# CS246E Final Exam Review Package

Fall 2017

**Disclaimer: This is a review package intended to aid studying for the final exam. It is not a past final exam, was designed without knowledge of the actual final exam, and the course staff make no guarantee that it is representative of the final exam in terms of length or difficulty.**

## Question 1: Misc. Short Answer (10 points)

1. (4 points) Consider the following class definitions:

```
class C {
  public:
    virtual void f() = 0;
    virtual ~C() = 0;
};

class D: public C {
  public:
    virtual void f() override { /* ... */ }
};

class E: public C {
  public:
    ~E() { /* ... */ }
};

class F: public C {
  public:
    virtual void f() override {/* ... */}
    ~F() { /* ... */ }
};
```

   List the classes that are abstract below.

2. (2 points) Suppose `T` is a template argument to a function. Explain why `T&&` may not be an rvalue reference to a `T` and suggest an alternative. You may use the STL.

3. (4 points) Explain what coupling and cohesion are and why we want high coupling and low cohesion.

4. (2 points) Consider the following implementation of `move`:

```
template <typename T>
inline T &&std::move(T &t) {
  return static_cast<T&&>(t);
}
```

Explain why this version of `move` may not correctly convert lvalues to rvalues as expected.

5. (2 points) Give one advantage and one disadvantage of virtual inheritance.

6. (2 points) `reinterpret_cast<T*>(p)` simply changes the type of `p` to a `T*` and does not translate to any actual runtime code. Give a circumstance where this may behave differently than `static_cast<T*>(p)` despite both compiling.

## Question 2: Reducing Copying (7 points)

Consider the following snippet from a data structure:

```
template<typename T>
class queue {
  T *ts_;
  size_t end_;
  /* ... */
  public:
    /* You may assume appropriate copy, move, default, and initializer_list ctors
       exist, as well as a destructor, copy and move assignment operators, and
       the increaseCap function below. Make no other assumptions. */
    void push_back(const T &t) {
      increaseCap();
      new (ts_ + end_) T{t};
      end_++;
    }
};
```

1. (1 point) Give an example of a use of `queue` which is inefficient due to the way the code above is written.

2. (3 points) Add an extra method to the queue class to improve the runtime of your example. You may assume the method is written within the body of public section of `queue`.

3. (3 points) Reduce code duplication by rewriting the appropriate method in `queue` so that the method you defined in part (b) is not needed.

## Question 3: SFINAE (8 points)

1. (4 points) Rewrite the following code to fix any compile errors.

```
template <typename T>
T create_T() {
  if (std::is_default_constructible<T>::value) {
    return T{};
  } else {
    return T{1};
  }
}
```

2. (4 points) Add any code necessary to make the code below correctly print "true" then "false".

```
#include <iostream>
#include <iomanip>

template <typename...> using ignore_args = void;

template <typename, typename T>
struct is_default_constructible_helper {
  static const bool value = false;
};

//Beginning of extra code









//End of extra code

template <typename T>
struct is_default_constructible
       : public is_default_constructible_helper<ignore_args<>, T> {};

class N {
  public:
    N(int i) {}
};

int main() {
  std::cout << std::boolalpha;
  std::cout << is_default_constructible<int>::value << std::endl;
  std::cout << is_default_constructible<N>::value << std::endl;
  return 0;
}
```

## Question 4: CRTP (6 points)

1. (2 points) We saw in class that you can modify the template method pattern to avoid using virtual methods via the CRTP. Explain why this does not make virtual methods in the template method pattern obsolete.

2. (4 points) Design a templated interface called "Assignable" which provides a non-virtual unified assignment operator to any class that inherits from it, such that the assignment operator takes a class of the same type as the inheriting class and returns that type as well (possibly with references involved in both cases). You may assume that the inheriting class provides a swap method with the usual behaviour.

