Table of Contents

Prevent exceptions from leaving destructors

C++ رخ دادن **exceptions** رو توی مخرب‌هارو منع نکرده، ولی قطعا استفاده ازش دلسرد کننـده خواهـد بود. برای این موضوع دلایل خوبی را خواهیم آورد، مثال زیر رو ببینید:

```
class Widget
{
public:
   ~Widget() {}   //assume this might emit an exception
};
void so_something()
{
   std::vector<Widget> v;
}                    // v automatically destroyed here
```

وقتی که بردار **v** نابود میشه، مسوول این هستش که هر چیزی که **Widget** داره رو هم نابود کنه. در نظر بگیرید که **v** دارای ده تا **Widget** باشه، و در طی **destruction** شیء اول، یـک **exception** رخ بـده. در این صورت همه‌ی **Widget** های دیگر هنوز نابود نشده‌اند(در غیر این صورت منابعی که نگه داشته‌اند به عنوان **leak** شناخته می‌شود)، بنابراین **v** باید **destructor** نه تای بـاقی مانـده را **invoke** کنـد. حـال فرض کنید که در طی فراخوانی دومی هم یک **exception** رخ بده. در این صورت دو تا **exception** فعال توی برنامه مون داریم و همین یه دونه اضافه هم برای C++ خیلی به حساب میاد. بر اساس این که دقیقا تحت چه شرایطی این دو تا استثنا به وجود اومده ممکنه برنامه **terminate** بشـه و یـا رفتـار نامشـخص داشته باشه. در مورد مثـال خودمـون رفتـار نامشـخص نـتیجه‌ی همچین چـیزی خواهـد بـود. در مـورد نگه‌دارنده‌های **STL** رفتـار نامشـخص را خـواهیم گـرفت و در مـورد نگـه دارنـده‌های **TR1** (آیتم ۵۴ رو ببینید) و یا حتی **array** نیز به همین صورت است. در این صـورت C++ قـرار دادن **exception** رو تـوی **destructor** رو به هیچ وجه دوست نداره.

فهم مفهومی که گفتیم سخت نیست، ولی چطور یک مخرب بسازیم که نیاز به اجرای یک عملیات داره و ممکنه این عملیات نیز **fail** بشه؟ به طور مثال فرض کنید که شما دارید روی یـک کلاس بـرای ارتبـاط دیتابیس کار می‌کنید:

```
class DBConnection{
public:

   static DBConnection create();  //function to return DBConnection
                                  //objects; params ommited for simplicity
   void close(); //close connection;throw an exception if closing fails
};
```

برای این که مطمعن بشیم که که مشتری‌های کد، فراموش نمی‌کنند که یک شیء **Dbconnection** رو **close** کنند، یک روش عقلانی این است که یک مدیریت منابع برای **DBConnection** بنویسیم که متد **close** رو توی مخرب کلاس فراخوانی کنه. چنین مدیریت منابعی به طور مفصل در فصل سوم مورد بررسی خواهد گرفت، اما در اینجا، ما صرفا به بررسی مخرب برای چنین کلاسی خواهیم پرداخت:

```
class DBconn{       //class to manage DBConnection
public:                 //objects
   ~DBconn()          //make sure database connections
   {                  //are always closed
      db.close();
   }
private:
   DBConnection db;
};
```

این به برنامه‌نویس اجازه میده که یک کد مثل زیر رو بنویسه.

```
{                                                   //open a block
   DBconn dbc(DBConnection::create());       // create DBConnection object
                                             //and turn it over to a DBCon object to manage
                                             //use the DBConnection object via the DBconn interface


}                          //at the end of block, the DBconn object is destroyed, thus automatically
                          // calling close on the DBConnection object
```

این کد در صورتی که **close** با موفقیت انجام بشه، مشکلی نداره، اما فرض کنید که در این کد یک استثناء رخ بده، در این صورت ما توی مخرب این کلاس مشکل داریم. دو روش برای جلوگیری کردن از این مشکل وجود داره.

یکی این که وقتی **close** به ما استثناء داد برنامه رو **terminate** کنیم، که این کار معمولا با فراخوانی **abort** انجام میشه

```
try{db.close();}
catch(...)
{
   //make log entry that the call to close failed;
   std::abort();
}
```

این کاری که کردیم در صورتی که برنامه نتونه بعد از این اجرا بشه منطقیه. و این مزیت رو داره که، اگه اجازه بدیم که یک **exception** از مخرب صادر بشه ممکنه که به یک رفتار **undefined** برخورد کنیم، که این کار از این مشکل جلوگیری می‌کنه. در این صورت فراخوانی کردن **abort** ممکنه از **undefined behavior** جلوگیری کنه.

روش دوم بلعیدن **exception** در هنگام فراخوانی کردن **close** است.

```
try{db.close();}
catch(...)
{
   //make log entry that the call to close failed;
}
```

عموما، بلعیدن **exception** ایده بدی است، چون در این مورد اطلاعات مفید رو از دست میدیم – یه چیزی اشتباهه، اما بلعیدن **exception** به این که برنامه **terminate** بشه و یا دچار **undefined behavior** بشه ارجحیت داره. برای این که این راه حل یک راه حل قابل اعتماد باشه، باید اطمینان داشته باشیم که برنامه پس از این که وارد **exception** میشه حتما بتونه ادامه‌ی برنامه رو بره.

مشکلی که در مورد این دوتا روش گفته شده داریم این است که راهی برای برخورد با این شرایط وجود ندارد که **close** منجر به **exception** میشه.

یک استراتژی بهتر این است که رابط **DBConn** رو به نحوی طراحی کنیم که مشتری این امکان رو داشته باشه که بر اساس مشکلی که وجود داره یکی رو انتخاب کنه. به طور مثال، **DBConn** می‌تونه یک تابع رو پیشنهاد بده که، که به مشتری این شانس رو میده که بتونه استثناء به وجود اومده رو تصحیح کنه. این تابع می‌تونه بفهمه که مشکل از کجا بوده، خودش رو توی **destructor** ببنده یا نبنده. این به ما این اجازه میده رو که ارتباط **leak** نداشته باشه. در هر صورت اگر **close** به ما دوباره یک استثنا بده می‌تونیم برنامه رو **terminate** کنیم و یا این که اون رو ببلعیم.

```
class DBconn{
public:
    DBconn(DBConnection){}
    void close()
    {
        db.close();
        closed=true;
    }
    ~DBconn()
    {
        if(!closed)
        {
            try {
                db.close();
            } catch (...) {
                //make log entry that call to close friend
            }
        }
    }
private:
    DBConnection db;
    bool closed;
};
```

Moving the responsibility for calling close from DBConn 's destructor to DBConn 's client (with DBConn 's destructor containing a “backup” call) may strike you as an unscrupulous shift of burden. You might even view it as a violation of Item 18's advice to make interfaces easy to use correctly. In fact, it's neither. If an operation may fail by throwing an exception and there may be a need to handle that exception, the exception has to come from some non-destructor function. That's because destructors that emit exceptions are dangerous, always running the risk of premature program termination or undefined behavior. In this example, telling clients to call close themselves doesn't

impose a burden on them; it gives them an opportunity to deal with errors they would otherwise have no chance to react to. If they don't find that opportunity useful (perhaps because they believe that no error will really occur), they can ignore it, relying on DBConn 's destructor to call close for them. If an error occurs at that point — if close does throw — they're in no position to complain if DBConn swallows the exception or terminates the program. After all, they had first crack at dealing with the problem, and they chose not to use it.