If an exception is thrown while executing a nondelegating constructor, the object being initialized is considered only **partially constructed** (i.e., the object is not yet known to be in a valid state), and hence its destructor will *not* be invoked:

```
#include <iostream>  // std::cout

struct S2
{
    S2()  { std::cout << "S2() ";  throw 0; }
    ~S2() { std::cout << "~S2() ";          }
};

void f() try { S2 s; } catch(int) { }
    // prints only "S2() " to stdout (the destructor of S2 is never invoked)
```

Although the destructor of a partially constructed object will not be invoked, the destructors of each successfully constructed base ~~and of~~ data members~~s~~ will still be invoked:

```
#include <iostream>  // std::string

using std::cout;
struct A { A() { cout << "A() "; } ~A() { cout << "~A() "; } };
struct B { B() { cout << "B() "; } ~B() { cout << "~B() "; } };

struct C : B
{
    A d_a;

    C()  { cout << "C() "; throw 0; }  // nondelegating constructor that throws
    ~C() { cout << "~C() ";         }  // destructor that never gets called
};

void f() try { C c; } catch(int) { }
    // prints "B() A() C() ~A() ~B()" to stdout
```

Notice that base class `B` and member `d_a` of type `A` were fully constructed, and so their respective destructors are called, even though the destructor for class `C` itself is never executed.

However, if an exception is thrown in the body of a delegating constructor, the object being initialized is considered **fully constructed**, as the target constructor must have returned control to the ~~delegator~~; hence, the object's destructor *is* invoked: