[temp] 항목 8. 예외가 소멸자를 떠나지 못하도록 붙들어 놓자

```
class Widget {
public:
        ...
        ~Widget() {...} // 소멸하는 과정에서 예외가 발생된다고 생각하자.
};

void doSomething() {
        std::vector<Widget> v;
        ...
}       // v는 여기서 자동소멸됨.
```

위 함수로 부터 예외가 발생한다고 하자. 이 때 v 에 들어있는 Widget이 10개라고 가정하면 다 정상적으로 소멸되는 case가 사용자가 원하는 것이다. 하지만, 첫 번째 것을 소멸시키는 도중 예외가 발생했다고 가정하면, 나머지 9개에 대해서는 여전시 소멸되어야 하므로 이들에 대한 소멸자를 호출할 것이다. 이런 예외 상황 발생은 대응에 어려움이 있다. 왜냐하면 이러한 예외 조건이 발생하면 프로그램이 종료되던지, 미정의된 동작을 보이게 되던지 하게 될텐데, 이 경우에는 정의되지 않은 동작을 보이게 된다. 이런 문제의 원인은 예외를 발생시키는 소멸자에 있다.

아래 예시를 통해 더 자세히 살펴보자
아래는 사용자가 DBConnection에 대해 직접 close를 하도록 설계되었는데, 이 경우 사용자가 close 사용을 망각했을 시, 정상적인 close가 실행되지 않는다는 문제가 있다.

```
class DBConeection {
public:
        ...
        // 편의상 매게변수는 제외함
        static DBConnection create();   // DBConnection의 객체를 반환함.

        void close();   // 연결을 닫는다. 이떄, 실패하면 예외를 던짐.
};
```

그래서 아래와 같이 소멸자를 통해 관리해주도록 하면 사용자의 기억에 의존하지 않고 안전하게 자원을 관리할 수 있다.

```
/* DB connection을 관리하는 클래스 */
class DBConn {
public:
        ...
```

```
        // DB가 항상 닫히도록 챙겨주는 함수이다.
        ~DBConn() {
                db.close();
        }

private:
        DBConnection db;
};
```

여기서 close에 예외가 발생한다면 소멸자는 예외를 전파하게 된다. 즉, 소멸자에서 예외가 나가도록 내버려 둔다. 이에 대해서는 2가지 방법으로 접근해볼 수 있다.

1. 프로그램 종료

    아래와 같이 예외가 발생하면 프로그램을 종료시켜버릴 수 있는데, 이 경우 예외로 인해 발생할 수 있는 미정의 동작이 발생하지 않게 해준다

```
DBConn::~DBConn() {
        try { db.close(); }
        catch (...) {
                // close 호출이 실패했다는 로그를 작성함.
                std::abort();
        }
}
```

1. 예외를 삼켜버림

    아래의 경우는 예외가 발생 했을 때, 경고메세지를 출력하고 이후 동작은 앞서 발생한 예외를 무시하고 그대로 실행하게 된다.

```
DBConn::~DBConn() {
        try { db.close(); }
        catch (...) {
                // close 호출이 실패했다는 로그를 작성함.
        }
}
```

위 두가지 경우 모두 그리 좋을 것은 없어보인다. 예외가 발생한 요인에 대한 어떤 조치를 할 수 없다.

그래서 아래와 같이 인터페이스를 잘 설계해서 발생할 소지가 있는 문제에 대처할 기회를 사용자가 가질 수 있도록 해줄 수 있다.

```
class DBConn {
public:
        ...

        void close() { // db.close랑 다름, DBConn에서 정의된 함수
                db.close();
                isClosed = true;
        }

        ~DBConn() {
                if (!isClosed)
                try {
                        db.close();
                }
                catch(...) {
                        // close 에 대한 호출 실패 로그
                        ...
                }
        }

private:
        DBConnection db;
        bool isClosed = false;
};
```

위 설계의 핵심은 "어떤 동작이 예외를 일으키면서 실패할 가능성이 있고, 그 예외를 처리해야할 필요가 있다면 그 예외는 소멸자가 아닌 다른 함수에서 비롯되어야한다"에 있다.

위와 같이 설계하므로써, 사용자는 함수에서 발생할 예외 case에 대해 대처할 수 있는 기회를 가질 수 있다. (비슷한 의미이긴한데 코드 작성 시, 해당 동작이 가질 수 있는 예외 상황에 대해 한 번 더 생각 해볼 수 있다는 장점도 있을 것 같다)

이것만은 잊지 말자!

- 소멸자에서는 예외가 빠져나가면 안된다. 만약 소멸자 안에서 호출된 함수가 예외를 던질 가능성이 있다면, 어떤 예외이든지 소멸자에서 모두 받아낸 후 삼켜 버리던지 끝내야한다.
- 어떤 클래스의 연산이 진행되다가 던진 예외에 대해 사용자가 반응해야할 필요가 있다면 해당 연산을 제공하는 함수는 반드시 보통의 함수 (즉, 소멸자가 아닌 함수)에 만들어 인터페이스로 제공해야한다.