C++ Programming for Scientists

Lecture # 4

C++ classes (quick review)

- a C++ class is essentially a C `struct` that also contains functions as member. It also has `private` and `public` sections which specify which portions are visible from application codes, e.g.

```
class C
{ private:
        float a;
        Book b;
        int  i;
    public:
        float f(int);
        /* ... */
}
```

- We say that `a`, `b` and `i` are *private members* of class `C`. Meanwhile, `C::f()` is a *public member function*. Classes can have other objects (e.g. `Book`) as private or public members.

- Members of a class (both data and function) default to `private` declarations, unless explicitly declared `public`. In the example above the word `private` could have been removed; however, programmers commonly leave it in for clarity.

C++ classes (cont'd.)

- Data members can be made `public`, although typically they are controlled through *access* functions. For example,

```
class Complex
{   private:
        double real_;
        double img_;
    public:
        Complex(double real, double img){ real_=real; img_=img;}
        double real() { return real_;}
        double img(){ return img_;}
        void setreal(double x){ real_ = x;}
        void setimg(double x){ img_ = x;}
        void set(double x, double y){ real_ = x; img_ = y;}
        /* ... */
};
```

  Used as

```
Complex u(0.0, 0.0);    // u = 0 + 0i

u.set(1.2, 3.7);        // u = 1.2 + 3.7i
u.setimg(4.9);          // u = 1.2 + 4.9i
```

- Why not just make the real and imaginary parts public and access them directly?

C++ constructors (revisited)

Constructors: (optional, but highly recommended.) If not specified, each object will be declare in an "unknown" state, much like when declaring `ints` or `floats` without initialization. Given a class `C`,

- copy constructor

  `C::C(const C&)`

  This is how one makes a *copy* of an object. This constructor is commonly called by the compiler, for example, when returning objects (by value) from functions and when requiring temporaries in subexpressions.

- null constructor:

  `C::C()`

  This is what the compiler calls when creating arrays of objects. For example,

```
BigInt A[10];
```

  creates ten `BigInts`, calling the null constructor on each (we've defined this to set each to "0".) To "initialize" the array with values other than this, you have to reset by hand:

```
BigInt A[10];

for (int i=0; i<10; i++)
    A[i] = "3145";
```

  There is no mechanism in C++ for doing this automatically.

Programming Tips

- Make sure that the `C::operator=` has the same semantics as the copy constructor, `C::C(const &X)`. Remember that

```
BigInt A = "385928498912";
```

  calls the copy constructor, *not* `BigInt::operator=` (!!) That is, the above is equivalent to

```
BigInt A("385928498912");
```

- Creating objects on the spot:

```
Complex *t = new Complex(1.2, 6.4);
```

  Note that `t` is a pointer to a `Complex`. It is utilized as

```
t->setimg(2.8);     // *t is 1.2 + 2.8i
```

  and must be explicited freed later, as

```
delete t;
```

- Aliases: use the `&` operator to denote different names for variables, e.g.

```
BigInt m = "49837609936516734"
BigInt i = "182352410683751";
BigInt *j = i;
BigInt &k = i;
```

  makes `k` another name for `i`. Both `j` and `k` reference `i`, but `k` looks more like another variable, rather than a pointer, e.g.

```
i = m + 50;            // these satements are the same;
k = m + 50;
j->operator=(m+50);
```


Operators (revisited)

Given that `A`,`B`,`C`, and `D` are classes, remember that

```
A + B
```

is shorthand for

```
operator+(A,B)
```

or

```
A.operator+(B)
```

if `+` is a member function of `A`.

Similarly

```
C = A + B
```

is the typically the same as

```
C.operator=(operator+(A,B));
```

The overloaded operators (`+, *, ||, /, =,` etc.) have the same *precedence* as the native counterparts, i.e. `*` has higher precedence than `+`, and son on.

For example,

$$C = A + B * D \qquad ;$$

is equivalent to

$$C = A + (B * D);$$

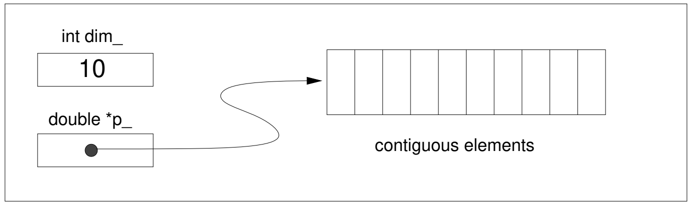
Adding vectors and matrices to C++

GOAL: make numerical vectors look like a natural part of C++.