## Question 5: Design Patterns (10 points)

Suppose we have a mathematical expression class hierarchy looking roughly like this (most methods are not included, but you may assume correct constructors and destructors exist):

```
class Expression { //Abstract
};

class Plus: public Expression {
  Expression *left, *right;
  public:
    Expression *getLeft() const { return left; }
    Expression *getRight() const { return right; }
};

class Variable: public Expression {
  std::string name;
  public:
    const std::string &getName() const { return name; }
};

//More classes
```

For our purposes we will assume only the pictured classes exist in our hierarchy. Suppose we want to evaluate these expressions. Naively, we might simply add a method called evaluate() to each class which evaluates the result of the expression. However, as we add more and more such functions with more and more complicated logic, this gets unwieldy: for example, tracking down a bug in the evaluate logic requires us to look through a number of source files, most of the code in each of which is completely unrelated to evaluating expressions. We want to refactor all the evaluation logic into one class. Design an abstract class called "Traversal", implement any new methods necessary in the Expression classes, and give a concrete subclass called "EvaluateTraversal" which can be used to evaluate expressions. You may use the following code as a starting point for EvaluateTraversal and do not need to rewrite it in your solution:

```
class EvaluateTraversal: public Traversal {
  std::map<std::string, int> varValues;
  public:
    EvaluateTraversal(std::initializer_list<std::pair<const std::string, int>> vals)
                      :varValues{vals} {}
};
```

## Question 6: Object-Oriented Design (6 points)

1. (3 points) Alice and Bob are designing a flight simulator. Being good object-oriented citizens, they've learned to make sure their classes are under interfaces, so they decide to provide an interface called "AbstractAirplane" which provides pure virtual methods for tasks like flight and driving. Later on, they want to extend their simulator to support other kinds of flying vehicles, such as hot air balloons and zeppelins. Explain why their idea constitutes bad OO design and suggest an alternative.

2. (3 points) Alice and Bob's flight simulator was a success and they've decided the next version will also feature a truck simulator. They decide that since they have already implemented all of the driving logic for land-based vehicles as part of their concrete Airplane class, they should make Truck extend Airplane and simply provide empty implementations of flight-related methods. Explain why their idea constitutes bad OO design and suggest an alternative.

## Question 7: Template Specialization (9 points)

Consider the following implementation of a barebones version of `unique_ptr`:

```
template <typename T>
class unique_ptr {
  T *p;
  public:
    unique_ptr():p{nullptr} {}
    unique_ptr(T *p):p{p} {}

    unique_ptr(const unique_ptr<T>&) = delete;
    unique_ptr(unique_ptr<T> &&o):p{o.p} { o.p = nullptr; }

    unique_ptr<T> &operator=(const unique_ptr<T> &) = delete;
    unique_ptr<T> &operator=(unique_ptr<T> &&o) { std::swap(p, o.p); }

    ~unique_ptr() { delete p; }

    T &operator*() { return *p; }
    T &operator[](int i) { return *(p+i); }
};
```

1. (3 points) Implement `make_unique`, a template function which takes a type `T` as a template argument and arguments to `T`'s constructor as arguments and produces a `unique_ptr<T>` to a newly constructed `T`.

2. (1 point) While vector is a better choice for managing an array with RAII, some users may require that `unique_ptr` supports arrays. Explain why this definition of `unique_ptr` doesn't support arrays.

3. (3 points) Write additional code so that `unique_ptr` can now be used with arrays. You may reuse code from the starter code without rewriting it as long as you make it clear where you are doing so.

4. (2 points) Write additional code so that `make_unique` for arrays.

## Question 8: Allocation (6 points)

Suppose we have a class called `C` and want to support the following allocation scheme:

- Enough memory for exactly one heap-allocated `C` object is allocated when the program begins.
- When a new C object is constructed via `new`, if the allocated memory is available then that is the memory that is returned. If the memory is already in use, an exception is thrown as if we had run out of memory.
- When a C object is deleted, its memory is made available for reuse by new.

1. (2 points) Provide a definition of `C` which includes any new fields or methods you want to define and their visibilities.

2. (2 points) Overload `new` to support the correct behaviour for `C` objects.

3. (2 points) Overload `delete` to support the correct behaviour for `C` objects.