USE OF A MATRIX CLASS TO INTRODUCE OBJECT ORIENTED PROGRAMMING

William H. Jermann, Ph. D.
Department of Electrical Engineering
The University of Memphis

ABSTRACT

At the end of a junior-level course called Matrix Computer Methods, students are introduced to object orientied programming through use of a user defined class called a Matrix class. The introductory example is nontrivial and illustrates differences in procedural techniques and object oriented programming.

INTRODUCTION

In our introductory programming course, students develop skills in procedural programming using standard C [l]. In their first assignment, and in all subsequent assignments, they develop and use separately compiled functions.

In a subsequent course called Matrix Computer Methods, they develop and use a collection of matrix functions. By the time they finish this course, they have accumulated considerable experience in using a procedural programming language.

Until recently, object oriented programming using C++ was first introduced in an upper division elective. However, we found that objects that are commonly used in C++ textbooks were not very satisfying as far as illustrating object oriented programming [2] [3]. Examples of such classes are arrays, complex numbers, stacks, screen-graphic classes, and general-type concepts such as "zoo-animal" classes.

We now introduce object-oriented programming at the end of the Matrix Computer Methods course using a Matrix class. The difference between object oriented techniques and procedural techniques is vividly illustrated.

In the following paper, the class definitions, class methods, and operation overloading functions associated with this example are discussed. Furthermore, the Matrix class is used to develop a hierarchy of derived classes that inherit attributes from the base class.

USING C++ FOR OBJECT ORIENTED PROGRAMMING

Students frequently use class libraries to interface applications programs to Windows environments. In fact, sometimes object oriented programming appears to be associated exclusively with the use of graphical objects. However, concepts of object oriented programming are much more general. The entire issue of BYTE in August 1981 relates to object oriented programming using smalltalk-80 [4].

Unlike smalltalk, C++ is not just an object oriented programming language, but rather a hybrid language. That is, the procedural techniques used in C may still be used in C++. But in addition, objects may be defined as instances of classes, and object oriented techniques may be employed.

In our Matrix Computer Methods course, students develop and separately compile a number of matrix functions. This set includes functions that put a number into a matrix, retrieve a specified element from a matrix, read and print matrices,perform addition and multiplication of matrices, and find inverses and determinants of square matrices. These functions are used in implementing Matrix class methods when object oriented programming is introduced.

C code may be recompiled and used in C++ programs. However, there are several differences between C++ and C. Some of these are:

1. Class definitions that lead to object oriented programming techniques.

2. Use of inline functions.

3. Function and operation overloading.

4. Default argument values.

5. Passing arguments by reference.

6. Use of the "new" and "delete" operators.

These differences are illustrated in the following example

USE OF A MATRIX CLASS

A program illustrating object oriented techniques usinq a Matrix class is shown in Figure 1. The Matrix class itself is defined in the header file, matrix5.h, and is shown in Figure 2. The class methods are shown in Figures 3 and 4. The code for these class functions employs procedural techniques and uses functions that have been developed by students earlier in the course.

Refer to the code shown in Figure 1. The difference between procedural techniques and object oriented techniques is quite clear. For example, to print matrix b, the statement b.print() is used. If procedural techniques were used, a function would have to be invoked, and arguments carried into the function, such as the number of rows in the matrix, the number of columns, a pointer to the matrix, and the column dimension of a 2-dimensional array.

Refer to the program statement in Figure 1,

```
a = b * e.inv() + d * f.inv()
```

Clearly the operations +, *, and = have been overloaded, so that in this context they imply matrix addition, matrix multiplication, and assignment of matrix values to a matrix object. To implement this statement using procedural techniques, an inverse function would have to be called twice, a multiplication function called twice, and a matrix addition function called once. Each function requires a set of arguments which must be transmitted in the proper sequence. This can be tedious since a general matrix multiplication function requires 9 arguments.

Refer to the read() function at the start of the program, and the read(p) function at the end of the program. The former reads from standard input, and the latter from a file. This is an example of either an overloaded function, or the use of a default argument being passed to a function.

