Q1 What is size_t and when should it be used?

A1 size_t is an unsigned integer type used for indexing containers and holding the size of a container. It is guaranteed to be large enough to hold the size, and thus any index, of any container on the system on which the code was compiled. Note that the .size() function of containers in the standard library return a size_t, so to avoid loss of precision, one should always use a size_t when storing the size of a container.

Q2 In what situation would one use a struct rather than a class?

A2 structs are the same as classes, except structs are by default public, whereas classes are by default private. Normally, structs consist solely of related data, without member functions.

Q3 What is const correctness?

A3 const correctness refers to the following practices:

1. Making variables that do not change const
2. Making member functions of a class that do not modify the state of the class const.

Q4 How can we make the following class const correct?

```cpp
// This class divides a passed-in double by a member double
class DivideBy {
    public:
        DivideBy(double _by) : by(_by) {}
        double operator()(double x) {
            return x/by;
        }
    void reset() {
        by = 1;
    }
    double change(double x) {
        double old = by;
        by = x;
        return by;
    }
    private:
        double by;
};
```

A4
The class divides a passed-in double by a member double:

```cpp
class DivideBy {
public:
    DivideBy(const double &by) : by(by) {}
    double operator()(const double x) const {
        return x/by;
    }
    void reset() {
        by = 1;
    }
    double change(double x) {
        const double old = by;
        by = x;
        return by;
    }
private:
    double by;
};
```

Q5 What is a situation when it is better to pass by value than by const reference?

A5 One situation is when we need to create a copy of the passed parameter. For example, when implementing the binary `operator+`, `operator-`, `operator\`, and `operator*` we passed the first parameter by value. Another situation is when we are passing fundamental data types, because a reference takes up about the same amount of member as data of fundamental type.

Q6 What is the purpose of separating header files and cpp files?

A6 By placing the definitions of normal functions inside of `.cpp` files and placing the declarations inside of `.h` files, one is able to avoid redundant compiling of the function definitions for each translation unit that uses the functions. Specifically, instead of recompiling the entire function wherever it is used, we compile the function once, and then depend on the linker to find the correct function definition for a given function declaration.

Q7 Find memory leak(s) in the following code:

```cpp
struct Example {
    Example(const double &x) ptr(&x) {}
    Example(const Example& e) {
        ptr = new double(*e.ptr);
    }
    void changeVal(const double &x) {
        double* p = new double(&x);
        std::swap(ptr, p);
    }
    double* ptr;
};
```
A7 There are three memory leaks. First, in the copy constructor, which will have a memory leak
because the old data assigned to ptr is not deleted. Secondly, in the changeVal function, since we
do not delete p. Lastly, we do not provide a destructor, so the memory assigned to ptr will not be
deallocated.

Q8 Consider the following code. Does it have a memory leak?

```c++
struct Base {
    Base(const int _x = 0) : x(_x) {}
    virtual void print() {std::cout << x;}
    int x;
};

struct Derived : Base {
    Derived(const int _x, const double _y) : Base(_x), ptr(new double(_y)) {}
    void print() override {std::cout << _x << " , " << _y << std::endl;}
    ~Derived() {delete ptr;}
    int* ptr;
};
```

A8 Maybe. It is undefined behavior to call non-virtual destructors in a polymorphic class. It is
possible that the following code

```c++
Base* ptr = new Derived(1,1.5);
delete ptr;
```

will only call the destructor of Base, which will leak the data pointed to by ptr in Derived.

Q9 What is RAII?

A9 RAII stands for "resource acquisition is initialization." It describes the practice of having
memory acquisition occur when an object is created and memory deallocation occur when an
object is destroyed. By consequence, code that abides by RAII should not require the programmer
to use new, new[], delete, or delete[]; instead all allocation and deallocated should occur during
the constructor or destruction of objects. Smart pointers allow the programmer to abide by RAII
even when handling dynamically allocated memory.

Q10 Create a simple integer array class that contains as member variables a raw integer pointer, a
size_t for the size, and a size_t for the capacity. Create a default constructor, a constructor taking
in a size, create copy and move constructors, and create copy and move assignment operators.
Further, create member functions pop_back, push_back, and operator[], implemented as in the
std::vector<int> class. Overload operator[] on const. Upon construction, set capacity to be twice
the size. Keep your class free of memory leaks.

