COMP6771 19T2 (https://webcms3.cse.unsw.edu.au/COMP6771/19T2)

Week 05 Tutorial Sample Answers

Advanced C++ Programming (https://webcms3.cse.unsw.edu.au/COMP6771/19T2)

1. Are the following lines constructions or assignments?
   1. `std::vector<int> a(1, 2, 3)`

      Constructor (direct initilization). We prefer not to do this.

   2. `std::vector<int> a{1, 2, 3}`

      Constructor (uniform initilization)

   3. `std::vector<int> b = {1, 2, 3}`

      Constructor (uniform initilization)

   4. `std::vector<int> c = a`

      Constructor (copy construction)

   5. `c = b`

      Assignment (copy assignment)

   6. `a = std::move(c)`

      Assignment (move assignment)

   7. `std::vector<int> d{std::move(a)}`

      Constructor (move constructor)

2.
```
#include <cassert>
#include <iostream>

class Point {
  public:
    Point(int x, int y) : x_(x), y_(y) {}
    int& operator[](int i) {
      assert(i == 0 || i == 1);
      if (i == 0) return this->x_;
      else return this->y_;
    }

  private:
    int x_;
    int y_;
};

int main() {
  Point q{99, -5};
  q[0] = -99;
  std::cout << q[0] << "\n";

  const Point p{99, -5};
  std::cout << p[0] << "\n";
}
```

Make this code const-correct so it can compile and run sucessfully

In what cases would we need to overload an operator for its const or non-const version?

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

class Point {
  public:
    Point(int x, int y) : x_(x), y_(y) {}
    int& operator[](int i) {
      assert(i == 0 || i == 1);
      if (i == 0) return this->x_;
      else return this->y_;
    }
    int operator[](int i) const {
      assert(i == 0 || i == 1);
      if (i == 0) return this->x_;
      else return this->y_;
    }

  private:
    int x_;
    int y_;
};

int main() {
  Point q{99, -5};
  q[0] = -99;
  std::cout << q[0] << "\n";

  const Point p{99, -5};
  std::cout << p[0] << "\n";
}
```

3. Why is prefix increment/decrement usually preferred to postfix increment/decrement? When would we use postfix?

    - Prefix is generally preferred (when no copy is needed) due to one less step (a copy and return) being completed. Therefore at scale this operation will yield finite performance improvements
    - Postfix is used when you want to receive a copy of the object's current state prior to the increment/decrement occuring

4.
```
// Point.h:
#include <vector>
class Point {
  public:
    Point(int x, int y) : x_(x), y_(y) {}


  private:
    int x_;
    int y_;
};

// Point.cpp:
#include <iostream>
#include <vector>
int main() {
  Point p{1,2};
  double length = p;
}
```

Modify this code to have a non-explicit type overload for a double that returns the length from the origin to the point's current coordinates.

```
// Point.h:
#include <vector>
#include <cmath>
class Point {
  public:
    Point(int x, int y) : x_(x), y_(y) {}
    operator double() {
      return sqrt(x_*x_ + y_*y_);
    }
  private:
    int x_;
    int y_;
};

// Point.cpp:
#include <iostream>
#include <vector>
int main() {
  Point p{1,2};
  double length = p;
}
```

5. Why do we use smart pointers instead of raw pointers whenever possible?

   Smart pointers mean:
   - We do not have to worry about deleting our heap memory
   - Move and copy semantics are handled for us

6. What is an example of a circumstance a std::shared_ptr<T> would be preferred over a std::unique_ptr<T>?

   In general, you can have it when either:
   - There is shared ownership (very rare, it's usually one owner and many observers).
   - There is a single owner, but the observers may stick around longer than the owner.

   When we have ownership loops (eg. a graph that isn't a dag), we have one of two options:
   - Preferred: have something like std::vector<std::unique_ptr<Node>>, and each node stores raw pointers to other nodes. This may not be possible if it's hard to find all the raw pointers and remove them when you delete the unique pointer.
   - Otherwise: Have something like std::vector<std::shared_ptr<Node>>, and each node stores weak pointers to other nodes (DO NOT store shared pointers in each node - the nodes will never get removed)

7.