Homework 04 Solutions

ECE 449/590, Fall 2022

1. *(30 points)* Consider the following piece of code. For each definition of the class `A`, determine if there will be any compiling error. If so, find the first line with the error and explain why.

   ```
   void some_function()
   {
       A a;       // line 1
       A b = a;   // line 2
       b = a;     // line 3
   }
   ```

   A.
   ```
   class A
   {
       const int member;
   };
   ```
   Answer: The compiler cannot generate `operator=` for `A` because no assignment can be done for the `const` member. So line 3 won't compile.

   B.
   ```
   class A
   {
       int member;
       ~A();
   };
   ```
   Answer: All lines 1, 2, 3 won't result in any compiling error. However, when `some_function` returns, the compiler won't be able to access the `private` destructor and will generate an error.

   C.
   ```
   class B
   {
       B &operator=(const B &);
   };
   class A
   {
       B member;
   ```

```
};
```

Answer: The compiler cannot generate `operator=` for `A` because it can't access `B`'s `private operator=`. So line 3 won't compile.

2. *(25 points)* Read the following program and decide what among the three members – default constructor, copy constructor, and `operator=`, should be synthesized or implemented in type `T` for the lines 1 to 5 to be compiled correctly. For example, the line 0 requires none of the three.

   Answer: See the comments in the code.

```
T create(size_t n);
bool condition_one(T t);
bool condition_two(const T &t);
void modify(T &t);

T generate(size_t n) {
    T a(1);                       // line 0: none
    T b = create(n);              // line 1: copy ctor

    while (!condition_one(b)) { // line 2: copy ctor
        modify(b);                // line 3: none
        if (condition_two(b))     // line 4: none
           break;
    }

    return b;                     // line 5: copy ctor
}
```

3. *(20 points)*

   Consider the following class definitions.

```
class base {
public:
    base(bool th) {
        std::cout << "ctor of base" << std::endl;
        if (th) {
            throw std::runtime_error("throw from ctor of base");
        }
    }
    ~base() {
        std::cout << "dtor of base" << std::endl;
    }
};

class member {
public:
    member(bool th) {
        std::cout << "ctor of member" << std::endl;
        if (th) {
            throw std::runtime_error("throw from ctor of member");
        }
    }
    ~member() {
        std::cout << "dtor of member" << std::endl;
    }
};

class derived : public base {
    member m_;
public:
    derived(bool base_th, bool member_th)
        : base(base_th), m_(member_th) {
        std::cout << "ctor of derived" << std::endl;
    }
    ~derived() {
        std::cout << "dtor of derived" << std::endl;
    }
};
```

Determine the outputs of the following functions and explain why.

1)
```
void test_1() {
    try {
        derived d(false, false);
    }
    catch (std::exception &e) {
        std::cout << e.what() << std::endl;
    }
}
```

Answer: There is no exception. `d` is constructed and destroyed as usual. Note that the base parts are constructed *before* members.

```
ctor of base
ctor of member
ctor of derived
dtor of derived
dtor of member
dtor of base
```

2)
```
void test_2() {
    try {
        derived d(false, true);
    }
    catch (std::exception &e) {
        std::cout << e.what() << std::endl;
    }
}
```

Answer: Only the `base` part of `d` is constructed.

```
ctor of base
ctor of member
dtor of base
throw from ctor of member
```

3)
```
void test_3() {
    try {
        derived d(true, false);
    }
    catch (std::exception &e) {
        std::cout << e.what() << std::endl;
    }
}
```

Answer: Nothing is constructed.

```
ctor of base
throw from ctor of base
```

4)
```
void test_4() {
    try {
        derived d(true, true);
    }
    catch (std::exception &e) {
        std::cout << e.what() << std::endl;
    }
}
```

Answer: Same as *4)* since the base parts are constructed *before* members.

```
ctor of base
throw from ctor of base
```

4. *(25 points)* Compile and execute `smart_pointer.cpp`. Explain the output.

Answer:

```
void test1() {
    shared_ptr<A> pa(new A); // (A, 1)         output: ctor of A
    shared_ptr<B> pb(new B); // (A, 1), (B, 1) output: ctor of B

    pa->b_of_A = pb;         // (A, 1), (B, 2)
    pb->a_of_B = pa;         // (A, 2), (B, 2)

    // pb is destroyed:         (A, 2), (B, 1)
    // pa is destroyed:         (A, 1), (B, 1)
}

void test2() {
    shared_ptr<A> pa(new A); // (A, 1)         output: ctor of A
    shared_ptr<B> pb(new B); // (A, 1), (B, 1) output: ctor of B

    pa->b_of_A = pb;         // (A, 1), (B, 2)

    // pb is destroyed:         (A, 1), (B, 1)
    // pa is destroyed:         (A, 0), (B, 1)
    //   A is destroyed                        output: dtor of A
    //     b_of_A is destroyed:         (B, 0)
    //       B is destroyed                    output: dtor of B
    //         a_of_B is destroyed
}

void test3() {
    shared_ptr<A> pa(new A); // (A, 1)         output: ctor of A
    shared_ptr<B> pb(new B); // (A, 1), (B, 1) output: ctor of B

    pb->a_of_B = pa;         // (A, 2), (B, 1)

    // pb is destroyed:         (A, 2), (B, 0)
    //   B is destroyed                        output: dtor of B
    //     a_of_B is destroyed  (A, 1)
    // pa is destroyed:         (A, 0)
    //   A is destroyed                        output: dtor of A
    //     b_of_A is destroyed
}

void test4() {
    shared_ptr<A> pa(new A); // (A, 1)         output: ctor of A
    shared_ptr<B> pb(new B); // (A, 1), (B, 1) output: ctor of B
    // pb is destroyed:         (A, 1), (B, 0)
    //   B is destroyed                        output: dtor of B
    //     a_of_B is destroyed
    // pa is destroyed:         (A, 0)
    //   A is destroyed                        output: dtor of A
    //     b_of_A is destroyed
}
```