Would like to see the following:

```
Vector A(N);                // create vectors with  N items
Vector B(N);                //   each element is initialized to 0.0

Vector C(N, 1.0);           // initialize N elments to 1.0

double a[] = {2.4, 6.8, 4.2, 9.9};
Vector D(4, a);                     // intialize from C array


C[i] = A[j] * B[k];         // access like regular C arrays


C = A + B;                  // vector arithmetic


B = A;                      // vector copy


C = 0.0;                    // a quick way to set all elements to a
```


Vector declaration (vector.h)



class Vector

```
class Vector
{
  private:
    int dim_;                                 // size of vector
    double *p_;                               // memory where data
                                              //    is kept.
  public:

    Vector();                                     // constructors
    Vector(unsigned int, double t=0.0);
    Vector(unsigned int, const double*);
    Vector(const Vector &);
    ~Vector();                                    // destructor


                                                  // access functions
    double&     operator[](int i){return p_[i];}
    int         size() const { return dim_;}
    int         null() const {return dim_== 0;}

    Vector& operator=(const Vector&);   // assignment
    Vector& operator=(double);
};
```

Vector declaration (cont'd.)

- Note that the `Vector::Vector(int, double)` constructor defaults elements to a value of 0.0.

- Note that `Vector::size()` returns the size of the of the vector, but it *cannot* be modified (thus ensuring integrity of the data structure).

- Note that there are two constructors, one that creates an new vector filled with equal scalar values (defaulting to 0.0), and another which makes a *copy* of a regular C array.

- Note that the `Vector::operator=` returns a *reference type*, (i.e. `double&` which can be modified, i.e. in

  ```
  A[i] = /* ... */
  ```

  Otherwise the elements of vector `A` could never be modified.

Problems with Vector

- what's wrong with the following code?

  ```
  double first_elm(const Vector &A)
  {
     return A[0];
  }
  ```

  Here's the actual compiler output (Sun CC v. 4.0.1):

  ```
  Error: Non-const function Vector::operator[](int) called for const object.
  ```

  What happened? The `operator[]` returns a value *address* (remember the `&` in the function signature) in which one can modify its contents, but the declaration of `A` inside function `foo()` is `const` –unmodifiable!

  That is, the compiler can't tell when it sees `A[i]` if one is modifying `A`, so it has to assume the worst, and throw an error.

- What to do? Add a `const` version of `operator[]` to `Vector`:

  ```
  double & operator[](int i);           // the old operator[]
  double   operator[](int i) const;     // new const-safe version
  ```

  Notice that the `const` version return a `double` by *value* not by *reference*; thus, there is no way for that operator to modify the actual values of `A`.

A Better Vector class

- add a `const` version of `operator[]`.
- make `operator[]` check for legal range:

```
#include <assert.h>
double Vector::operator[](int i) const
{
    assert(i >= 0 && i < size());
    return p_[i];
}
```

The `assert()` function is actually a macro that takes a boolean expression and will stop your program if the condition is not met. It will print something like:

```
Assertion failed: file "vector.cc", line 37.
```

where line 37 is where the `assert()` function failed in your code. (This is not the best method of error handling; we'll talk about this later..)

- add `operator+()` and `operator*()` for vector addition and dot product. Make these operators check that both vectors are of the same size, e.g.

```
Vector operator+(const Vector &A, const Vector& B)
{
    assert( A.size() == B.size() );

    int N = A.size();
    Vector C(N);

    for (int i=0; i<N; i++)
        C[i] = A[i] + B[i];

    return C;
}
```


Now let's add matrices to C++...

GOAL: make numerical matrices look like a natural part of C++.

Like `Vectors`, we'd like to see the following:

```
Matrix A(M,N);                // create matrices with  M*N items
Matrix B(M,N);                // stored column ordered (ala Fortran).
                              // each element is initialized to 0.0

Matrix C(N,N, 1.0);           // initialize elments to 1.0

double a[] = {2.4, 6.8, 4.2, 9.9};
Matrix D(2, 2, a);                    // initialize from contiguous C
                                      // array, in column major order.
                                      // i.e. {2.4, 6.8} form the
                                      // first column


C(i,j) = A(i,j) * B(j,k);         // access like Fortran arrays
                                  // but remember:  first element
                                  // is at (0,0) not (1,1).

                                 // NOTE: we can't use [] anymore,
                                 // because it only allows one
                                 // argument.


C = A + B;                   // matrix arithmetic (A and B must be
                             // the same size)


B = A;                       // matrix copy (A and B must conform.)


C = 0.0;                     // a quick way to set all elements to
                             //   one single scalar.
```

