Range Checking in C++

Until now, Inforum C++ tools have performed only a limited degree of range checking of Vectors and Matrices. This likely is true not only of Bump but also of Bump-based software, including Interdyme, and similar tools. Full range checking has been performed with the Vector classes and with the () operator in the Matrix class. Frequently, however, daring programmers use the [] operator to read and write matrix elements. Unfortunately, the current [] operator range checks the row index but not the column index. It has seemed to be impossible to check the column index within the Matrix class, and indeed that likely remains true. However, full range checking can be achieved very easily, with little change in code, with nearly no additional memory requirements, and with little sacrifice in speed.

A small "Row" class is required and is specified in the following header:

```
class Matrix;
class Row {
   public:
   float* r;
   int fc, lc;
   inline float& operator [] (const int j) const;
   inline Row(float* i, int mfc, int mlc){
              r = i; fc = mfc; lc = mlc; }
   inline Row(Row& rw) { r = rw.r; fc=rw.fc; lc=rw.lc; }
   };
```

The Row class contains a pointer "r" and integers for the first and last column positions. In addition, there is a [] operator for reading elements of the row, an inline default constructor, and an inline copy constructor. Both constructors take a pointer and numbers of the first and last position. They set the corresponding Row class parameters to these values. Because they are inline functions, the time penalty for calling them should be small.

Only one function remains to be specified in the Row class implementation: the [] operator.

```
const char Rowerr1[] = "Error: Reference to a deleted Row.\n";
const char Rowerr2[] = "Illegal column index: ";

inline float& Row::operator [] (const int j) const{
   if(r==0){
      cerr << Rowerr1;
      exit(1);
      }
   if (fc > j || lc < j){
      cerr << Rowerr2 << j << "\n";
      exit(1);
      }
```

```
      return(r[j]);
      }
```

This function checks whether the index is within bounds. The function then returns a reference to element *j* of the row so that the element satisfies requirements for both "rvalues" and "lvalues." So far, this has been very simple, but what about modifying Bump to use the Row class? Fortunately, that proves just as simple (assuming that further testing validates these methods). First, make the following change to the Matrix declaration.

```
   //float* operator [](const int i) const;
   Row operator [](const int i) const;
```

The Matrix [] operator now returns a Row object instead of a pointer. Recall that the Row object contains a pointer; before the new function returns, a Row object is created and the pointer is set equal to the return value of the original version.

Several lines have been changed in the implementation of the Matrix [] operator. First, the return type is changed to Row. After the function range checks the row index, it calls the Row constructor with a pointer to the i'th row and integer values of the first column and last columns. The function then returns the Row object.

```
//float * Matrix::operator [](const int i) const
Row Matrix::operator [](const int i) const{
   if(m==0){
      cerr << "Error: Reference to a deleted matrix.\n";
      tap();
      exit(1);
   if(i < fr || i > lr ){
      cerr << "illegal matrix row index: " << i << "." << "\n";
      cerr << "returning first row. \n";
      //return m[fr];
      Row row(m[fr],fc,lc);
      return row;
      }
   //return(m[i]);
   Row row(m[i],fc,lc);
   return row;
   }
```

What does this code do? Let us begin with the operation of a Matrix member function (e.g. the Matrix * operator). These functions can access the private pointer "m**" and hence never call the [] operator. Instead, they access the position in memory directly (m[i][j]). Thus, there is no reduction in speed for these functions. This also is important for functions like the Matrix destructor, which depends on "m[i]"; this does NOT call the [] operator.

The change comes when non-member functions and users' code access elements of a matrix. Suppose the user declared a Matrix as "Matrix<int> matty(2,2);" and set the value in element (1,2) with the statement "matty[1][2] = 1;". When this line is executed,

the Matrix [] operator is called first. It verifies that row 1 is in bounds. It then calls the Row constructor with a pointer to row 1 of the matrix and the first and last column positions (1 and 2). These values are copied to the Row parameters. The Matrix [] operator then returns the Row object. In the process of returning, first the Row copy constructor is called, and then the Row [] operator is called for the (copy of the) Row object that is being returned. The Row [] operator first determines whether the column index is within bounds and then returns the reference to the *j*'th element (of row *i*). It is important to return the reference so that "matty[i][j]" can be an "lvalue."

Assuming that this modification is bug-free and complete, no other changes are required. Matrix elements are stored in the same way and accessed in the same way by member functions. Row objects are created only by the Matrix [] operator and are destroyed immediately after use; their only purpose is to allow full range checking. Their small size and use of inline constructors make them extremely efficient.

The results of a rather informal experiment showed that inclusion of the Row class yielded a time penalty of less than one second when IdLift was run through 2010 on a P4 machine. The value of full range checking was established immediately when I tested it on my own programs; in several cases, the program was reading or writing to elements outside of the column ranges, but the program previously gave no obvious indication of trouble. IdLift also was tested; perhaps surprisingly, no errors were found. Keep in mind, though, that Interdyme continues to use Numerical Recipes vectors and matrices for many operations. These are not range checked.

Please let me know of your thoughts and experiences with this code.

Ron Horst
Inforum
horst@econ.bsos.umd.edu
horst@inforum.umd.edu