The program in Figure 1 clearly illustrates the difference between procedural techniques and object oriented techniques. If procedural techniques are used, much effort is required in the use of language semantics, whereas in object oriented programming, the programmer can concentrate on the manipulations involving the objects.

Refer to Figure 2. A Matrix class is defined, and several of the methods are implemented using inline functions. The class definition can be extended significantly.

Class functions and operator overload functions are shown in Figures 3 and 4. Some of these involve use of functions that have been previously written by students in the matrix course. These include cget, cput, matread, matprint, matadd, matmul, and inv, and are declared in the header file csubs5.h. Thus, many of the class functions are implemented using procedural techniques and previously written code. In Figures 3 and 4, there are illustrations of transmitting arguments by reference as well as illustrations of use of the new and delete operators.

DERIVED CLASSES AND INHERITANCE

Refer to Figure 5. Clearly the set of all square matrices is a subset of the set of matrices. Similarly, other subclasses are illustrated. Determinants, inverses, and eigenvalues are attributes of square matrices, but not of general matrices. Simiarly, we would probably not use a general function to find the determinant or the eigenvalues of a triangular matrix, nor would we use a general inverse routine to obtain the inverse of a diagonal matrix.

Each derived class has all the attributes of its base classes, as well as its own unique attributes. Refer to Figure 2. A Square class is defined. This derived class contains all the attributes of the matrix class plus its own attributes. We may wish to define the inv method as part of the Square class rather than as part of the general Matrix class. Similarly, the Diag class is defined as a derived class of the Square class. Additional class methods (and members) may be defined. It should be mentioned that if the class functions of the derived classes are to have access to the members of the base class, the word "private" should be replaced with the word "protected" in the Matrix class definition.

CONCLUSIONS

We first introduce object oriented programming to students after they have nearly completed a course in Matrix Computer Techniques. By this time, they have considerable experience in procedural programming techniques. They also have considerable knowledge about a non graphical object, the matrix. We believe this introduction clearly illustrates the difference between procedural and object oriented techniques.

Although the use of the Matrix class serves as a meaningful introduction to object oriented programming, it does not give the students skills or experience in developing object oriented programs using C++. Those who are sufficiently motivated take a subsequent upper division elective called Engineering Software. Over the past few semesters, this elective has been taken by a large number of students.

Those who would like copies of the software contained in this paper or the supporting software can get copies via email. Contact "wjermann@memphis.edu".

REFERENCES

1. w. Jermann, "The Freshman Programming Course: A New Direction, Proceedings of the Annual ASEE Conference, Washington D.C., June, 1996.

2. s. Lippman, C++ PRIMER, 2nd Edition, Addison-Wesley Publishing Company, 1991.

3. J. Adams, S. Leestma, & L. Nyhoff, C++ AN INTRODUCTION TO COMPUTING, Prentice Hall, 1995.

4. BYTE, Vol. 6, No. 8, August 1981, pp 14 - 387.

```
#include <assert.h>
#include <stdio.h>
#include "a:matrix5.h" // Class definition. Overloaded operator defs.

  int main()
  {Matrix a(3,3), b(l0,l0),c;  //Instantiate Matrix objects
       a.read() ;            // Object oriented technique
       b                     // Matrix assignment (overloaded operation)
       b.print();
       c = a + b ;            // Matrix addition  (overloaded operation)
       c.print
       a = a + b + c + a ;       // Matrix addition
       a.print();   b.print();     c.print();
       a = b * c;  a.print();    // Matrix multiplication
       Matrix d,e,f;          // Invoking constructor again
       d = e = f = b;
       a = b * e.inv() + d * f.inv(); // Extended Matrix operations
       a.print();
       Matrix bi;    bi = b.inv();    // Finding an inverse
       b.print(); bi.print();
       Matrix g(3,3), h(3,l);
       double x[] = { 1, 1, 1,
                      0,  2, -1,
                      4, -3, 2,
                      6,  1, 4 };
       g.set(x); g.print();  h.set(x+9);  h.print();
       a = g.inv() * h;  // solve simultaneous equations
       a.print();
   FILE *p;
   Matrix aa(8,7);
       assert (p = fopen("a:crap.dat","r"));
       aa .read(p);   aa.print();       // Matrix method overloading
       return 0;
  }
```

