PHYS4070/7270 Worksheet: Week 6 (01/04/2021)

This work sheet is broken into two parts.
Part A acts as a basic introduction to using arrays and classes in c++, and part B is the actual worksheet problems.
Feel free to skip the part A and move straight to the problems in B - you can refer back to part A if need be.

Part A: Intro to c++ arrays, matrices, classes

You are not expected to know all the ins-and-outs of c++ - this will not "be on the test".
However, you may find it useful to have a basic understanding of these concepts; they will help you understand what is going on and why, and will hopefully be helpful for the assignment.
You can also access an even more basic c++ "cheat sheet" here:

- https://github.com/benroberts999/cpp-cheatsheet
- (also in course git repo)

This is by no means a complete overview - just a very basic introduction.

1. Crash-course: Memory, data, arrays

Pointers: memory locations

- Pointers are variables that store memory addresses (memory locations)
- You do not really need this for this class, however, a basic understanding will help with understanding why certain things are the way they are.

```
double x = 3.14;
std::cout << x << '\n';
// '&' symbol gets the memory address:
std::cout << &x << '\n';
// Define a pointer variable (pointer to double), px,
// set it equal to the memory location of x:
double *px = &x;
std::cout << px << '\n';
// Here, the '*' symbol "dereferences" the pointer, accesses the value
std::cout << *px << '\n';
```

Old C-style arrays

- Contiguous chunks of memory
- see: http://www.cplusplus.com/doc/tutorial/arrays/

```
double a[3] = {40.70, 72.70, 2021.0};
std::cout << a[0] << ", " << a[1] << ", " << a[2] << '\n';
// The memory location of a c-style array is the memory location of its first element
std::cout << &a << ' ' << &(a[0]) << '\n';
// The memory location are off-set by 8 bytes (in hex)
std::cout << sizeof(double) << '\n';
std::cout << &a[0] << ", " << &a[1] << ", " << &a[2] << '\n';
// No array bounds checking: a[x] just means go 'x' memory slots after a[0]
// This would work, but is undefined behaviour
// std::cout << a[3] << '\n'; // woops!
```

- The way I've done it here, the size of the matrix must be known at compile time - this is *static* memory allocation
- It is also possible to create c-style arrays where the dimension is not known until runtime (i.e., can re-size the array). This is known as *dynamic* memory allocation, and is done with the `new` and `delete` keywords
- In modern c++, we rarely (if ever) need to do this; instead we will use std::array for fixed-size (static) arrays, and std::vector for variable-sized (dynamic0 arrays

c++ style static array: std::array

- std::array is a fixed-size sequence container: holds a specific number of elements ordered in a strict linear sequence
- Size is constant, must be known at compile time
- see: https://www.cplusplus.com/reference/array/array/
- see: https://en.cppreference.com/w/cpp/container/array

```
std::array<double, 3> b{40.70, 72.70, 20.21};
std::cout << b[0] << ", " << b[1] << ", " << b[2] << '\n';
// Data still stored in contiguous block
std::cout << &b[0] << ", " << &b[1] << ", " << &b[2] << '\n';
// Knows its own size:
std::cout << b.size() << '\n';
// Have array-bounds checking
std::cout << b.at(0) << ", " << b.at(1) << ", " << b.at(2) << '\n';
// This would *not* work - will get sensible error message
// std::cout << b.at(3) << '\n';
// Ranged-based for loops
for(double element : b){
    std::cout << element << '\n';
}
```

c++ style dynamic array: std::vector

- std::vector is a sequence container representing an array that can change in size
- Data is stored contiguously (in single block), compatible with c-style array
- see: https://www.cplusplus.com/reference/vector/vector/
- see: https://en.cppreference.com/w/cpp/container/vector
- In general: std::vector should often be your default data-structure

```
std::vector<double> v1(3); // this auto-fills 3-element vector with 0s
// create vector with 2 elements:
std::vector<double> v{40.70, 72.70};
std::cout << v.size() << '\n';
// and add a third:
v.push_back(20.21);
std::cout << v.size() << '\n';
// Data also stored in contiguous block
std::cout << &v[0] << ", " << &v[1] << ", " << &v[2] << '\n';
// But memory address of v _not_ same as first element:
std::cout << &v << ' ' << &(v[0]) << '\n';
// To use existing (C/Fortran) libraries, we can access the underlying c-style array with .data()
std::cout << v.data() << ' ' << &(v[0]) << '\n';
```

- .data() -- returns a pointer to underlying data (compatible with c-style array)
- .resize(n) -- changes size of vector to 'n' (fills new elements with zero)

2. STL: Standard Template Library

- STL is an enormous collection of algorithms
- Many located in `<algorithm>` and `<numeric>` headers
- Common ones include: std::sort, std::accumulate, std::for_each

```
std::vector<int> v{6,3,7,9,1,0,2,5};

for(auto x : v){
    std::cout << x << ", ";
}
std::cout << '\n';

std::sort(v.begin(), v.end());

for(auto x : v){
    std::cout << x << ", ";
}
std::cout << '\n';

int sum = std::accumulate(v.begin(), v.end(), 0);
std::cout << sum << '\n';
```

