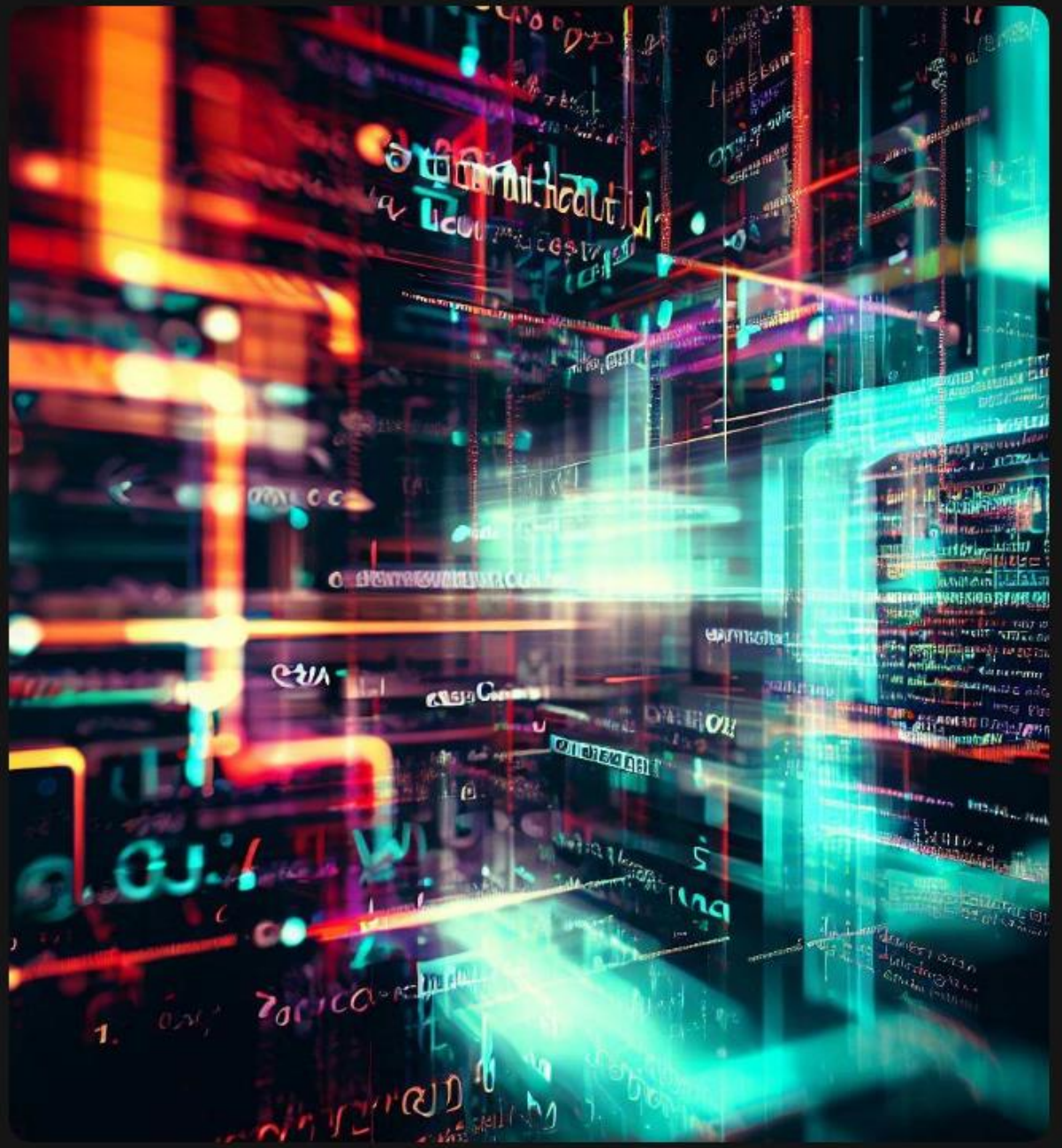
Mastering Exception-Safe Code: The Rule of Three, Five, and Zero in C++

Introduction to the Rule of Three and Rule of Five

In C++, when we create a class that manages resources, such as dynamic memory allocation or file handles, we need to ensure that the class is exception-safe. This means that if an exception is thrown during the execution of a member function, the object's internal state remains consistent and any resources it manages are properly released.