Figure 1:   A C++ main program that introduces object oriented
            programming using Matrix objects

```
#include <stdio.h>
class Matrix {
       friend Matrix&
               operator *(Matrix&, Matrix&);
    public:
        Matrix(int x=5,int y=5);
        ~Matrix();
        Matrix&  operator=(Matrix &);
        //  will also overload ops + and *

        int rows(void) { return r; }
        int cols(void) { return c; }
        void setrows(int x) { r = x; }
        void setcols(int x) {c = x; }
        double get(int i, int j);
        void put(int i, int j, double x);
        void set(double *); // assigns Matrix from array of reals
        void read(FILE * p=stdin);  // matread
        void print(void); // matprint
        Matrix& inv(); // returns inverse
        double * getptr(void) { return ptr; }
        void setptr(double *p) {ptr = p; }
     private:
        int r;
        int c;
        double *ptr;
     public:
        virtual void dumb();
   };   // end of class definition

class Square:  public Matrix {
      public:  Square(int n = 3);
               void dumb();
      };

class Diag:    public Square {
      public:  Diag(int n=4);
               void dumb();
     };

Matrix& operator +(Matrix&,Matrix&);
Matrix& operator *(Matrix&,Matrix&);
```

Figure 2: Definition of a Matrix class and overloaded operators + and * (contained in header file, matrix5.h)

```
#include "a:matrix5."
#include "a:csubs5.h"
#include <assert.h>
#include <stdio.h>

   Matrix::Matrix(int m, int n)
     { r = m; c = n; ptr = new double[m*n]; assert(ptr);
       int i; for(i=O; i< m*n; i++) ptr[i] = 0; }

   Matrix::~Matrix()
         { delete ptr ; }

   Matrix& Matrix::operator=(Matrix &x)
      ( delete ptr;
        r = x.rows(); c = x.cols();
        ptr = new double[r*c];
        double *p; p = x.getptr();
        int i; for(i=O; i< r*c; i++) ptr[i]=p[i];
        return *this; }

   Matrix&  operator*(Matrix &x, Matrix &y)  // A friend
   {  assert(x.c == y.r );
      static Matrix z;  delete z.ptr;  z.r = x.r;
      z.c = y.c;
      z.ptr = new double[z.r * z.c;
      matmul(x.r,y.c,x.c,x.ptr,x.c,y.ptr,y.c,z.ptr,,z.c  );
     return z;
}

   Matrix&  operator+(Matrix &x, Matrix &y) // not a friend
   (assert(x.rows() == y.rows() ); assert(x.cols() == y.cols());
     static Matrix z; z = x;
     matadd(x.rows(),x.cols(),x.getptr(),x.cols(),
                          y.getptr(),y.cols(),z.getptr(),z.cols());
     return z;
   }
```

Figure 3: Class methods and operators for the Matrix class

```
    double Matrix::get(int i, int j)
       { return cget(i,j,ptr,c); }

    void Matrix::put(int i, int j, double x)
      { cput,j,x,ptr,c); }

    void Matrix::read(FILE *p)
      { matread(r,c,ptr,c,p); }

    void Matrix::print(void)
     ( matprint(r,c,ptr,c); }

    Matrix& Matrix::inv(void)
    {  assert( r == c);
        static Matrix x;  delete x.getptr();
       x.setrows(r);  x.setcols(c);
       x.setptr(new double[r*c] );
       assert(!inverse(r,ptr,c,x.getptr(),x.cols()));
       return ( x);
   }

  void Matrix:: set(douhle *x)
    {int i,j; for(i=l; i<=r; i++)
                   for(j=l; j<=c; j++)   (*this).put(i,j,*x++);
    }

#include <iostream.h>

void Matrix::dumb() {cout << "Matrix\n" ; }

     Square:: Square(int n)
              : Matrix(n,n)
                { }
     Diag:: Diag(int n)
              : Square(n)
                { }
     void Diag::dumb() {cout << "Diagonal\n" ; }

     void Square:: dumb() {cout << "Square\n" ; }
```

