Arrays in c++: std::vector and more

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.

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

See also the examples at: https://github.com/benroberts999/cpp-cheatsheet

Pointers, memory locations, arrays of data

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.

Interactive Example: https://godbolt.org/z/cqTb33aG8

```
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';

// Following are equivilant
// double* px = &x;
// double * px = &x;
// double *px = &x;

// Here, the '*' symbol "de-references" the pointer, accesses the value
std::cout << *px << '\n';
```

Old C-style arrays (we don't do this anymore)

- 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
    - (If you're coming from c, these are the c++ versions of malloc and free)
- 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 (dynamic) arrays

---

std::vector

c++ style dynamic array: std::vector - this should be your *go-to* container for arrays

- 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

Interactive Example: https://godbolt.org/z/bvPKn518h

```
std::vector<double> v1(3);  // this auto-fills 3-element vector with 0s
// create vector with 2 elements:
std::vector<double> v2{40.70, 72.70};

// knows its own size:
std::cout << v2.size() << '\n';
// and add a third element:
v2.push_back(20.21);
std::cout << v2.size() << '\n';
```

```
// access elements, with array-bounds checking
std::cout << v2.at(0) << ", " << v2.at(1) << ", " << v2.at(2) << '\n';
// This would *not* work - will get sensible error message
// std::cout << b.at(3) << '\n';
// can also access without checking (prefer not to do this)
std::cout << v2[0] << ", " << v2[1] << ", " << v2[2] << '\n';

// Data stored in contiguous block
std::cout << &v2[0] << ", " << &v2[1] << ", " << &v2[2] << '\n';

// But memory address of v _not_ same as first element:
std::cout << &v2 << ' ' << &(v2[0]) << '\n';
// To use existing (C/Fortran) libraries, we can access the underlying c-style array
with .data()
std::cout << v2.data() << ' ' << &(v2[0]) << '\n';

// Ranged-based for loops: easiest way to access elements
for (double element : v2) {
    std::cout << element << '\n';
}
// same as:
for (int i = 0; i < v2.size(); ++i) {
    std::cout << v2.at(i) << '\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)
- See several more interactive examples at ComilerExplorer website using this link: **https://godbolt.org/z/8a796Ev1o**
- There is also std::array - 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

Some refresher tips of using std::vector, including using vector-of-vector.

For loop examples
Interactive version: https://godbolt.org/z/5csjznTfc

```
#include <iostream>
#include <vector>
int main() {
  std::vector<int> v{3, 1, 4};

  std::cout << "Regular for loop:\n";
  for (int i = 0; i < v.size(); ++i) {
    std::cout << v.at(i) << "\n";
```

```
  }
  std::cout << "Ranged for loop:\n";
  for (auto x : v) {
    std::cout << x << "\n";
  }
  std::cout << "Ranged for loop, by reference:\n";
  for (auto &x : v) {
    std::cout << x << "\n";
    x *= 2;  // by reference, can modify values
  }
  std::cout << "Ranged for loop, after modifying:\n";
  for (auto x : v) {
    std::cout << x << "\n";
    x *= 2;  // by reference, can modify values
  }
}
```

---

std::vector of std::vector

std::vector is a c++ *container* - and can hold values of *any* type, including another vector. A vector-of-vector is therefore, in a sense, a 2D vector.

NOTE however, that a vector-of-vector does not store all the memory contiguously in a single block. Each internal vector stores all the elements in a single block, but each vector is stored in a different location. That is why we cannot use a vector-of-vector in place of regular C-array to pass matrix to LAPACK.

However, vector-of-vector is still very useful in many situations, since it lets us easily pass individual internal vectors around to different functions. For example, you may store a set of wavefunctions in a vector-of-vector-of-double, then you can pass individual wavefunctions around, which is very nice.

Example for vector-of-vector:
Interactive version: https://godbolt.org/z/7v67obYef

```
#include <iostream>
#include <vector>

// Simple function; prints elements of a vector of int
void printVector(const std::vector<int>& v) {
  for (auto x : v) {
    std::cout << x << ", ";
  }
  std::cout << '\n';
}

int main() {
  // Create a vector of 3 empty vectors
  std::vector<std::vector<int>> v(3);
  std::cout << v.at(0).size() << "\n";
  std::cout << v.at(1).size() << "\n";
```

```
  std::cout << v.at(2).size() << "\n";

  // Nb: Can use ranged for loop too!
  for (const auto& x : v) {
    // 'auto' evaluiates to 'std::vector<int>' here
    // By reference, avoid copying entire vector
    std::cout << x.size() << "\n";
  }

  // Can fill internal vectors
  v.at(0).push_back(3);
  v.at(0).push_back(2);
  v.at(0).push_back(1);

  // Or set them equal to new vectors
  v.at(1) = {6, 7, 8, 9, 10};

  // Can send each internal vector to function!
  printVector(v.at(1));
}
```

---

std::array

- The standard c++ fixed-size (statically allocated) array
- Works similar to vector, but size must be known at compile time
- Slight performance improvement in some cases over vector (but less versatile)

```
  std::cout << "\nstd array: a fixed-sized array of contiguous memory\n";
  // We must tell the compiler the size of the array, at compile time
  std::array<int, 5> array1{1, 2, 3, 4, 5};
  // std::array array2{1, 2, 3, 4, 5}; //with c++17, types can be deduced
  for (auto element : array1) {
    std::cout << element << "\n";
  }
```

---

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';
```

Operator overloads

You can make parts of your code much simpler to read/write by defining *operator overloads* for std::vector.

Consider the below example, which provides a '+' operator:

Example
Interactive version:https://godbolt.org/z/d8GP9PWsf

```
#include <cassert>
#include <iostream>
#include <vector>

// Overload for '+'
std::vector<double> operator+(std::vector<double> &a,
                              const std::vector<double> &b) {
  assert(a.size() == b.size() && "a and b must be same size");
  for (auto i = 0ul; i < b.size(); ++i) {
    a.at(i) += b.at(i);
  }
  return a;
}

// Simple function; prints elements of a vector of double
void printVector(const std::vector<double> &v) {
  for (auto x : v) {
    std::cout << x << ", ";
  }
  std::cout << '\n';
}

int main() {
  std::vector<double> v1{3.0, 3.5, 4.0};
  std::vector<double> v2{0.1, 0.2, 0.3};

  printVector(v1);
  printVector(v2);
  printVector(v1 + v2); //much easier than doing slow way!
```

```
}
```

NOTE it can be considered bad practice to overload operators for built-in types (like std::vector) - since it may conflict with other parts of a code base. For this reason, it is encouraged to wrap these overloads in a namespace. Then, use the `using namespace .....` directive to provide *local* access to the operators only within a small scope, reducing risk of conflicts (hardly important for small projects)

For example, should '+' add together each element of a vector like {a+x, b+y, c+z}? Or join two vectors, like {a,b,c,x,y,z}? What if you want different behaviour in different cases? What if you and a co-worker disagree? In fact, this is the reason these are not already define as standard in c++_

Example

```
namespace MyVectorOverloads {
std::vector<double> operator+(std::vector<double> &a, const std::vector<double> &b)
{...}
}

int main() {
  using namespace MyVectorOverloads; //put this in as narrow a scope as practicle!
  std::vector<double> v1{3.0, 3.5, 4.0};
  std::vector<double> v2{0.1, 0.2, 0.3};
  printVector(v1 + v2);
}
```