int*** *stride***);**
MultiIndex(int *dim***, int*** *start***, int*** *end***);**
MultiIndex(int *dim***, int*** *end***);**
MultiIndex(int *dim***, int** *start***, int** *end***, int** *stride***);**
MultiIndex(int *dim***, int** *start***, int** *end***);**
MultiIndex(int *dim***, int** *end***);**
MultiIndex(const MultiIndex& *x***);**
void resetBounds();
void resetForNext();
void resetForNext(int *leastSignificantIndex***);**
int start();
int start(int *leastSignificantIndex***);**
int next();
int next(int *leastSignificantIndex***);**
int prev();
int prev(int *leastSignificantIndex***);**
bool inRange();
int getOffset();
int getOffset(int *leastSignificantIndex***);**
int getOffset(int* *index***);**
int getDelta(int* *del***);**
int* getCounterArray();
int* getCounterArray(int *bias***);**
int* getCounterStart();
int* getCounterStart(int *bias***);**
int* getCounterEnd();
int* getCounterEnd(int *bias***);**
int* getCounterStride();
int* getCounterStride(int *bias***);**
int getCounterValue(int *i***);**
int getCounterDim();
int getCounterExtent();
std::string toString();
std::string toString(std::string *sep***);**

DESCRIPTION

Provides an easy interface use a single *for*-loop to do the work of arbitrarily deeply nested *for*-loops. Often such nested loop constructs are used to index a multidimensional array. This class allows a single contiguous block of memory (i.e. a single dimensional array) to be treated as if it is a multidimensional array.

There are several different forms of the constructor. Each takes the argument *dim* which is the size of the counter array (i.e. the number of loops being represented). The arguments *start*, *end*, and *stride* can either be integers or pointers to arrays of integers (each containing *dim* entries) that represent the starting, ending, and stride values for each of the counter values. If *stride* is not supplied then it is assumed to be 1. If *start* is not supplied then it is assumed to be 0. Please note that, unlike the usual convention in C++ looping constructs, the counter values are increased up to *end* **inclusive.**

Perhaps the most common use of the **MultiIndex** class is to iterate over a multidimensional array whose dimensions are not known until runtime. This is handled using a single dimensional array to hold the multidimensional data and mapping the multidimensional counter location to a single offset from the start of the array. The offset value can be accessed through the **getOffset**() method. Without an argument, this

method returns the offset is to the location corresponding to the current counter location. If the argument is an integer array then the offset is to the location specified by the array. A single nonnegative integer argument specifies the least significant counter value to use when computing the offset. It is important to note that offsets are relative to the zeroth element of an array, even if the starting values supplied to the constructor are not zero.

The **start**() method resets all counter values from the most significant (leftmost) to the least significant (rightmost) to their starting values and returns the corresponding offset. The method **next**() increments the counter to its next value and returns the new offset. The optional integer argument indicates the least significant counter value to increment; if not supplied it is assumed to be zero, indicating the rightmost counter value. The increment is done by adding the stride to the specified location. If this results in a value that is larger than the corresponding ending value then value is reset to the starting value and a "carry" is performed to the next counter value on the left. The **prev**() method works like **next**() but decrements the counter rather that incrementing it.

As an alternative to **start**(), one can initialize (or reset) a counter using the **resetForNext**() method. This initializes the counter values so that the first call to **next**() will increment the counter to its initial location and return the corresponding offset.

The boolean method **inRange**() returns *true* if the current counter values are all between (inclusive) the starting and ending values supplied to the constructor, otherwise *false* is returned. The **resetBounds**() method can be used to change the starting, ending and stride values for each counter value.

Often one needs to compute the offset from the current counter location to a location specified by a *delta*, an array of positive, zero, or negative values that are added to the current counter values. This is done by the **getDelta**() method.

Access to the counter values as an array is provided by the **getCounterArray**() method which returns a pointer to the counter array. The optional argument *bias* can be used to "shift" the pointer so that the array can be indexed from a number other than zero. For example, if *bias* = 1 and the return value of the **getCounterArray**() method is *x*, then *x*[1] will contain the most significant (leftmost) counter value while accessing *x*[0] should be considered an error. The methods **getCounterStart**(), **getCounterEnd**(), and **getCounterStride**() provide similar access to the current start, end, and stride values. **Warning:** Since these methods return pointers to the internal counter arrays, changing values in these arrays may have unintended consequences.

