EFFECTIVE WAY OF OVERRIDING C++ OPERATORS FOR MATRIX OPERATIONS

Jan CVEJN
Technical University of Liberec, Faculty of Mechatronics and Interdisciplinary Engineering Studies
Hálkova 6, 461 17 Liberec, Czech Republic, E-mail: jan.cvejn@vslib.cz

SUMMARY

The article describes an effective way how to create a library for manipulating matrices in a similar manner as if they were numbers in programming language C++. Although overriding arithmetic operators for manipulating classes such as vectors and matrices seems to be a standard procedure in C++, there occur serious problems when performance aspects are taken into account. The main disadvantage consists in frequent allocating and deallocating memory, which can significantly slow down evaluating arithmetic expressions. In the article an implementation is described that eliminates redundant memory allocations and enables using overridden operators for effective evaluating matrix expressions.

Keywords: *C++ programming, overloading operators, object-oriented programming*

1. INTRODUCTION

Creating classes to simplify operations with dynamic structures is one of key concepts of object-oriented programming in general and implementing such a class is a common procedure in any object–oriented language. In C++, it is however possible to go further and extend applicability of arithmetic operators to manipulate with structured objects. This would be especially useful at work with objects that naturally participate in arithmetic expressions, such as vectors and matrices. For example, let us consider the following matrix expression:

$$C = BAB^{-1} + D^T E$$

A general approach to creating a class to simplify programming such expressions in an object-oriented language would be defining methods for matrix addition, multiplication, transposition and inversion. The expression above would be then transcribed as follows:

```
Z.inv(B);
Y.mul(B,A);
X.mul(Y,Z);
W.tr(D);
V.mul(W,E);
C.add(X,V);
```

In this example it is needed to define help matrix variables Z, Y, X, W, V. However, which is worse, from the lines above it is not very obvious what they actually do. In C++ we can, under some conditions, which are discussed later, override operators to enable writing the expression in natural form as follows:

```
C = B*A*B.inv()+D.tr()*E;
```

In this way matrix expressions can be computed in a similar manner as in some high-level matrix language while keeping high performance and full control of evaluating. Since such expressions are usually parts of complex numerical algorithms, effectivity is very important.

2. BASIC IMPLEMENTATION

To start, let us define a simple C++ class for matrix operations and describe briefly some elementary steps. The memory for floating-point data is allocated dynamically.

```
#define NUM double

class Matrix
{
   NUM *pdata;     //buffer for data
   int dataLen;    //length of
                    //allocated buffer
   int m,n;       //dimensions of matrix

public:
   Matrix()
   {
      pdata=NULL;
      dataLen=m=n=0;
   }

   Matrix(int m,int m)
   {
      pdata=NULL;
      dataLen=m=n=0;
      redim(m,n);
   }

   ~Matrix() { delete[] pdata; }

   void copy(const Matrix &A);
   void redim(int _m, int _n); //method
      //for (re)allocating memory

   Matrix(const Matrix &A)  { copy(A);}

   Matrix &operator=(const Matrix &A)
   {
      delete[] pdata;
      copy(A); return *this; }
```

```
   void setValue(NUM value);

   void checkBounds(int i, int j)
   { assert(0<=i && i<m
      && 0<=j && j<n); }

   NUM &operator()(int i, int j)
   { checkBounds(i,j);
      return pdata[i*n+j]; }

   NUM getAt(int i, int j)
   { checkBounds(i,j);
      return pdata[i*n+j]; }

   void setAt(int i, int j, NUM v)
   { checkBounds(i,j); pdata[i*n+j]=v;}
};
```

Using this class we can create a matrix variable A of dimensions m x n as follows:

```
Matrix A(m,n);
```

To access a particular element of the matrix, the overridden operator () can be used:

```
A(0,0)=1;
```

or

```
A.setAt(0,0,1);
```

Methods for accessing elements use standard debugging macro assert() to check for correct values of indexes. This macro does not affect performance since it is automatically omitted in the release build of the program. All these methods are written as inline for eliminating overhead of calling functions.