3. Storing matrix in c++

- Focus on 2D matrix; generalisation to multi-dimension matrix is fairly simple
- Old C-style (for fixed-size, static memory) array:

```
static const int dim = 2; //'static const': must be compile-time constant!
double x[dim][dim] = {0.5, 1.5, 2.5, 3.5};
for(int i=0; i<dim; i++){
    for(int j=0; j<dim; j++){
        std::cout << x[i][j] << ' ';
    }
    std::cout << '\n';
}

// Exactly equivalent to:
double y[dim*dim] = {0.5, 1.5, 2.5, 3.5};
for(int i=0; i<dim; i++){
    for(int j=0; j<dim; j++){
        std::cout << y[i*dim + j] << ' ';
    }
    std::cout << '\n';
}
```

- The 2D array is actually stored as a single chunk of memory (i.e. in 1 dimension), and [i][j] is just short-hand for (i*dim+j)
- Can do this using dynamic memory allocation too, but we will not; in c++ there are nicer ways
- In modern c++, we would not use a basic c-array, but instead use a class (data structure from a library). Examples below.

nested classes:

- It is possible to declare an std::array of std::arrays:
- This is exactly identical to the above c-style array
- Though, you can see that this will get cumbersome quickly

```
std::array<std::array<double,dim>,dim> z{0.5, 1.5, 2.5, 3.5};
for(int i=0; i<dim; i++){
    for(int j=0; j<dim; j++){
        std::cout << z[i][j] << ' ';
    }
    std::cout << '\n';
}
```

- It is also possible to store a std::vector of std::vectors.
- `std::vector<std::vector<double>> x;`

- HOWEVER, this data is *not* stored as a contiguous array, so this is *not* equivalent to the c-style array
- We *can* however use an std::vector to store a 2D matrix, we just need to access the elements using the i*dim+j
  - nb: in some old versions of c++ (old compilers), you were not allowed to have double `>>` (since this is a special operator)
  - i.e.: `std::vector<std::vector<double>> x;` --> `std::vector<std::vector<double> > x;` (extra space)
  - This essentially was a bug, which has been fixed (but may persist in old compilers, such as those on smp-teaching)

Using regular std::vector, wrap in a class:

- We *can* however use an std::vector to store a 2D matrix, we just need to access the elements using the i*dim+j

```
  std::vector<double> v{0.5, 1.5, 2.5, 3.5};
  int dim = 2; //regular int
  for(int i=0; i<dim; i++){
      for(int j=0; j<dim; j++){
          std::cout << v.at(i*dim + j)<<" ";
      }
      std::cout<<'\n';
  }
```

- In order to make things easier, we will make our own class that holds a matrix using std::vector to store data
- In real-world code, many such classes exist already, and we would use one of these matrix classes (e.g., from the great 'Eigen' library)
- But here, we will "re-invent the wheel", since it is a good learning exercise, and will greatly help you understand classes in `c++`

4. Classes

- Classes (and structs) are data structures that contain data (member variables), and functions (sometimes called methods)
- Some are provided by c++ libraries; we can also write our own

```
class MyClass{};
struct MyStruct{};
```

- Notice the semi-colon
- Only difference between class and struct: members in class are *private* by default; those of struct are *public* by default
- Public mean accessible outside the class; private means not
- By convention, structs are used for very small objects (one or two members, no functions), and classes are used for large ones with functions

```
// define a class:
class MyClass{
private:
  int a = 1;
public:
  int b = 2;
};

// Construct object of type 'MyClass', call it mc:
MyClass mc;
std::cout << mc.b << '\n';
// std::cout << mc.a << '\n'; - does not work, "a is private in this scope"
```

Constructor, member initialiser list

- Constructor is a function that is called
- *member initializer list* initialises member variables
    - funny syntax: a colon ':' after (), but before {}

```
class MyClass{
public:

  // data stored in the class
  int a;

  // Constructor, using member initializer list:
  MyClass(int in_a) : a(in_a) {}

  // We could also write it as:
  // MyClass(int in_a){
  //   a = in_a;
  // }
  // However, in this case, the int a in constructed, then set to in_a
  // With the member initializer list, these happen at the same time
};

// Use it:
MyClass mc{4};
std::cout << mc.a << '\n';
```

Member functions

- Member functions work just like other functions, but have access to class data
- called using the '.'
    - object.function()

```
class MyClass{
private:
  int b = 7;

public:

  int a;

  MyClass(int in_a) : a(in_a) {}


  // Member function:
  double square_a(){
    return a*a;
  }

 // Can use member functions to interface with private variables
  void print_b(){
    std::cout << b << '\n';
  }

};

// Use it:
MyClass mc{4};
std::cout << mc.a << '\n';
std::cout << mc.square_a() << '\n';
mc.print_b();
```