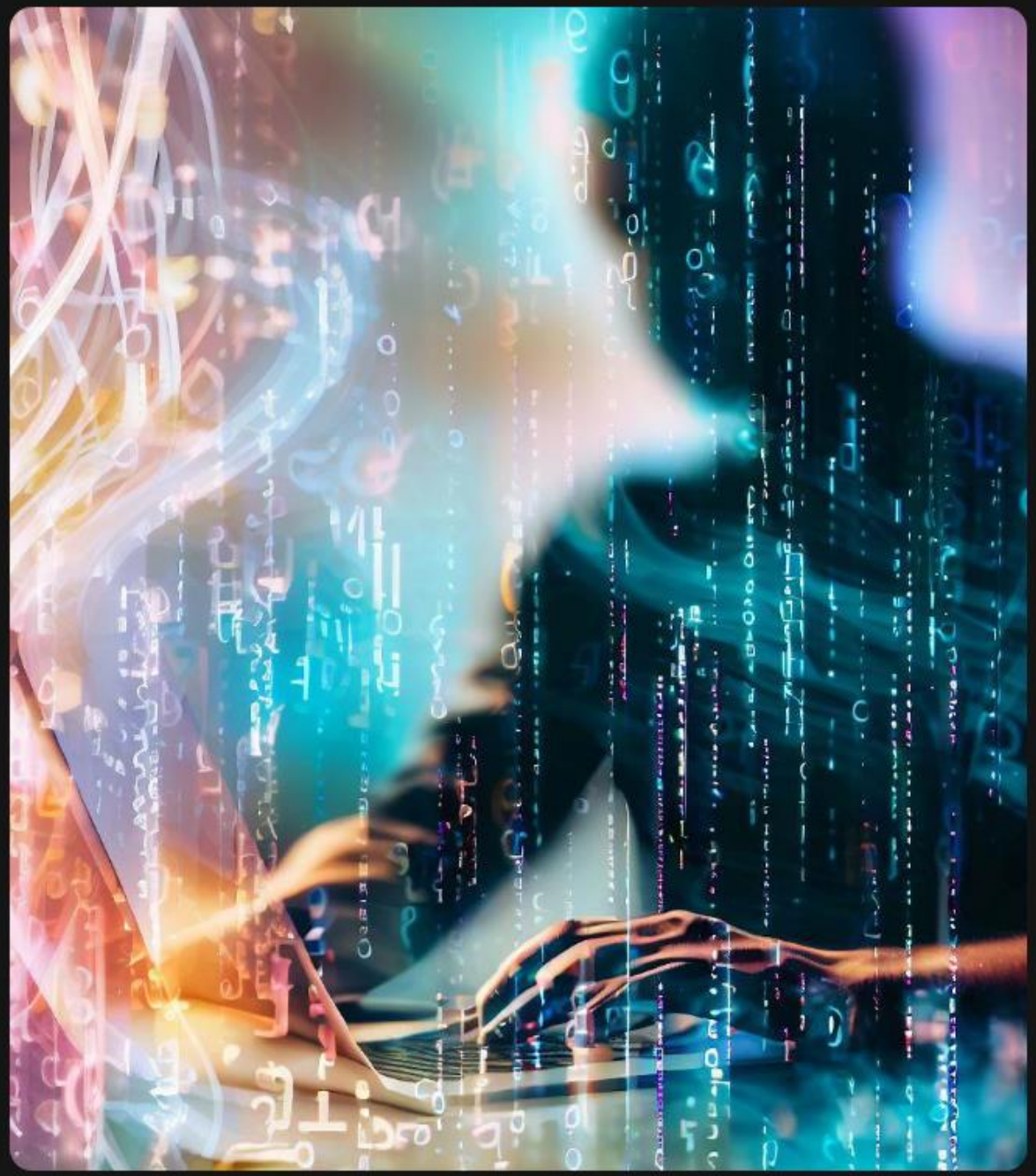
The rule of three and rule of five are two guidelines that help us achieve this goal. The rule of three states that if a class needs to manage a resource, it must define a copy constructor, a copy assignment operator, and a destructor. The rule of five extends this to include move constructor and move assignment operator for classes that manage moveable resources.



Implementing the Rule of Three and Rule of Five

To implement the rule of three and rule of five, we need to follow certain guidelines. First, we should use smart pointers or containers to manage resources whenever possible, as they automatically handle memory management and can be easily copied or moved.

Second, we should define the copy constructor and copy assignment operator to perform deep copies of the managed resource, ensuring that each object has its own copy of the resource. Similarly, the move constructor and move assignment operator should transfer ownership of the resource from the source object to the destination object, leaving the source object in a valid but unspecified state.