Since the copy constructor and operator = are overridden, we can write

```
Matrix B(A),C;
...
C=B;
```

Note that operator= has to deallocate data unlike the copy constructor.

There remain several things to write – method redim() used for allocating and reallocating the memory, method setValue() used for initializing data and method copy() for copying the data.

```
void Matrix::setValue(NUM value)
{
   for(int i=0;i<m*n;i++)
      pdata[i]=value;
}

void Matrix::redim(int _m, int _n)
{
   if(m!=_m || n!=_n)
   {
      if(_m*_n>dataLen)
      {
         dataLen=_m*_n;
         delete[] pdata;
         pdata=new NUM[dataLen];
      }
      m=_m;
      n=_n;
   }
}

void Matrix::copy(const Matrix &A)
{
       redim(A.m, A.n);
       memcpy(pdata,A.pdata,
         m*n*sizeof(NUM));
}
```

Note that dataLen is greater or equal to m.n. The buffer is reallocated only if it is necessary, i.e. if it would be m.n > dataLen. This is one of the most important features of this class implementation. If any global or static object of class Matrix is used for storing results, its data buffer is reallocated only once, or possibly a few times if the variable is used at more places of the program for storing matrices with different dimensions.

Internal implementation of Matrix could be different. For example, it would be possible to define it as a dynamic array of pointers to dynamic arrays that are allocated separately. In this case, the access to an element may be on some machines little faster, since there is no need of multiplication:

```
class Matrix
{
public:
   NUM **pdata;
   int m, n;       //dimensions of
                    //the matrix
   ...

   NUM& operator()(int i,int j)
        //obtain ref. to A(i,j) elem.
   {
      return pdata[i][j];
   }
};
```

However, in this case, reallocating the data buffers is more complicated. Another advantage of the first implementation is the possibility of iterating through data in a single loop, which is faster (see implementation of methods setValue() and copy() above).

3. OVERRIDING OPERATORS

Now, some operators for manipulating matrices can be defined. At first, two simple and commonly used ways are described with their disadvantages.

The operators +, += can be defined as follows:

```
class Matrix
{
   ...

   void operator+=(const Matrix &A);
   Matrix operator+(const Matrix &A);
};
```

```
void Matrix::operator+=(const Matrix
&A)
{
   assert(m==A.m && n==A.n);
       //+ defined only for matrices of
       //the same dimensions

   for(int i=0;i<m*n;i++)
      pdata[i]+=A.pdata[i];
}

Matrix Matrix::operator+(const Matrix
&A)
{
   assert(m==A.m && n==A.n);

   Matrix B(m,n);

   for(int i=0;i<m*n;i++)
      B.pdata[i]=pdata[i]+A.pdata[i];
   return B;
}
```

Operator += can be overridden in this way without any problems. Operator + implementation requires some discussion. The previous simple way to define operator + is correct, but unfortunately ineffective. Each time it is called, it has to allocate memory for the local object B that it returns. However, when B is returned, a new temporary object is created to hand over the data. This needs one more memory allocation. Finally, assignment operator = used in expression C=A+B may require one more call of redim(). Here the reallocation is not done if the matrix C has already sufficiently large data buffer. The right-hand arguments of the operators are handed over using references to save more memory allocations. Nevertheless, the operating system is called two times at minimum to allocate memory at each time the + operator is invoked. This allocation can be much more demanding than pure copying and summing the data.

Let us look at a different approach to defining operators. To omit unnecessary copying of data, arithmetic operators are often defined to return the reference to a local static object:

```
class Matrix
{
   ...

   Matrix &operator+(const Matrix &A);
   Matrix &operator*(const Matrix &A);
};

Matrix &Matrix::operator+(const Matrix
&A)
{
   assert(m==A.m && n==A.n);

   static Matrix B;    //!
   B.redim(m,n);

   for(int i=0;i<m*n;i++)
      B.pdata[i]=pdata[i]+A.pdata[i];
   return B;
}
```

```
Matrix &Matrix::operator*(const Matrix
&A)
{
   assert(n==A.m);

   static Matrix B;
   B.redim(m,A.n);

   for(int i=0;i<m;i++)
   {
      for(int j=0;j<A.n;j++)
      {
         NUM sum=0;
         for(int k=0;k<n;k++)
         sum+=getAt(i,k)*A.getAt(k,j);

         B.setAt(i,j,sum);
      }
   }
   return B;
}
```