The method **getCounterDim**() returns the number of counters, while **getCounterExtent**() returns the total number of values the counter will iterate over. The method **toString**() returns a space-delimited string representation of the counter; the delimiter can be changed by supplying the desired delimiter as an argument to the method.

EXAMPLES

Consider the triply nested *for*-loop

```
for (int b0 = 0; b0 < 2; b0++)
{
    for (int b1 = 0; b1 < 2; b1++)
    {
        for (int b2 = 0; b2 < 2; b2++)
        {
            cout << b0 << b1 << b2 << endl;
        }
    }
}
```

which counts in binary from 000 to 111. This can be implemented using a **MultiIndex** loop as

```
MultiIndex xcnt(3, 0, 1);
int* x = xcnt.getCounterArray();
```

```
for (xcnt.start(); xcnt.inRange(); xcnt.next())
{
    cout << b[0] << b[1] << b[2] << endl;
}
```

Notice that all three counter values start at 0 and are increased up to 1 inclusive.

In some cases it may be desirable to have the counter array indexed from some number other than 0. This is frequently done in mathematical code that works with counters subscripted from 1. The loop

```
MultiIndex xcnt(3, 0, 1);
int* x = xcnt.getCounterArray(1);
for (xcnt.start(); xcnt.inRange(); xcnt.next())
{
    cout << b[1] << b[2] << b[3] << endl;
}
```

will produce the same output as the previous loop but uses indices starting with 1.

For a more general example, consider the quadruply nested *for*-loop with different bounds for each loop that accesses a portion of the a four-dimensional array:

```
double a[10][10][5][5];
int x[4];
for (x[0] = 0; x[0] <= 10; x[0] += 1)
{
    for (x[1] = 2; x[1] <= 10; x[1] += 2)
    {
        for (x[2] = 0; x[2] <= 1; x[2] += 1)
        {
            for (x[3] = 1; x[3] <= 1; x[3] += 1)
            {
                // statements accessing x[0], x[1], x[2], and x[3]
                // or accessing a[x[0]][x[1]][x[2]][x[3]]
            }
        }
    }
}
```

This can easily be implemented as a **MultiIndex** loop with

```
double a[10][10][5][5];
double* A = &a[0][0][0][0];
int min[4]    = { 0,  2, 0 ,1};
int max[4]    = {10, 10, 1, 1};
int stride[4] = { 1,  2, 1, 1};
MultiIndex xcnt(4, min, max, stride);
int* x = xcnt.getCounterArray();
for (int X = xcnt.start(); xcnt.inRange(); X = xcnt.next())
{
    // statements accessing x[0], x[1], x[2], and x[3]
    // or accessing A[X]
}
```

To improve efficiency it helpful to reduce the number of calls to **next**() by spitting the single loop into two loops; an outer loop over all but rightmost dimension and an inner loop over the rightmost dimension. This

requires a little more work but can greatly improve performance while still allowing general code when at two or more dimensions are needed.

```
double a[10][10][10][20];
double* A = &a[0][0][0][0];
int min[4]    = { 0,  0,  0,  0};
int max[4]    = {10, 10, 10, 20};
int stride[4] = { 1,  1,  1,  1};
MultiIndex xcnt(4, min, max, stride);
int* x = xcnt.getCounterArray();
int* str = xcnt.getCounterStart();// same as min[]
int* end = xcnt.getCounterEnd();  // same as max[]
int len = end[3] - start[3]; // could also use max[3]-min[3] or 20
for (int X0 = xcnt.start(); xcnt.inRange(); X0 = xcnt.next(1))
{
    for ( int X = X0; X <= X0 + len; X++ )
    {
        x[3] = X - X0; // set rightmost counter value
        // statements accessing x[0], x[1], x[2], and x[3]
        // or accessing A[X]
    }
}
```

AUTHOR

The idea for this class came from code written by Nathan Walker in 2007. The initial C++ class based on this idea was written by Christopher Pfohl in 2008 The class described here was implemented by Jonathan Senning in 2009 and this manual page was adapted from one written by Christopher Pfohl.