

Exceptions: The basics

```
/*  
 * Exception basics:  
 * #1: Exceptions basics  
  
 * #2: The need for exceptions  
  
 * #3: Exceptions at different levels  
  
 * #4: Multiple handlers for an exception  
  
 * #5: Nested try blocks  
 */
```

Exceptions: The basics



```
// When an exception is thrown, the control immediately exits the try block and goes into the catch block.  
int a{10};  
int b{10};  
  
try{  
    Item item; // When exception is thrown, control immediately exits the try block  
    // an automatic local variables are released  
    // But pointers may leak memory.  
    if( b == 0 )  
        throw 110;  
    a++; // Just using a and b in here, could use them to do anything.  
    b++;  
    fmt::println( "Code that executes when things are fine" );  
} catch(int ex){  
    fmt::println( "Something went wrong. Exception thrown : {}", ex );  
}  
fmt::println( "Done!" );
```

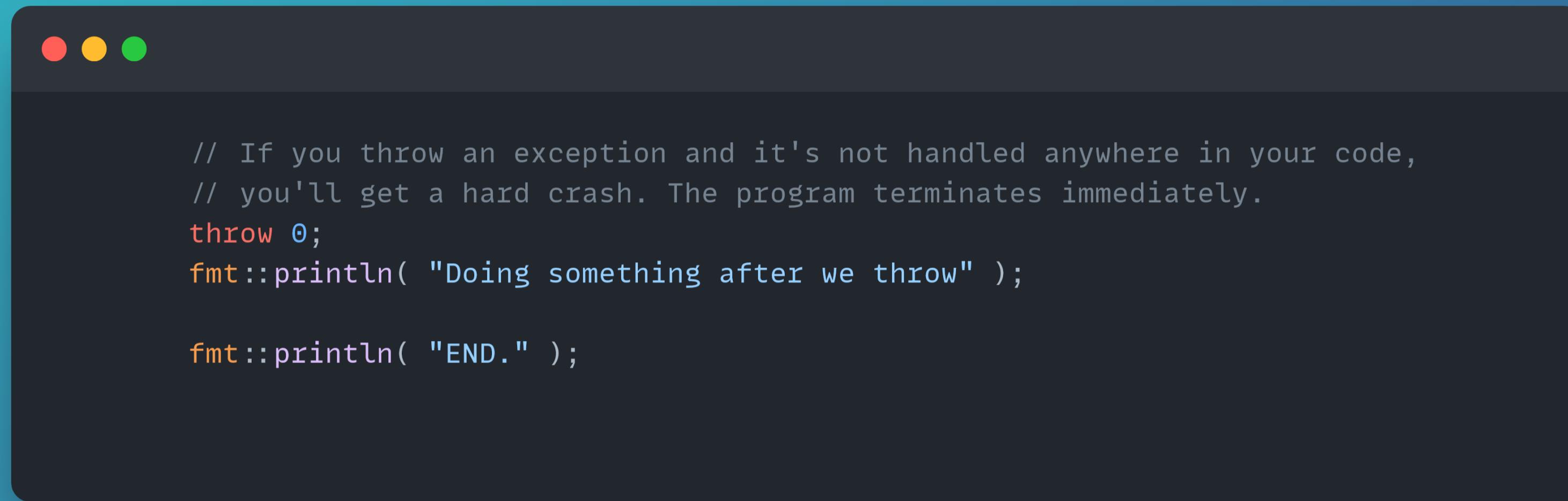
Exceptions: Throwing pointers

```
// Throwing pointers is a recipe for disaster, as by the time
// the catch block executes, the memory allocated and used in try
// block is pointing to invalid data. The program may seem to work
// sometimes but there are no guarantees you'll always get valid stuff
// if you dereference pointers allocated in the try block.
int c{0};
try{
    int var{55};
    int* int_ptr = &var;
    if(c == 0)
        throw int_ptr;
    fmt::println( "Keeping doing some other things..." );
}catch(int* ex){
    fmt::println( "Something went wrong. Exception thrown : ", *ex );
}
fmt::println( "END." );
```