In this case, the expression A+B will return reference to the static variable, defined in the operator + body, which holds the result. Expressions C=A+B and D=A+B+C will be evaluated correctly. However, let us look at an expression like this:

```
E=A*B+C*D;
```

C++ language keeps priorities of operators even if they are overridden for user classes. In the case of expression above, products A*B and C*D have to be held in two separate places before summation. However, there is only one help static variable in operator * to hold the temporary result. Therefore, the result cannot be correct.

Obviously, both the approaches are not suitable – either ineffective, or incorrect in some cases.

Note that an operator function has to return either Matrix or reference to Matrix so that more complicated expressions can be created.

4. EFFECTIVE IMPLEMENTATION OF OPERATORS

One approach to correct and yet effective implementation is described further. The main ideas are the following:

1. There is needed a static array of objects to hold the temporary results that occur while evaluating complex expressions. It can be called “array of results” ant its items “result objects”. The array has to be static because in the opposite case it would be allocated and deallocated each time any expression is evaluated.

2. Each time a result object is needed, the array will be searched for the first free result object. This means, each item of the array has to have a flag signalizing that it is free or not.

3. After evaluating the expression the flags of all the result objects have to be set to the free state so

that the objects can be used again for evaluating next expressions.

Since the result objects can be repeatedly used for matrices of different dimensions, before they are used, redim() method has to be called. Here one of the key ideas of this implementation is used - method redim() allocates memory only if the data buffer needs to grow. In consequence, after a small number of evaluations of expressions no more memory reallocation will be done.

In the beginning, the objects in the array of results have not allocated memory. The memory will be allocated the first time the object is used (calling method redim()). Since the array is static, memory will be automatically destroyed when the program ends.

The previous paragraphs imply that after many different expressions were evaluated, the number of allocated result objects will correspond to the number of operators in the most complex expression and the size of the largest memory block held by the array of results is the same as the size of the largest matrix that occurred in any expression.

Each operator could now ask the Matrix class for a free result object, perform the operation and return the reference to the result object. In this case, it is not yet clear how to realize step **3**. The result objects will not be informed in any way that the whole expression was evaluated and cannot reset their free flags. The solution of the outlined problem is as follows.

The operators will return temporary objects by value (not by reference). The temporary object will reference to a result object that will hold the result of the operation. As was explained after the first implementation of operator +, the returned object is two times copied before it is assigned to the destination matrix. It is not possible to avoid copying the objects, but it is not necessary to copy the dynamically allocated data. However, for normal operations the copy-constructor still has to allocate memory and make a "deep" copy. Destructor in the case of temporary objects must not deallocate the data, since they only reference to the result objects, but when it is called, it can set the flag of the referenced result object to the free state – it signalizes that the temporary object will be destroyed and the result object gets free for a next use.

Since, in a C++ program, all the temporary objects that were created during evaluating an expression are destroyed immediately after the whole result was obtained, point **3** is satisfied. This behavior was practically verified using Microsoft and GNU C++ compilers.

Instances of the Matrix class will work in two modes of operation:

A. Normal mode
In this mode the copy constructor allocates memory using new operator, destructor deallocates memory.

B. Temporary mode
In this mode the copy constructor copies pointer to the static result object and sets its flag to non-free state, destructor sets the flag of the result object to the free state, but does not deallocate its memory. Temporary objects are intended only for internal purposes of methods and operators of class Matrix.

There still remains one problem. When a temporary object is returned, it is copied to another temporary object. At this time, both the objects reference to the same data in the result array and also the same flag of the free state. Then, the first object is destroyed. However, it also sets the flag to the free state, in spite of the fact that the result object should still hold the data for the second temporary object. That is why some more sophisticated mechanism is needed to lock and unlock result-holding objects. Our solution uses a counter of references (refCnt). If the counter is zero, the result object is considered as free. The copy-constructor on the temporary object increases the counter, while destructor decreases it.