Operator overloading

- In c++, we may define operators that act on our own user-defined classes
- We may overload: +,-,/,*,==,>,< etc. etc.
- This is very useful: and leads to nice-looking code. e.g.,
  - Matrix3 = Matrix1 * Matrix2, instead of
  - Matrix3 = matrix_multiplication(Matrix1,Matrix2);

```
class MyClass{
public:

  int a;

  MyClass(int in_a) : a(in_a) {}

  friend MyClass operator+(MyClass lhs, MyClass rhs){
    int sum_of_a =  lhs.a + rhs.a;
    MyClass result{sum_of_a};
    return result;
  }

};

// Use it:
MyClass mc1{4};
MyClass mc2{6};
MyClass mc3 = mc1 + mc2;
std::cout << mc1.a << '\n';
std::cout << mc2.a << '\n';
std::cout << mc3.a << '\n';
```

Part B: Worksheet Tasks

1. Write a class to store a 2D square matrix

- Use std::vector to hold data
- The constructor should take the dimension N as input, and create a vector of correct length (N*N)
- Provide a member function to return .data() from vector, so we can access the c-style array (needed to interface with lapack)
  - `double * get_data() { return v.data(); }`
- Provide a function that allows us to read and edit the i,j element
  - To edit, this must return a *reference*, e.g.,
  - `double & at(int j, int j) { return v.at(i*dimension + j); }`
  - This allows, e.g., `x = matrix.at(i,j);` *and* `matrix.at(i,j) = x;`
- Provide operator overload of '+', that allows us to add two matrices together

2. Write a function that takes in a matrix (class you created above) and find the eigenvalues and eigenvectors

- Assume the matrix is real and symmetric, so use LAPACK function DSYV
- Documentation: http://www.netlib.org/lapack/explore-html/index.html
- In/out parameters listed below:

```
int dsyev_(
  char * jobz,  // 'V' = compute e. values and vectors. 'N' = values only
  char * uplo,  // 'U' = upper triangle of matrix is stored, 'L' = lower
  int * n,      // dimension of matrix a
  double * a,   // c-style array for matrix a (ptr to array, pointer to a[0])
                // On output, a contains matrix of eigenvectors
  int * lda,    // For us, lda=n
  double * w,   // array of dimension n - will hold eigenvalues
  double * work,// 'workspace': array of dimension lwork
  int * lwork,  // dimension of workspace: ~ 6*n works well
  int * info    // error code: 0=worked.
);
```

- This function should return the eigen values *and* vectors [eigenvectors are stored in a matrix (2D array)]
- Eigenvalues are sorted, and eigenvectors are normalised to 1 (via inner product)
- Normally, functions in c++ return only one thing. We have two options:
- A: Pass in/out parameters to function by reference like this:
  - `void solveEigenSystem(Matrix matrix, Matrix &eigenvectors, Vector &eigenvalues);`
  - This is typically frowned upon, since it makes code difficult to read (which value is input, which is output?)
- B: Define a class/struct that holds a matrix of eigenvectors and a vector of eigenvalues, (e.g., called *MatrixAndVector*), and returns this
  - `MatrixAndVector solveEigenSystem(Matrix matrix);`
- Note: FORTRAN (language LAPACK is written in) uses column-major ordering to access 2D arrays, wile c and c++ use row-major. This means m[i][j] in c++ is m[j][i] in FORTRAN.. so we often need to transpose the matrix before sending to LAPACK
  - Our matrix is symmetric, so this doesn't matter, except for 'uplo'
  - 'uplo': 'U' means upper triangle *in FORTRAN* is stored -- so lower in c++ [we can just fill entire matrix though]
  - For other LAPACK functions, you can often just tell them the matrix is a transpose, so we don't need to waste time transposing it ourselves
- Don't forget `extern "C"`, and to declare the `dsyev_` function. and the -llapack linker (compile) flag (on macos, you may also need the -lblas flag)

3. Use your code to calculate eigen values/vectors of simple 2x2 matrix

`m_ij := 1.0 / (i + j + 1.0), (for i & j = 0,1)`

- Expected eigenvalues should be: {1.26759, 0.0657415}
- With corresponding eigenvectors: {{1.86852, 1.}, {-0.535184, 1.}}

4. Quantum simple harmonic oscillator

- The Hamiltonian of QSHO, in simplest units case, is
  - `H = p^2/2 + x^2/2`
  - p is momentum operator, x is position
- Use finite-difference method to solve Shrodinger equation on interval x=[-5,5] by casting problem to matrix eigenvalue problem
  - Encode derivative operator as a matrix: (… 1, -2, 1, 0, …) / dx^2
    - dx = (xmax - xmin)/Nsteps
  - Form full symmetric Hamiltonian matrix
  - Use hard boundary condition
  - Probably need at least a few hundred steps
- Compare eigenvalues to known energies: En = (n + 1/2)
- We also have a full set of orthogonal wavefunctions (eigenvectors). These are not yet properly normalised
- Check that the first two wavefunctions (eigenvectors) are indeed orthogonal
- Plot First 3 wavefunctions - do the look how you expect?