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

Week 00 - Tutorial - Sample Answers -

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

1. **Section 1 Practice** Which part of this code is either a compile error, or bad style?

```
#include <iostream>
#include <string>

class Lemon {
 public:
  explicit Lemon(const int r, std::string s) : sourness_{s}, radius_{r} {}
  friend std::ostream& operator<<(std::ostream& os, const Lemon& l) { os << l.radius_; return os; }
 private:
  int radius_;
  std::string sourness_;
};

int main() {
  Lemon l(3, "Very sour!");
  std::cout << l << "\n";
}
```

   A. A non-const member "radius_" cannot be initialised with a const parameter "r" in the constructor
   B. The initialiser list is not in the correct order
   C. The output operator for Lemon only takes in const Lemons, but we try and use it on a non-const Lemon object.
   D. Lemon cannot be constructed this way when the associated constructor is marked explicit

   B

2. **Section 1 Practice** Which of the following are NOT valid function overloads?

   A.
```
bool foo(int a, int b);
bool foo(double a, double b);
```
   B.
```
bool foo(int a, int b);
double foo(int c, int d);
```
   C.
```
bool foo(int a, int b);
double foo(int c, double d);
```
   D.
```
int foo(char* a, char* b);
int foo(std::string a, std::string b);
```

   B - can't distinguish by return type

3. **Section 1 Practice** Define "iterator" in the context of C++ STL
   A. The operator ++ used as either pre-increment or post-increment
   B. A variable used in for-loops (typically called "i") used to keep track of the index of an item during a loop iteration
   C. An abstraction of a pointer that represents a container in a linear form
   D. A pair of functions (usually begin() and end()) used by a container

   C

4. **Section 1 Practice** Which of the following is NOT a major reason why we prefer references over pointers in many instances?
   A. References often require less syntatic changes (e.g. no explicit deferencing require to get value) than pointers
   B. References cannot be set to null, unlike pointers
   C. References use less memory than pointers

   C - references do not use less memory than pointers

5. **Section 1 Practice** Observe the following code

```
void foo(const int& a) { std::cout << a; }
int i = 3;
foo(5); // usage 1
foo(i); // usage 2

int j = std::move(i);
foo(j); // usage 3
```

In each of the following 3 usages, is the call argument an lvalue or an value?

A. Usage1=rvalue, Usage2=lvalue, Usage3=lvalue
B. Usage1=lvalue, Usage2=lvalue, Usage3=lvalue
C. Usage1=lvalue, Usage2=rvalue, Usage3=rvalue
D. Usage1=rvalue, Usage2=rvalue, Usage3=lvalue
E. Usage1=rvalue, Usage2=rvalue, Usage3=rvalue

A

6. **Section 1 Practice**

```
class Foo {
 public:
  Foo() : Foo(5) {}
  Foo(int var) : var_{var} {}
 private:
  int var_;
}
```

What is the term we use to refer to the function Foo() in class Foo?

A. Delegating Constructor
B. Delegating Assignment
C. Polymorphic Constructor
D. Polymorphic Assignment

A

7. **Section 2 Practice** (1 mark) What is static polymorphism (i.e. templates)? Why do we use them/it?

Static polymorphism / templates are a form of generic programming that allow us to define types and operations that provide functionality independent of the the underlying type they operate on. They allow us to provide the same capability for a range of "generic" types without redefining code. (Note, a typical answer may be around 50-75% the length of this one. This is just longer to provide the full explanation).

8. **Section 2 Practice** (2 mark) What are the 3 components of the STL? What is a benefit of using the STL?

(1). Iterators, Containers, Algorithms. (2). Containers not needing to know about algorithms; algorithms not needing to know about iterators

9. **Section 3 Practice (2 marks)**

```
template <typename T>
auto createDefaultArray(int size) {
  return std::make_unique<T[]>(size);
}
```

Add a non-type parameter for "size" such that the createDefaultArray function is now templated across two template parameters (one typed, one non-typed).

```
template <typename T, int size>
auto createDefaultArray() {
  return std::make_unique<T[]>(size);
}
```

10. **Section 3 Practice (2 marks)**

```
class Magnitude {
 public:
  Magnitude() : Magnitude(1) {}
  Magnitude(int l) : litres_{l} {}
  static Magnitude multiply(const Magnitude& a, const Magnitude& b) {
    return Magnitude{a.getLitres() * b.getLitres()};
  }
  int getLitres() {
    return litres_;
  }
 private:
  int litres_;
}

int main() {
  Magnitude a{8};
  Magnitude b{9};
  std::cout << Magnitude::multiply(a, b);
}
```

This code currently does not take advantage of operator overloading. Re-write the code with the appropriate multiplication operator. You can re-write the main function to demonstrate your knowledge of this. Your new code should work with the following main function.

```
int main() {
  Magnitude a{8};
  Magnitude b{9};
  std::cout << (a * b) << "\n";
}
```

```
class Magnitude {
 public:
  Magnitude() : Magnitude(1) {}
  Magnitude(int l) : litres_{l} {}
  friend Magnitude operator*(const Magnitude& a, const Magnitude& b) {
    return Magnitude{a.getLitres() * b.getLitres()};
  }
  friend std::ostream& operator<<(std::ostream& os, const Magnitude& m) {
    os << m.getLitres();
    return os;
  }
  int getLitres() const {
    return litres_;
  }
 private:
  int litres_;
};

int main() {
  Magnitude a{8};
  Magnitude b{9};
  std::cout << (a * b) << "\n";
}
```

11. **Section 3 Practice (4 marks)**

```
template <typname T>
class EuclideanVector {
 public:
  EuclideanVector() : EuclideanVector(10) {}
  EuclideanVector(int size) : vec_{new T[size]}, size_{size} {}
 private:
  T *vec_;
  int size_;
}
```

Implement the rule of 5 (copy and move semantics, and destructor)

```
template <typname T>
class EuclideanVector {
 public:
  EuclideanVector() : EuclideanVector(10) {}
  EuclideanVector(int size) : vec_{new T[size]}, size_{size} {
    for (int i = 0; i < size_; ++i) {
      vec_[i] = 0;
    }
  }
  EuclideanVector(connt EuclideanVector& old) {
    size_ = old.size_;
    vec_ = new T[size_];
    for (int i = 0; i < size_; ++i) {
      vec_[i] = old.vec_[i];
    }
  }
  EuclideanVector(EuclideanVector&& old) {
    size_ = std;:move(old.size_);
    vec_ = new T[size_];
    for (int i = 0; i < size_; ++i) {
      vec_[i] = old.vec_[i];
    }
    delete[] old.vec_;
  }
  EuclideanVector& operator=(const EuclideanVector& old) {
    size_ = old.size_;
    delete[] vec_;
    vec_ = new T[size_];
    for (int i = 0; i < size_; ++i) {
      vec_[i] = old.vec_[i];
    }
    return *this;
  }
  EuclideanVector& operator=(EuclideanVector&& old) {
    delete[] vec_;
    size_ = std::move(old.size_);
    vec_ = old.vec_;
    old.vec_ = nullptr;
    return *this;
  }
  ~EuclideanVector() {
    delete[] vec_;
  }
 private:
  T *vec_;
  int size_;
}
```

12. **Section 3 Practice (3 marks)**

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

int main() {
  std::vector<std::string> vec;
  vec.push_back("Hayden");
  vec.push_back("is");
  vec.push_back("made");
  vec.push_back("of");
  vec.push_back("pineapple");
  int sum = 0;

  std::cout << "Sum: " << sum << "\n";
}
```

Complete this function using STL algorithms and C++11 lambdas to find the sum of all lowercase vowels in all of the strings inside the vector "vec". You will be marked on appropriate use of STL algorithms.

```cpp
#include <iostream>
#include <vector>
#include <string>
#include <numeric>

int main() {
  std::vector<std::string> vec;
  vec.push_back("Hayden");
  vec.push_back("is");
  vec.push_back("made");
  vec.push_back("of");
  vec.push_back("pineapple");
  int sum = 0;

  sum = std::accumulate(vec.begin(), vec.end(), 0, [] (auto sum, const std::string& item) {
        int num_vowels = std::accumulate(item.begin(), item.end(), 0, [] (auto sum2, const char i) {
          return (i == 'a' || i == 'o' || i == 'e' || i == 'u' || i == 'i') ? sum2 + 1 : sum2;
        });
        return sum + num_vowels;
  });

  std::cout << "Sum: " << sum << "\n";
}

// Note: there may be better answers than this, but this one is acceptable
```