A10

```c++
class SimpleArray {
public:
    SimpleArray() : cap(1), sz(0), arr(new double[cap]()) {}
```
SimpleArray(const size_t _sz) : cap(_sz*2), sz(_sz), arr(new double[_sz]) {}

SimpleArray(const SimpleArray& e) : cap(e.cap), sz(e.sz),
ptr(new double[e.cap]) {
    for (size_t i = 0 ; i < sz ; ++i) {
        arr[i] = e.arr[i];
    }
}

// one option: unnecessary allocation but puts moved–from object
// in a nice (valid) state
SimpleArray(Simple&& e) : SimpleArray() {
    std::swap(cap, e.cap);
    std::swap(sz, e.sz);
    std::swap(arr, e.arr);
}

// copy–swap idiom: both move and copy assignment
// no need to check for self–assignment since a copy is made.
SimpleArray& operator=(SimpleArray e) {
    std::swap(cap, e.cap);
    std::swap(sz, e.sz);
    std::swap(arr, e.arr);
}

double& operator[] (const size_t i) {
    if (i >= sz) {
        throw std::out_of_range("index too large");
    }
    return arr[i];
}

double operator[] (const size_t i) const {
    if (i >= sz) {
        throw std::out_of_range("index too large");
    }
    return arr[i];
}

void push_back(const double i) {
    if (sz == cap) {
        cap *= 2;
        double* tmp = new double[cap];
        try {
            for (size_t i = 0 ; i < sz ; ++i) {
                tmp[i] = arr[i];
            }
        }
        std::swap(tmp, arr);
    }
catch (...) {
    delete [] tmp;
    throw;
}

delete [] tmp;

arr[sz++] = i;
}

void pop_back() {
    --sz;
}

private:
size_t cap;
size_t sz;

double* arr;

};

Q11 Add the specifiers final, virtual, const, override, constexpr, and static to the following class where appropriate.

class Animal {
public:
    Animal(std::string& _name) : name(_name) {++number_of_animals;}
    std::string get_name()
    std::string talk() = 0;
    unsigned get_number_of_animals();
private:
    std::string name;
    unsigned number_of_animals;
};

std::string Animal::get_name() {
    return name;
}

unsigned Animal::number_of_animals = 0;

unsigned Animal::get_number_of_animals() {
    return number_of_animals;
}

class Dog : public Animal {
public:
    Dog(std::string& _name) : Animal(_name) {++number_of_dogs;}
    unsigned get_number_of_dogs();
    std::string talk();
private:
    unsigned number_of_dogs;
    std::string noise;
};
Q12 Explain the value categories in C++. Give examples of each.
Expressions can be either pr-values, x-values, or l-values. These three categories are further placed into the categories of gl-values and r-values: pr-values and x-values are r-values, and x-values and l-values are gl-values. These categories are defined via two properties: having identity and being able to be moved-from. An expression has identity if it has a memory address and an expression can be moved-from if it is allowed to leave the moved-from object in an indeterminate, but valid, state.

r-values do not have identity but can be moved-from. x-values have identity and can be moved-from. l-values have identity and cannot be moved-from. The last option, does not have identity and cannot be moved-from, is not used in C++.

Examples:
1. `std::string x = "hello"; // "hello" is an l-value`
2. `std::string y(std::move(x)) // std::move(x) is an x-value`
3. `std::string z = y + " world"; // y + " world" is a pr-value`

Q13 Explain declarations and definitions. Will the code below compile? If not, indicate where the errors occur and/or how to fix them.

```cpp
1 class A {
2    B b;
3 }
4 class B {
5    A a;
6 }
7 extern int x;
8 class A;
9 std::string s;
10 int x = 10;
11 std::string s = "Hello world!";
```

A13 Declarations tell the compiler that a symbol is valid. Definitions give symbols meaning by associating symbols with code.

```cpp
1 class B; // need to declare B before using
2 class A {
3    B b;
4 }
5 class B {
6    A a;
7 }
8 extern int x;
9 class A;
10 std::string s;
11 int x = 10;
12 std::string s = "Hello world!"; // ERROR! Redefining s
```

Q14 Suppose we have two .cpp files as follows: 1.cpp
Will the code compile? If not, where do the error(s) occur?

A14 It is not allowed to have multiple definitions of the variable x. To fix this, one can use the `extern` keyword to declare `int x` in 1.cpp. Secondly, inline functions have internal linkage; thus, the linker will be unable to find the definition in 2.cpp for the declaration `int fun()`. To fix this, one can either remove the `inline` keyword, or place the full definition of `fun` in 2.cpp into 1.cpp.

Q15 What is the output of the following code? Will the code finish normally or be terminated prematurely?

```cpp
void fun1() {
    throw std::runtime_error("stack unwind");
    std::cout << "1" << std::endl;
}
void fun2() {
    try {
        fun1();
    } catch (const int x) {
        std::cerr << "2" << std::endl;
    } catch (const std::invalid_argument& e) {
        std::cerr << "3 " << e.what() << std::endl;
        throw;
    } catch (const std::runtime_error& e) {
        std::cerr << "4 " << e.what() << std::endl;
        throw;
    } catch (...) {
        std::cerr << "5" << std::endl;
        throw;
    }
    std::cout << "6" << std::endl;
}
int main() {
    try {
        fun2();
    } catch(const std::exception& e) {
        std::cerr << "7" << std::endl;
    } catch (const _exception_type& e) {
        std::cerr << "8" << std::endl;
    }
    return 0;
}
```
throw;

    }  

    std::cout << "8" << std::endl;

} 

A15 Output:

1 4 Stack unwind
2 5
3 7

The code will terminate early because the exception is uncaught.