Figures 1 and 2 illustrate the two modes of operation.

From the previous explanation the solution may look to be rather complicated. Actually, only several modifications are made to the class Matrix implementation described above. Note: "n.c." in the program comments below means "no changes".

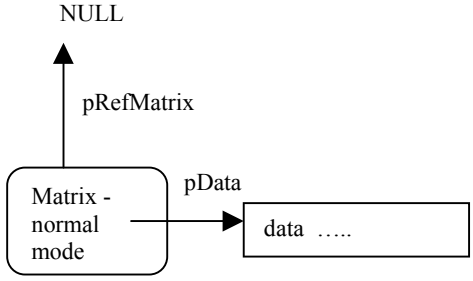

Fig. 1 Normal mode

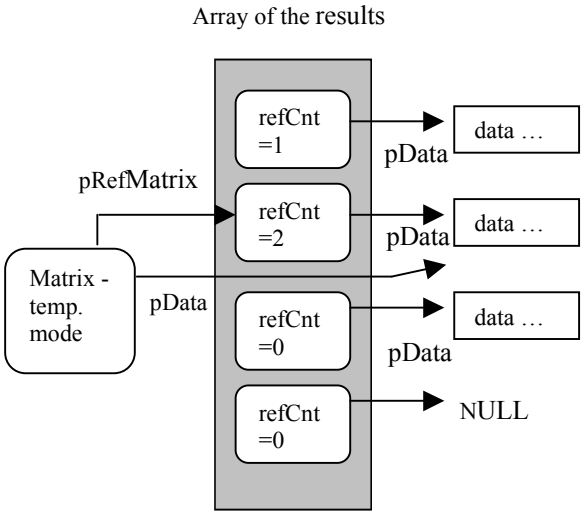

Fig. 2 Temporary object and the array of results

```
#define RESULT_MAX 1000

class Matrix
{
   int refCnt;  //counter of references
   Matrix *pRefMatrix;
       //NULL for normal objects,
       //non-zero for temporary ones

   NUM *pdata;  //buffer for data
   int dataLen;  //length of allocated
                 //buffer
   int m,n;    //dimensions

   static Matrix
   resultArray[RESULT_MAX];   // array
                            //of results
   static Matrix *pArrayTop;
              // pointer on
              //the first free element
public:
   Matrix()
   {
      refCnt=0;
      pRefMatrix=NULL;
      pdata=NULL;
      dataLen=m=n=0;
   }

   Matrix(int m, int m);
   Matrix(const Matrix &A);

   ~Matrix();

   void copy(const Matrix &A); //n.c.
   void redim(int _m, int _n); //n.c.
   Matrix &operator=(const Matrix &A);
                                  //n.c.
   void setValue(NUM value); //n.c.

    NUM &operator()(int i, int j);//n.c.
   NUM getAt(int i, int j); // n.c.
   void setAt(int i, int j, NUM v);
                             //n.c.
   void setTempObj(int _m,int _n);
      // creates a new temporary object

   void operator+=(const Matrix &A);
      // n.c.
   Matrix operator+(const Matrix &A);
   Matrix operator*(const Matrix &A);
};
```

Implementation of the methods that changed follows. New version of the copy-constructor behaves differently in the normal mode and in the temporary mode. The temporary mode is active if the pRefMatrix pointer is nonzero.

```
Matrix::Matrix(const Matrix &M)
//extended copy-constructor
{
   refCnt=0;
   pRefMatrix=M.pRefMatrix;

   if(pRefMatrix)
   //this object is temporary
   {
      pdata=pRefMatrix->pdata;
         // set reference to data
      m=pRefMatrix->m;
      n=pRefMatrix->n;

      ++pRefMatrix->refCnt;
         // lock the result object
   }
   else  // normal operation
      copy(M);
}
```

Destructor decreases the reference count if the object is in the temporary mode. Otherwise, it deallocates memory.

```
Matrix::~Matrix()
{
   if(pRefMatrix)
   //this object is temporary
      --pRefMatrix->refCnt;
   else
      delete[] pdata;
}
```

