Programming Skills: C++ **Homework 3: C++11**

Due Tues 4/23 (start of class)

Name: _____

1. Draw a memory diagram of all the variables, their associated values, and allocated space in use at the end of this code snippet (all variables stay in scope). Make sure to distinguish between the stack and the heap.

```
shared_ptr<string> ptrA{new string{"one"}};
shared_ptr<string> ptrB{ptrA};
string s1{"two"};
string& s2 = s1;
s2 = "three";
ptrA.reset(new string{"four"});
unique_ptr<string> ptrC{new string{"five"}};
weak_ptr<string> ptrD = ptrB;
string* ptrE = ptrC.release();
```

2. A shop owner selling `Thing`s wants to maintain them in a collection with blazing fast insert and retrieve capability. Each `Thing` is uniquely identified by an id (a long). Sometimes, other `Thing`s will come in to the shop that are tagged with the same id, but may have a different description and/or price. In this instance, the shop owner does not want them added to their collection.

```
struct Thing {
    string desc_;
    long id_;
    double price_;

    Thing(const string &desc, long id, double price) :
            desc_{desc}, id_{id}, price_{price} {}

    bool operator==(const Thing &t) const { return id_ == t.id_; }
};

int main() {
    unordered_set<Thing, function<size_t(const Thing &t)>>
            things{10, [](const Thing &t) {
                return hash<string>()(t.desc_) +
                       hash<long>()(t.id_) +
                       hash<double>()(t.price_);
            }
    };
    things.emplace("Toy", 1095, 11.99);
    things.emplace("Device", 4103, 33.15);
    things.emplace("Widget", 109, 18.00);
    things.emplace("Weapon", 1095, 95.19);
    //...
};
```

(a) What `Thing` is getting added to the collection that shouldn't be?

(b) Briefly explain what is wrong with the code.

(c) Modify the code so that it behaves correctly.

3. Rewrite the call to `sort` to use a lambda expression that sorts the `Students` by increasing GPAs, instead of using the `StudentCompare` function object.

```
struct Student {
    string name_;
    double gpa_;
    Student(const string& name, double gpa) :
        name_{name}, gpa_{gpa} {}
};

class StudentCompare {
public:
    bool operator()(const Student& a, const Student& b) const {
        return a.gpa_ > b.gpa_;
    }
};

int main() {
    array<Student, 4> students {
        Student{"Tom", 3.5},
        Student{"Mary", 3.66},
        Student{"Brad", 2.6},
        Student{"Amy", 3.1}
    };
    sort(students.begin(), students.end(), StudentCompare());
    //...
}
```

4. Explain what, if anything, is wrong with each of the following code snippets, from a memory management perspective. Assume all variables go out of scope after the last statement.

   (a)
   ```
   int* ptrA = new int(10);
   unique_ptr<int> ptrB{ptrA};
   delete ptrA;
   ```

   (b)
   ```
   int x{10};
   shared_ptr<int> ptrA{&x};
   ```

   (c)
   ```
   unique_ptr<int[]> ptrA{new int[20]};
   int* ptrB = ptrA.release();
   delete ptrB;
   ```

   (d)
   ```
   unique_ptr<int> ptrA{new int(10)};
   unique_ptr<int> ptrB{ptrA.get()};
   ```

5. Write the move constructor and move assignment for the `Buffer` class. It should transfer all ownership to the new instance and leave the moved object in a valid, default state.

```
template <typename T>
class Buffer {
     std::string name_;
     size_t size_;
     std::unique_ptr<T[]> buffer_;
public:
     Buffer(const std::string& name, size_t size):
          name_{name},
          size_{size},
          buffer_{new T[size]} {}
```

6. Briefly answer each of the following questions.
   (a) What can you say about the values for the local variables of `f()`?

```
void f() {
    int a;
    int b{};
    //...
}
```

   (b) Why does the following code not increment each value in the map by one?

```
map<string, int> names {
    {"tom", 10},
    {"harry", 4},
    {"mary", 14},
};
for (auto kv : names) {
    kv.second += 1;
}
```

   (c) What is the formal name for the container that is passed to `vector`'s constructor when this line of code is run?

```
vector v1{1, 2, 3, 4};
```

7. A developer has come up with this thread program. It is supposed to create five threads, each that are responsible for printing out 5 characters. The threads should begin by entering a queue, and are only allowed to run and print the characters if they are at the front of the queue, and there are no more than 2 threads currently printing.

```
deque<int> q;
mutex print_mutex;
mutex queue_mutex;
condition_variable queue_cond;

void begin(int num) {
    unique_lock<mutex> ul{queue_mutex};
    q.emplace_back(num);
    queue_cond.wait(ul, [num]{
        return q.front() == num; });
    q.pop_front();
    cout << num << "␣leaves␣begin" << endl;
}

void end(int num) {
    lock_guard<mutex> lg{queue_mutex};
    queue_cond.notify_all();
    cout << num << "␣has␣ended" << endl;
}

void run(int num, char ch) {
    begin(num);
    for (int i=0; i<5; ++i) {
        {
            lock_guard<mutex> lg{print_mutex};
            cout << ch << endl << flush;
        }
        sleep_for(milliseconds(250));
    }
    end(num);
}

int main() {
    vector<thread> threads{};

    for (int i=0; i<5; ++i) {
        threads.push_back(thread{run, i, static_cast<char>(65+i)});
    }
}
```

There are several problems with this implementation. The first issue is that it crashes very quickly when run, e.g.:

```
0 leaves begin
A
1 leaves begin
libc++abi.dylib: terminating
```

Fix all the problems so the code runs how it is expected to.

8. What is the output of the following code that uses several Boost algorithms?

```
#include <boost/range/combine.hpp>
#include <boost/foreach.hpp>
#include <iostream>
#include <vector>
#include <list>

int main(int, const char*[]) {
    std::vector<int> v;
    std::list<char> l;
    for (int i = 0; i < 5; ++i) {
        v.push_back(i);
        l.push_back(static_cast<char>(i) + 'a');
    }

    int ti;
    char tc;
    BOOST_FOREACH(boost::tie(ti, tc), boost::combine(v, l)) {
        std::cout << '(' << ti << ',' << tc << ')' << '\n';
    }

    return 0;
}
```