Matrix declaration (matrix.h)

$$\begin{pmatrix} a_{00} & a_{10} & a_{20} & a_{30} \\ a_{01} & a_{11} & a_{21} & a_{31} \\ a_{02} & a_{12} & a_{22} & a_{32} \\ a_{03} & a_{13} & a_{23} & a_{33} \end{pmatrix}$$

```
class Matrix
{
  private:

    Vector v_;                            // contiguous array is really
                                          //    just one long vector.

    int dim_[2];                          //  the matrix size in each
                                          //    dimension
  public:

    // constructors
    Matrix();
    Matrix(int, int, double t=0.0);
    Matrix(int, int, const double*);
    Matrix(const Matrix &);

    // destructor
    ~Matrix();


    // access functions

    double&     operator()(int i, int j);
    double      operator()(int i, int j) const;
    int         size(int i) const;
    int         null() const {return (dim_[0] == 0 || dim_[1] == 0); }

    Matrix& operator=(const Matrix&);   // assignment
    Matrix& operator=(double);

};
```


Some details of the Matrix implemenatation

- How do we initialize the private `Vector` member when creating a `Matrix`?

  The constructor for the `Vector` goes right after the function signature, separated by a colon, e.g.

```
Matrix::Matrix(int M, int N, double s) : v_(M*N,s)
{
       dim_[0] = M;
       dim_[1] = N;
}
```

  In general if class `A` contains classes `a`, `b` and `c` as private members, they are constructed as

```
A::A() : a(), b(), c()
{ /* ... */ }
```

  where the `()` can be replaced with any of the valid constructors for that class.

- How do we access elements in the matrix?

  Can no longer use the `operator[]` because it only allows one argument, i.e. we can't write `A[i,j]`. [1] Instead we'll use the natural `operator()` for matrix indexing. Since the elements are arranged in colum order form, we perform simple indexing arithmetic for compute the address of the element:

```
double Matrix::operator()(int i, int j) const
{
    return v_[ j * dim_[0] + i ] ;
}
```

  We can also check that the indicies are within range with an `assert()` function, like we did with the `Vector` class.

[1] Well, we could, but it would be rather meaningless. Remember that the comma operator in C acts as a way to group several expressions together, while returning the value of the last expression. Thus `A[i,j]` is equivalent to `A[j]`. Probably not what one intended.

So C++ now has vectors and matrices...

SO WHAT?

... languages like Fortran 90, Matlab, and Mathematica have these as well.

... is this any better?

Matrices come in many flavors...

Hermitian

Storage	Hermitian matrix A	Storage in array A
`Upper`	$\begin{pmatrix} a_{00} & a_{10} & a_{20} & a_{30} \\ \bar{a}_{01} & a_{11} & a_{21} & a_{31} \\ \bar{a}_{02} & \bar{a}_{12} & a_{22} & a_{32} \\ \bar{a}_{03} & \bar{a}_{13} & \bar{a}_{23} & a_{33} \end{pmatrix}$	$\begin{matrix} a_{00} & a_{01} & a_{02} & a_{03} \\ * & a_{11} & a_{12} & a_{13} \\ * & * & a_{22} & a_{23} \\ * & * & * & a_{33} \end{matrix}$
`Lower`	$\begin{pmatrix} a_{00} & \bar{a}_{10} & \bar{a}_{20} & \bar{a}_{30} \\ a_{10} & a_{11} & \bar{a}_{21} & \bar{a}_{31} \\ a_{20} & a_{21} & a_{22} & \bar{a}_{32} \\ a_{30} & a_{31} & a_{32} & a_{33} \end{pmatrix}$	$\begin{matrix} a_{00} & * & * & * \\ a_{10} & a_{11} & * & * \\ a_{20} & a_{21} & a_{22} & * \\ a_{30} & a_{31} & a_{32} & a_{33} \end{matrix}$

Triangular

Storage	Triangular matrix A	Storage in array A
`Upper`	$\begin{pmatrix} a_{00} & a_{01} & a_{02} & a_{03} \\ & a_{11} & a_{12} & a_{13} \\ & & a_{22} & a_{23} \\ & & & a_{33} \end{pmatrix}$	$\begin{matrix} a_{00} & a_{01} & a_{02} & a_{03} \\ * & a_{11} & a_{12} & a_{13} \\ * & * & a_{22} & a_{23} \\ * & * & * & a_{33} \end{matrix}$
`Lower`	$\begin{pmatrix} a_{00} & & & \\ a_{10} & a_{11} & & \\ a_{20} & a_{21} & a_{22} & \\ a_{30} & a_{31} & a_{32} & a_{33} \end{pmatrix}$	$\begin{matrix} a_{00} & * & * & * \\ a_{10} & a_{11} & * & * \\ a_{20} & a_{21} & a_{22} & * \\ a_{30} & a_{31} & a_{32} & a_{33} \end{matrix}$