The setTempObj() method links a temporary object to a static result object of required dimensions. It searches the results array for the first free result object and sets references onto it. It would be possible to search always from the beginning of the array. Instead, rather more effective algorithm was used – is stores the position of the last used element of the result array and starts searching from it. In most cases, only one step is needed to find the next free element. However, the array will not behave like a stack because the elements can set reference counters to zero at any position and not only on the top. The setTempObj() method will be used instead of redim() for temporary objects.

```
void Matrix::setTempObj(int _m,int _n)
{
   while(pArrayTop->refCnt==0 &&
      pArrayTop>=resultArray)
      --pArrayTop;

   ++pArrayTop;

   assert(pArrayTop-
      resultArray<RESULT_MAX);
   assert(pArrayTop->refCnt==0);

   pRefMatrix=pArrayTop;
   ++pRefMatrix->refCnt;

   pRefMatrix->redim(_m,_n);

   pdata=pRefMatrix->pdata;
   m=pRefMatrix->m;
   n=pRefMatrix->n;
}
```

The static variable pArrayTop is initially set on the beginning of the array:

```
Matrix *Matrix::pArrayTop=resultArray;
```

Constant RESULT_MAX should be chosen to be sufficiently large (e.g. 1000). It would be also possible to implement resultArray as dynamic at the cost of some more complexity. However, as noted before, only limited number of elements will have allocated memory (number of allocated elements depends on how complicated expressions will be used in the program). The other unallocated elements use only a little amount of memory (about 24 bytes each). Therefore constant RESULT_MAX can be chosen to be sufficiently large and the array can have fixed size.

Finally, we can define operators + and * :

```
Matrix Matrix::operator+(const Matrix
&A)
{
   assert(m==A.m && n==A.n);

   Matrix B;
   B. setTempObj(m,n);
   //instead of redim() for temp.
   //objects

   for(int i=0;i<m*n;i++)
      B.pdata[i]=pdata[i]+A.pdata[i];

   return B;
}

Matrix Matrix::operator*(const Matrix
&A)
{
   assert(n==A.m);

   Matrix B;
   B. setTempObj(m,A.n);
      //instead of redim() for temp.
      //objects

   for(int i=0;i<m;i++)
      for(int j=0;j<A.n;j++)
      {
         NUM sum=0;
         for(int k=0;k<n;k++)
         sum+=getAt(i,k)*A.getAt(k,j);

         B.setAt(i,j,sum);
      }
   return B;
}
```

The other operators and methods would be defined in a similar manner. If they returns a matrix result, they first create a temporary object as a local variable on stack and then link it to a result object using setTempObj(). The operators will have the same priorities as if they were used with numbers. The order of evaluating can be changed using parentheses.

Although the overhead of frequent memory allocation was eliminated, there is still some more work done if compared to manipulating using methods. For example, if a method that sums two matrices and stores the result in the current object is used:

```
C.sum(A,B);
```

there is no need to copy the result data to the C object as if the operators + and = were used. This may be especially true for special kinds of expressions that can be much effectively evaluated using special methods than by operators. Compare:

```
C=A*p+B*q;    //p,q - numbers
```

and

```
C.linearCombine(A,B,p,q);
```

In general, for complicated expressions, one more copying of the result data is insignificant. But if operators are used, we get much better looking and sinoptical program.

Eventually, if very large matrices are worked with, using methods would be preferred to operators for the reasons of saving memory – items in the array of results will not be deallocated until the program ends. It is however also possible to free the memory held by the array on demand calling some method.

A matrix library based on the described principles was tested and successfully applied using Microsoft Visual C++ and GNU C++ compilers.

BIOGRAPHY

Jan Cvejn (Ing., Ph.D.) was born in 1972. In 1996 he graduated at the Technical University of Liberec (Czech Republic), where he still works as a tutor. His professional interest is focused on the problems of optimization, optimal control, algorithms and data structures. He is also author and co-author of several computer programs that were applied in industry.