Rule of Three:

```
class rule_of_three
{
    char* cstring; // raw pointer used as a handle to a dynamically-allocated memory block

    rule_of_three(const char* s, std::size_t n) // to avoid counting twice
        : cstring(new char[n]) // allocate
    {
        std::memcpy(cstring, s, n); // populate
    }
public:
    rule_of_three(const char* s = "")
        : rule_of_three(s, std::strlen(s) + 1) {}

    ~rule_of_three() // I. destructor
    {
        delete[] cstring; // deallocate
    }

    rule_of_three(const rule_of_three& other) // II. copy constructor
        : rule_of_three(other.cstring) {}

    rule_of_three& operator=(const rule_of_three& other) // III. copy assignment
    {
        if (this == &other)
            return *this;

        std::size_t n{std::strlen(other.cstring) + 1};
        char* new_cstring = new char[n];             // allocate
        std::memcpy(new_cstring, other.cstring, n); // populate
        delete[] cstring;                           // deallocate

        cstring = new_cstring;
        return *this;
    }

    operator const char *() const // accessor
    {
        return cstring;
    }
};
```

Rule of Five:

```
class rule_of_five
{
    char* cstring; // raw pointer used as a handle to a dynamically-allocated memory block
public:
    rule_of_five(const char* s = "") : cstring(nullptr)
    {
        if (s)
        {
            std::size_t n = std::strlen(s) + 1;
            cstring = new char[n];      // allocate
            std::memcpy(cstring, s, n); // populate
        }
    }

    ~rule_of_five()
    {
        delete[] cstring; // deallocate
    }

    rule_of_five(const rule_of_five& other) // copy constructor
        : rule_of_five(other.cstring) {}

    rule_of_five(rule_of_five&& other) noexcept // move constructor
        : cstring(std::exchange(other.cstring, nullptr)) {}

    rule_of_five& operator=(const rule_of_five& other) // copy assignment
    {
        return *this = rule_of_five(other);
    }

    rule_of_five& operator=(rule_of_five&& other) noexcept // move assignment
    {
        std::swap(cstring, other.cstring);
        return *this;
    }

// alternatively, replace both assignment operators with
//  rule_of_five& operator=(rule_of_five other) noexcept
//  {
//      std::swap(cstring, other.cstring);
//      return *this;
//  }
};
```

The Rule of Zero

In addition to the rule of three and rule of five, there is also the rule of zero.

The "Rule of Zero" is a design guideline in C++ that advocates for minimizing explicit manual resource management in your code. The rule states that instead of manually managing resources, such as memory allocation and deallocation, you should leverage existing resource-managing classes and avoid writing explicit destructors, copy constructors, copy assignment operators, move constructors, and move assignment operators.

The key idea behind the Rule of Zero is to delegate the responsibility of resource management to appropriate classes in the C++ standard library or other well-established libraries. By doing so, you can benefit from the tested and optimized implementations provided by these classes, reducing the chances of resource leaks, memory errors, and other issues that arise from manual resource management.

This guideline is especially useful when working with classes that rely on RAII (Resource Acquisition Is Initialization) techniques, such as smart pointers (`std::unique_ptr` or `std::shared_ptr`)and containers (`std::vector` , `std::string`) .

By following the Rule of Zero, you can write cleaner, more maintainable, and less error-prone code. It reduces the likelihood of resource leaks, improves code readability, and promotes code reuse by leveraging existing well-tested resource-managing classes in the standard library or other libraries.

```cpp
#include <iostream>
#include <memory>

class RuleOfZeroExample {
    std::shared_ptr<int> data;

public:
    RuleOfZeroExample() : data(std::make_shared<int>(42)) {}

    void printData() const {
        std::cout << "Data: " << *data << std::endl;
    }
};

int main() {
    RuleOfZeroExample example;
    example.printData();

    // No need for explicit destructors or deallocation.
    // The smart pointer will automatically handle memory management.

    return 0;
}
```

RAII:

RAII stands for Resource Acquisition Is Initialization. It's a programming technique used in C++ to manage resources like memory, files, network connections, and so on. The basic idea behind RAII is to tie the lifetime of a resource to the lifetime of an object (resources are properly initialized and deinitialized).

The main idea of RAII is that the resource acquisition (like allocation of memory) happens in the constructor of an object. The deinitialization (like deallocation of memory) happens in the destructor of the object. This ensures that the resource is always properly initialized and deinitialized even if there is an exception or early return statement and that there are no memory leaks or resource leaks.

For example, consider memory allocation. Without RAII, we have to write code like this:

```
int* ptr = allocateMemory();  // Acquire resource
// Use the resource
deallocateMemory(ptr);      // Deinitialize resource
```

If there is an exception in the use of the resource, we could "leak" the memory by forgetting to call `deallocateMemory`.