Banded

Band matrix A	Band storage in array AB
$\begin{pmatrix} a_{00} & a_{01} & & & \\ a_{10} & a_{11} & a_{12} & & \\ a_{20} & a_{21} & a_{22} & a_{23} & \\ & a_{31} & a_{32} & a_{33} & a_{34} \\ & & a_{42} & a_{43} & a_{44} \end{pmatrix}$	$\begin{matrix} * & a_{01} & a_{12} & a_{23} & a_{34} \\ a_{00} & a_{11} & a_{22} & a_{33} & a_{44} \\ a_{10} & a_{21} & a_{32} & a_{43} & * \\ a_{20} & a_{31} & a_{42} & * & * \end{matrix}$

Triangular Packed

Storage	Triangular matrix A	Packed storage in array AP
`Upper`	$\begin{pmatrix} a_{00} & a_{01} & a_{02} & a_{03} \\ & a_{11} & a_{12} & a_{13} \\ & & a_{22} & a_{23} \\ & & & a_{33} \end{pmatrix}$	$a_{00}\ \underbrace{a_{01}\,a_{11}}\ \underbrace{a_{02}\,a_{12}\,a_{22}}\ \underbrace{a_{03}\,a_{13}\,a_{23}\,a_{33}}$
`Lower`	$\begin{pmatrix} a_{00} & & & \\ a_{10} & a_{11} & & \\ a_{20} & a_{21} & a_{22} & \\ a_{30} & a_{31} & a_{32} & a_{33} \end{pmatrix}$	$\underbrace{a_{00}\,a_{10}\,a_{20}\,a_{30}}\ \underbrace{a_{11}\,a_{21}\,a_{31}}\ \underbrace{a_{22}\,a_{32}}\ a_{33}$

Orthogonal

$$Q = \underbrace{H_1 H_2 \ldots H_q}_{\text{Householder vectors}}$$

Consider also...

- Unstructured Sparse Matrices
  - compressed row
  - compressed column (Harwell-Boeing)
  - coordinate
  - skyline
  - ITPACK format
  - blocked compressed column
  - block compressed
  - symmetric variants of the above..
- Distributed (Parallel) Matrices
  - 1-d cyclic
  - 1-d block
  - 2-d square-block scattered
  - 2-d block/cyclic
  - random scatter
  - dynamic distributions of above...
- Algebraic Operators (matrix not explicitly formed)
  - stiffness matrices from finite-element methods
  - implicit operators for preconditioning
  - ...

Reusing code for matrix algorithms

For example, in solving $Ax = b$ using a preconditioned conjugate gradient method, we'd like to have only *one* piece of code to handle *all* matrix types:

Initial $r^{(0)} = b - Ax^{(0)}$
for $i = 1, 2, \ldots$
  solve $Mz^{(i-1)} = r^{(i-1)}$
  $\rho_{i-1} = r^{(i-1)^T} z^{(i-1)}$
  if $i = 1$
    $p^{(1)} = z^{(0)}$
  else
    $\beta_{i-1} = \rho_{i-1}/\rho_{i-2}$
    $p^{(i)} = z^{(i-1)} + \beta_{i-1} p^{(i-1)}$
  endif
  $q^{(i)} = Ap^{(i)}$
  $\alpha_i = \rho_{i-1}/p^{(i)^T} q^{(i)}$
  $x^{(i)} = x^{(i-1)} + \alpha_i p^{(i)}$
  $r^{(i)} = r^{(i-1)} - \alpha_i q^{(i)}$
  check convergence;
end

```
r = b - Ax;
for (int i=1; i<maxiter; i++){
    z = M.solve(r);
    rho = r * z;
    if (i==1)
       p = z;
    else{
       beta = rho1/ rho0;
       p = z + p * beta;
    }
    q = A*p;
    alpha = rho1 / (p*q);
    x += alpha * p;
    r -= alpha * q;
    if (norm(r)/normb < tol) break;
}
```

we'll see how to do this in the next two lectures...

Reminder...

Don't forget the *Grand Finale* next time at:

2:00-4:00 pm

Friday, July 14th.

Homework #4

1. Finish the `Vector` class example we started out in lecture. Experiment with the following features:
   - has a const-safe version of `operator[]`
   - uses `assert()` to simple error checking
   - has the basic operators `*, +, -` defined on the vectors
   - make sure the assignment operator (`operator=`) agrees with your copy constructor
2. Now that you've build your `Vector` class, how hard would it be to build a `Vector` of `int`s?
3. **Extra for experts:** Our `Vector` class is missing one very important function: there is no way to *resize* it to make it bigger or smaller after it has been declared.

   Add a member function `resize()` to `Vector` to accomplish this. (Remember, you can't make it a `const` function, since it modifies the `Vector` object.)
4. Finish out the `Matrix` example. Include similar functionality to the `Vector` class above. The `operator*` should perform matrix multiply, not an element-by-element multiplication. (An `Array` class would be better suited for that.)