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

Week 04 Tutorial Sample Answers

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

1.
```
#ifndef TUTORIALS_WEEK3_CAR_H_
#define TUTORIALS_WEEK3_CAR_H_

#include <string>

class Car {
 public:
  // TODO(tutorial): fill me in

 private:
  std::string manufacturer_;
  int num_seats_;
};

#endif  // TUTORIALS_WEEK3_CAR_H_
```

```
#include "tutorials/week3/car.h"

// TODO(tutorial): fill me in
```

1. **Open tutorials/week3/car.(cpp/h) (you should have started on this last week):**
2. Create a copy constructor, should the object count change?
3. Create a move constructor, should the object count change?
4. Create a copy assignment operator, should the object count change?
5. Create a move assignment operator, should the object count change?
6. What is the difference between a copy constructor, copy assignment operator, move constructor and move assignment operator?

   Copy and move constructors both create a new object. Keep in mind that for move constructors, the moved-from object still exists, and so you have an extra object. When that moved-from object is destructed is when the object count should go back down. Hence, both constructors increase the object count.
   Assignments simply overwrite one object with the contents of another. Hence, they don't change the object count.
   The difference between a constructor and an assignment operator is that a constructor creates an object from nothing, while an assignment operator overwrites an object with the content from an existing one (or in other words, the difference is that the target you want to fill with data already contains some valid data)
   The difference between copy and move is that move calls the T&& overload of the function. This is usually a faster implementation, which might do something such as steal memory from the moved-from object instead of allocating more memory.

7. Run your code against //tutorials/week3:car_main

   The output should be:

```
2
3
4
5
5
5
```

```
#ifndef TUTORIALS_WEEK3_CAR_H_
#define TUTORIALS_WEEK3_CAR_H_

#include <string>

class Car {
 public:
  Car(): Car{"unknown", 4} {}
  Car(const std::string& manufacturer, int n_seats): manufacturer_{manufacturer}, num_seats_{n_sea
    ++n_objects;
  }

  Car(const Car& other);
  Car(Car&& other) noexcept;
  Car& operator=(const Car&) = default;
  Car& operator=(Car&&) = default;


  ~Car() noexcept;

  const std::string& GetManufacturer() const;
  int GetNumSeats() const;

  static int GetNumCars();

 private:
  std::string manufacturer_;
  int num_seats_;

  static int n_objects;
};

#endif  // TUTORIALS_WEEK3_CAR_H_
```

```
#include "car.h"

int Car::n_objects = 0;

Car::Car(const Car& other): manufacturer_{other.manufacturer_}, num_seats_{other.num_seats_} {
  n_objects++;
}

Car::Car(Car&& other) noexcept: manufacturer_{std::move(other.manufacturer_)}, num_seats_{other.nu
  n_objects++;
}

Car::~Car() noexcept {
  --n_objects;
}

const std::string& Car::GetManufacturer() const {
  return manufacturer_;
}

int Car::GetNumSeats() const {
  return num_seats_;
}

int Car::GetNumCars() {
  return n_objects;
}
```

2. Complete the istream operator overload from lectures

```
// Point.h:
#include <ostream>
#include <istream>
class Point {
  public:
    Point(int x, int y) : x_{x}, y_{y} {};
    friend std::ostream& operator<<(std::ostream& os, const Point& type);
    friend std::istream& operator>>(std::istream& is, Point& type);

  private:
    int x_;
    int y_;
};

// Point.cpp:
#include <ostream>
#include <istream>
#include <iostream>
std::ostream& operator<<(std::ostream& os, const Point& p) {
  os << "(" << p.x_ << "," << p.y_ << ")";
  return os;
}
std::istream& operator>>(std::istream& is, Point& type) {
  // To be done in tutorials
}

int main() {
  Point p{1,2};
  std::cout << p << '\n';
}
```

```
std::istream& operator>>(std::istream& is, Point& p) {
  return is >> p.x_ >> p.y_;
}
```

3. Why would this be considered not a very wise decision to provide this overload?

```
Point& operator*=(const Point& p);
```

```
// Point.h:
class Point {
  public:
    Point& operator+=(const Point& p);
    Point& operator-=(const Point& p);
    Point& operator*=(const Point& p);

  private:
    int x_;
    int y_;
};

// Point.cpp:
Point& Point::operator*=(const Point& p) {
  this->x_ *= p.x_;
  this->y_ *= p.y_;
  return *this;
}
```

The semantic of multiplying two vectors together is ambiguous. It could be "multiplied" a few different ways (dot product, lexigraphically). We only use operator overloading when the semantic is clear and intuitive.