Figure 4: Methods for the Matrix class and its derived classes

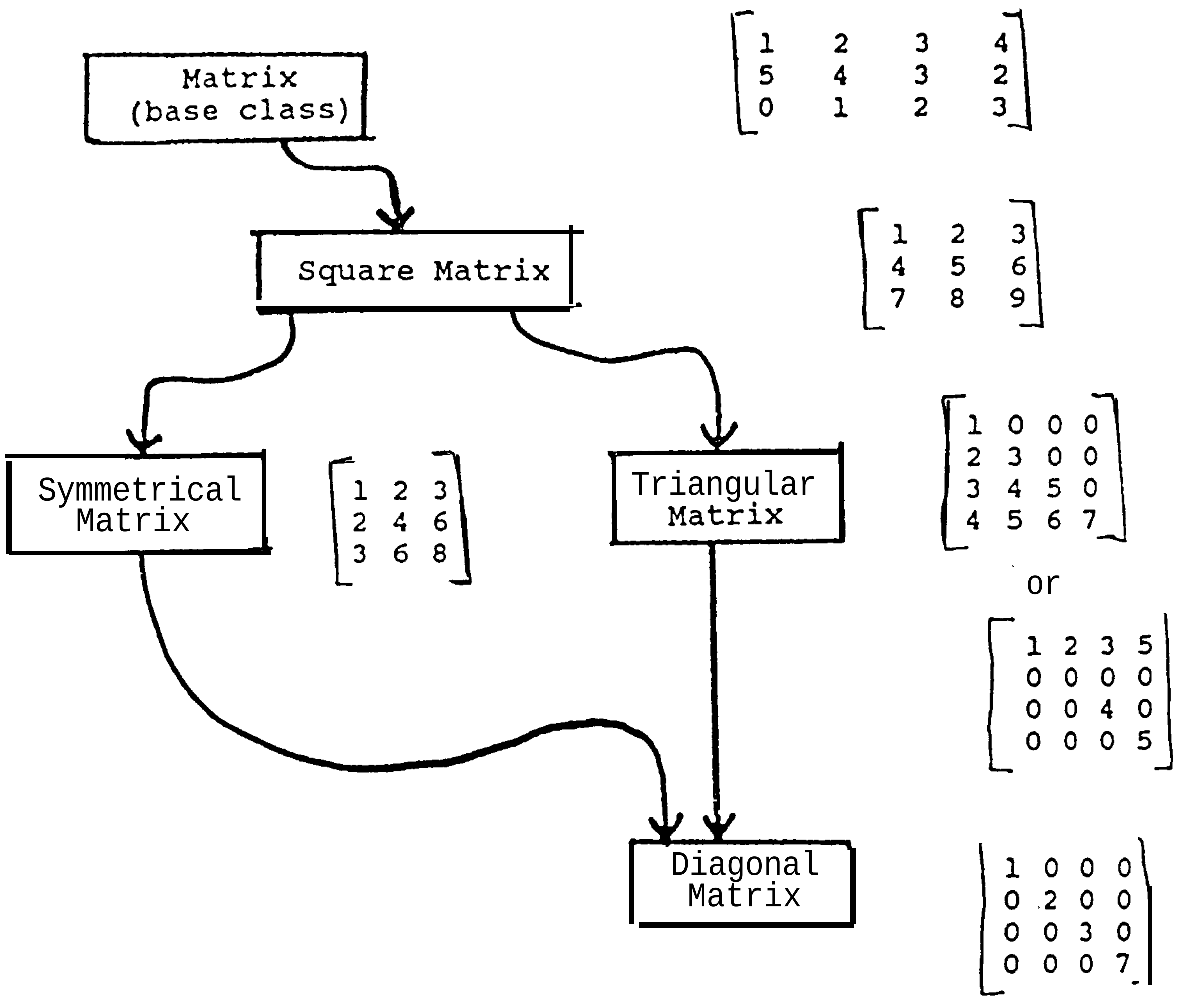

Figure 5. Illustration of a matrix base class and derived classes