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

struct S3
{
    S3()           { std::cout << "S3() ";             }
    S3(int) : S3() { std::cout << "S3(int) "; throw 0; }
    ~S3()          { std::cout << "~S3() ";            }
};

void f() try { S3 s(0); } catch(int) { }
    // prints "S3() S3(int) ~S3() " to stdout
```

Use Cases

Avoiding code duplication among constructors

Many consider avoiding gratuitous code duplication a best practice. Having one ordinary member function call another has always been an option, but having one constructor directly invoke another constructor has not. Classic workarounds included repeating the code or else factoring the code into a private member function that would be called from multiple constructors. The drawback with this workaround is that the private member function, not being a constructor, would be unable to make use ~~efficiently~~ of member initializer lists to initialize base classes and data members. As of C++11, *delegating constructors* can be used to minimize code duplication when some of the same operations are performed across multiple constructors without having to forgo efficient initialization. Consider an `IPV4Host` class representing a network endpoint that can be constructed either (1) by a 32-bit address and a 16-bit port or (2) by an IPV4 string with `XXX.XXX.XXX.XXX:XXXXX` format[1]:

```
#include <cstdint>  // std::uint16_t, std::uint32_t
#include <string>   // std::string

class IPV4Host
{
     // ...
private:
    int connect(std::uint32_t address, std::uint16_t port);

public:
    IPV4Host(std::uint32_t address, std::uint16_t port)
    {
        if (!connect(address, port))  // code duplication: BAD IDEA
        {
            throw ConnectionException{address, port};
```

[1]Note that this initial design might itself be suboptimal in that the representation of the IPV4 address and port value might profitably be factored out into a separate **value-semantic** class, say, `IPV4Address`, that itself might be constructed in multiple ways; see *Potential Pitfalls — Suboptimal factoring* on page 51.