4. Why are some operator overloads done as member functions, and others as friends?

- Non-members: For when the operator doesn't take an instance of the class as the LHS argument
  - If we need this non-member function to access private fields of the class type, we define this non-member overload as a friend
- Members: For when the operator does take an instance of the class as the LHS argument

5.

```
#include <string>

class Book {
 public:
  int GetIsbn() const { return isbn_; }
  double GetPrice() const  { return price_; }

 private:
  std::string name_;
  std::string author_;
  int isbn_;
  double price_;
};
```

1. Write the function definition for a constructor that takes values for name, author, isbn and price and uses a member initializer list to set the value of each data member.
2. Write the overloaded == operator to compare if two book objects are identical
3. Write the overloaded != operator that calls the already overloaded == operator to return true when two book objects are not identical.
4. Write the overloaded << operator to print out the name, author, isbn and price of a book using std::cout
5. Write the overloaded type conversion operator to enable the conversion of the Book class to a std::string in the form of "author, name"
6. Create a main function that creates a std::vector<book>, add a few Book objects into the vector with different isbn numbers and prices
7. Write the overloaded < operator to allow you to sort books by their isbn number. Test this in your main method using std::sort
8. Call std::sort again with a lambda function as the predicate that sorts the books by price.

```
#include "tutorials/week4/book.h"

#include <algorithm>
#include <iostream>
#include <vector>

// Helper function to print vectors of books.
void print(const std::string& title, const std::vector<Book>& books) {
  std::cout << title << '\n';
  for (const auto& b: books) {
    std::cout << "\t";
    std::cout << b << '\n';
  }
  std::cout << '\n';
}

int main() {
  std::vector<Book> books1{
      Book{"Book1", "Author1", 2222, 12.2},
      Book{"Book2", "Author2", 1111, 11.50},
      Book{"Book3", "Author3", 3333, 10.50}};

  std::vector<Book> books2{
      Book{"Book1", "Author1", 2222, 12.2},
      Book{"Book2", "Author2", 1111, 11.50},
      Book{"Book3", "Author3", 3333, 10.50}};

  print("Books:", books1);

  // Call the std::vector<Book> operator==() overload which in
  // turn calls Book operator==() overload.
  std::cout << "Vector of books are equal: "
            << "books1 " << (books1 == books2 ? "==" : "!=")
            << " books2\n\n";

  std::sort(books1.begin(), books1.end());
  print("Default (ISBN) sorted books:", books1);

  std::cout << "Vector of books are no longer equal: "
            << "books1 " << (books1 == books2 ? "==" : "!=")
            << " books2\n\n";

  std::sort(books1.begin(), books1.end(),
            [](const Book& a, const Book& b) {
              return a.getPrice() < b.getPrice();
            }
  );

  print("Price sorted books:", books1);
}
```

```
#ifndef TUTORIALS_WEEK4_BOOK_H_
#define TUTORIALS_WEEK4_BOOK_H_

#include <string>
#include <ostream>

class Book {
 public:
  Book(const std::string& name, const std::string& author, int isbn, double price):
      name_{name}, author_{author}, isbn_{isbn}, price_{price} {}

  // Since these are just getters, we put them in the header file.
  int getIsbn() const { return isbn_; }
  double getPrice() const { return price_; }

  // Type conversion operators
  operator std::string() const;

  // Friend comparison operators. Note that we always declare these inline.
  friend bool operator==(const Book& a, const Book& b) {
    return (a.name_ == b.name_) && (a.author_ == b.author_) &&
        (a.isbn_ == b.isbn_) && (a.price_ == b.price_);
  }

  friend bool operator!=(const Book& lhs, const Book& rhs) {
    return !(lhs == rhs);
  }

  friend bool operator<(const Book& a, const Book& b) {
    return a.isbn_ < b.isbn_;
  }

  friend std::ostream& operator<<(std::ostream& out, const Book& b) {
    return out << "Name: \"" << b.name_ << "\", "
               << "(Author: \")" << b.author_ << "(\", )"
               << "ISBN: " << b.isbn_ << ", "
               << "Price: " << b.price_;
  }

 private:
  std::string name_;
  std::string author_;
  int isbn_;
  double price_;
};

#endif // TUTORIALS_WEEK4_BOOK_H_
```

```
#include "tutorials/week4/book.h"

#include <ostream>

Book::operator std::string () const {
  return author_ + "," + name_;

  // Potentially more flexible.
  /*
  std::stringstream ss;
  ss << author_ << "," << name_;
  return ss.str();
  */
}
```