Exceptions: Potential Memory Leaks

```
// Potential memory leaks
// The destructor for Item is never called when we're thrown out of the
// try block, and memory is leaked.
// Using smart pointers solves the issue. Memory is released when we get out of scope.
int d{0};
try{
    //try_catch_blocks::Item * item_ptr = new try_catch_blocks::Item();
    std::shared_ptr<Item> item_ptr = std::make_shared<Item>();
    if(d == 0)
        throw 0;
    fmt::println( "Keeping doing some other things..." );
} catch(int ex){
    fmt::println( "Something went wrong. Exception thrown : {}", ex );
}
fmt::println( "END." );
```

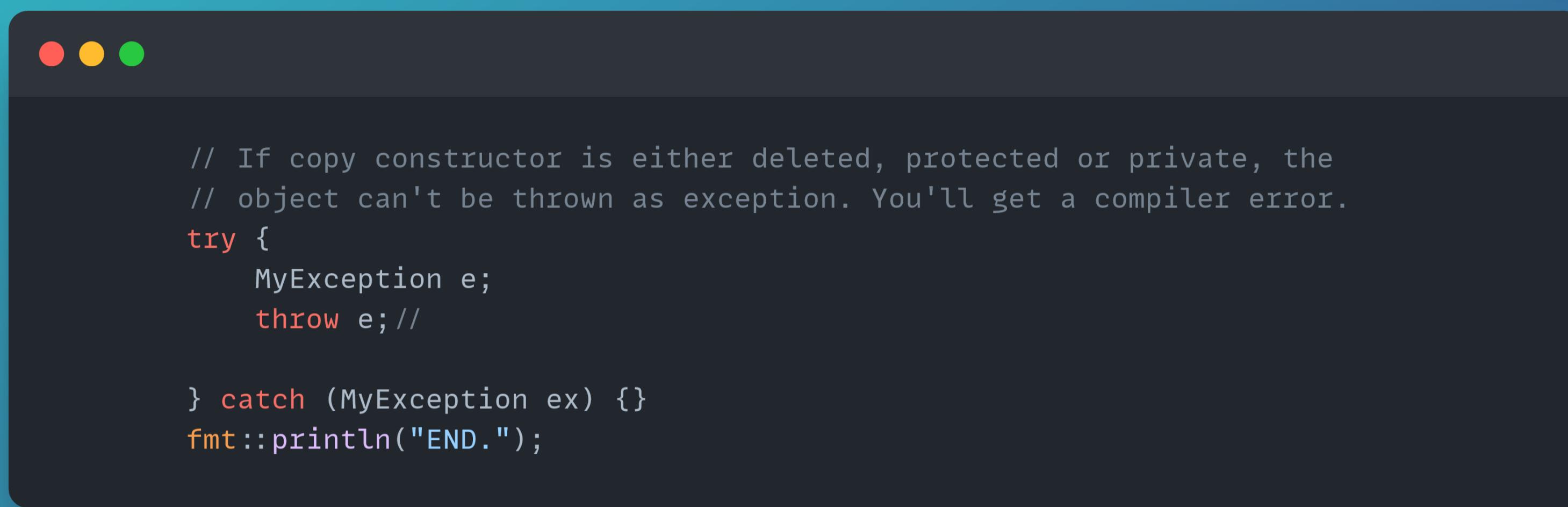
Exceptions: Unhandled exceptions



A screenshot of a terminal window with a dark background and light-colored text. The window has three colored window control buttons (red, yellow, green) at the top left. The terminal output shows the following C++ code:

```
// If you throw an exception and it's not handled anywhere in your code,  
// you'll get a hard crash. The program terminates immediately.  
throw 0;  
fmt::println( "Doing something after we throw" );  
  
fmt::println( "END." );
```

Exceptions: Copy Constructors

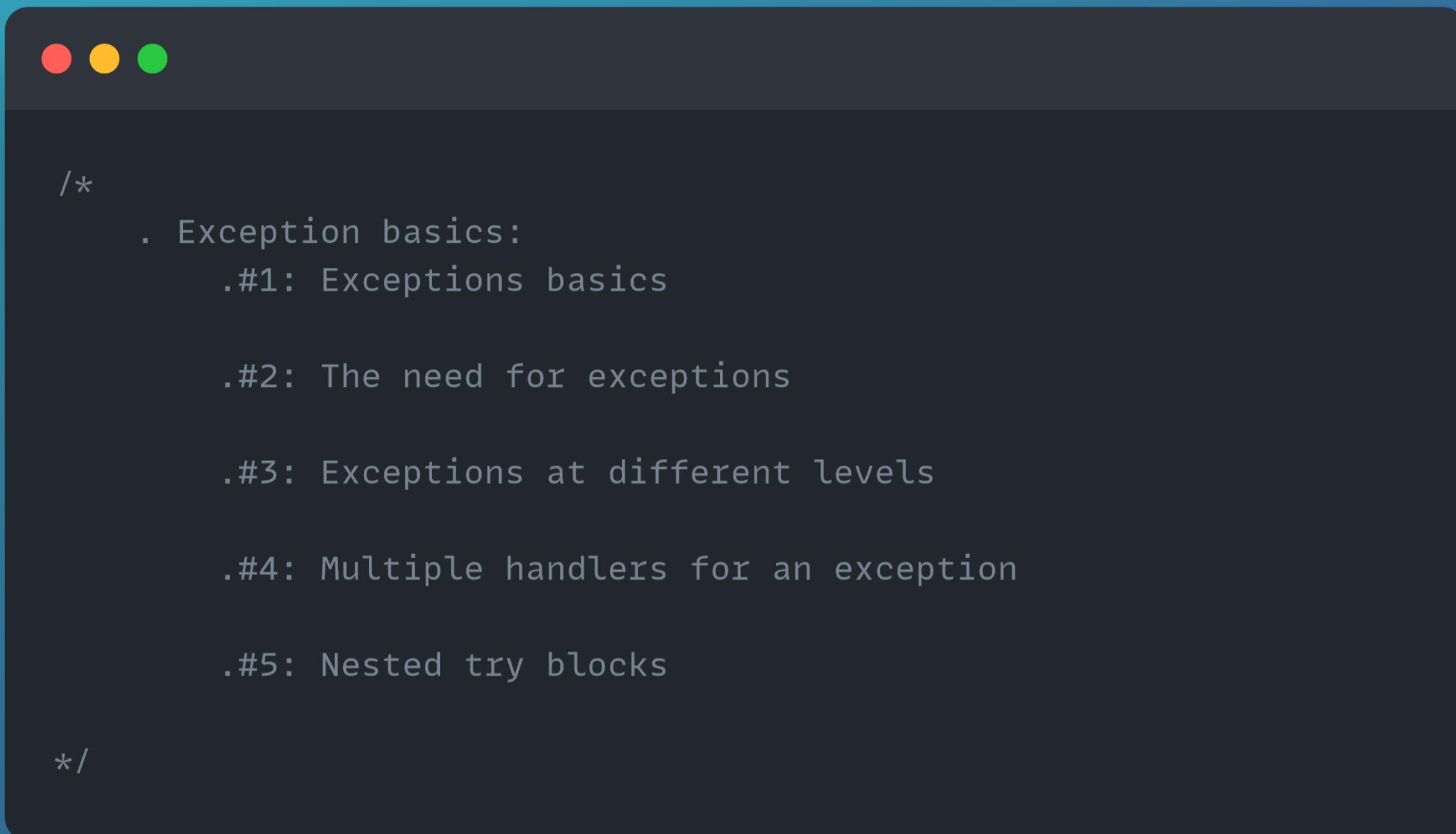


A screenshot of a terminal window with a dark background and light-colored text. The window has three colored window control buttons (red, yellow, green) at the top left. The text in the terminal is a C++ code snippet:

```
// If copy constructor is either deleted, protected or private, the
// object can't be thrown as exception. You'll get a compiler error.
try {
    MyException e;
    throw e; // Compiler error here

} catch (MyException ex) {}
fmt::println("END.");
```

Exceptions: The basics - Recap



The image shows a terminal window with a dark background and light-colored text. At the top, there are three small colored circles (red, yellow, green) typically used for window control buttons. The text in the terminal is a list of five points under a single-line comment block:

```
/*
 . Exception basics:
 .#1: Exceptions basics

 .#2: The need for exceptions

 .#3: Exceptions at different levels

 .#4: Multiple handlers for an exception

 .#5: Nested try blocks

 */
```