Using RAII, we can write:

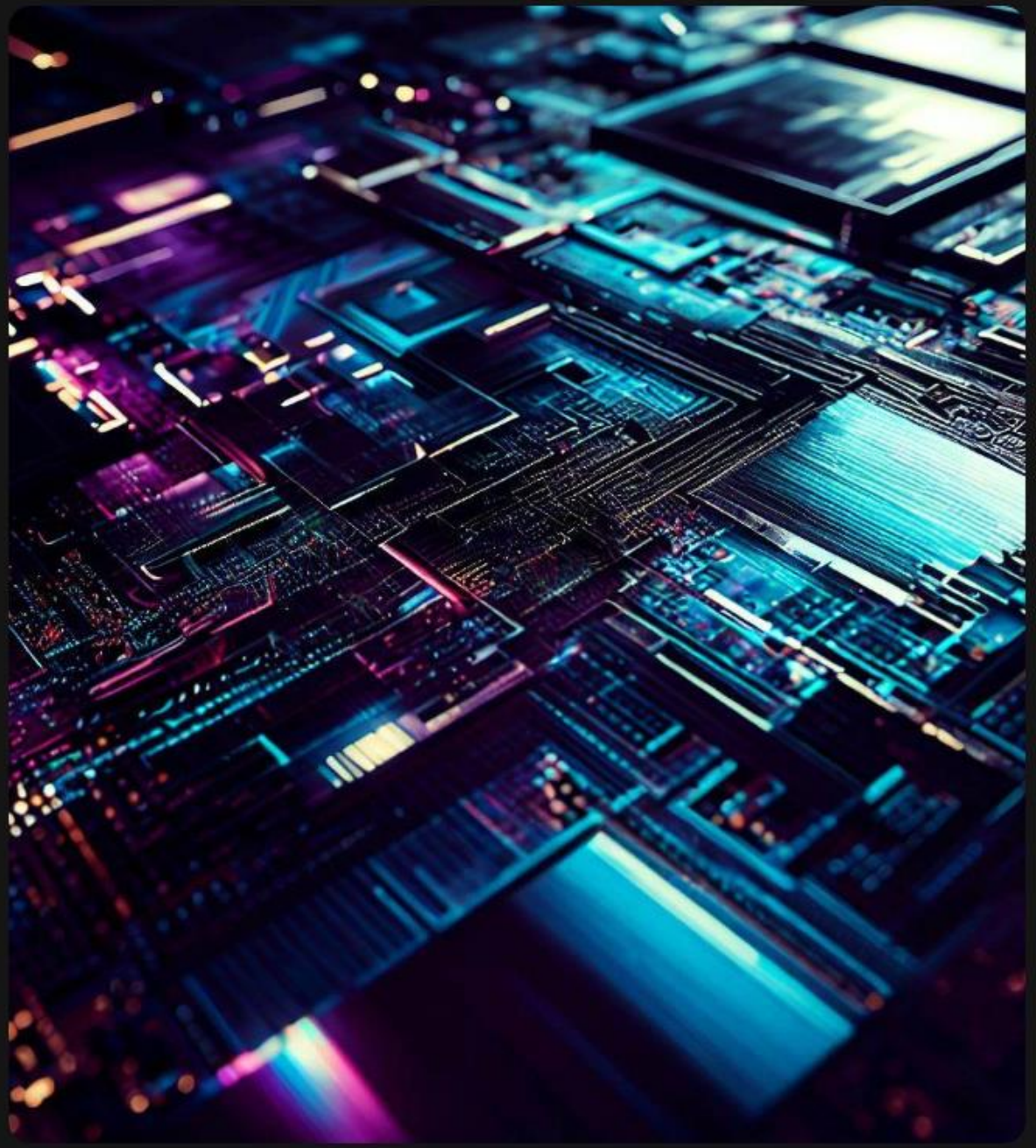
```
class MemoryBlock {
  public:
    MemoryBlock() {  // Constructor - acquire resource
        ptr = allocateMemory();
    }

    ~MemoryBlock() {  // Destructor - deinitialize resource
        deallocateMemory(ptr);
    }

  private:
    int* ptr;
};

int main() {
  // MemoryBlock created - memory allocated
  MemoryBlock block;

  // Use memory...

  // block goes out of scope - destructor called automatically,
  // memory deallocated
}
```

Now there is no chance of "leaking" the memory, since the destructor will be called no matter how we exit the scope, and deallocate the memory.

RAII is a very powerful paradigm used extensively in C++ for resources like:

- Memory (as shown above)
- Files
- Mutexes
- Database connections
- etc.

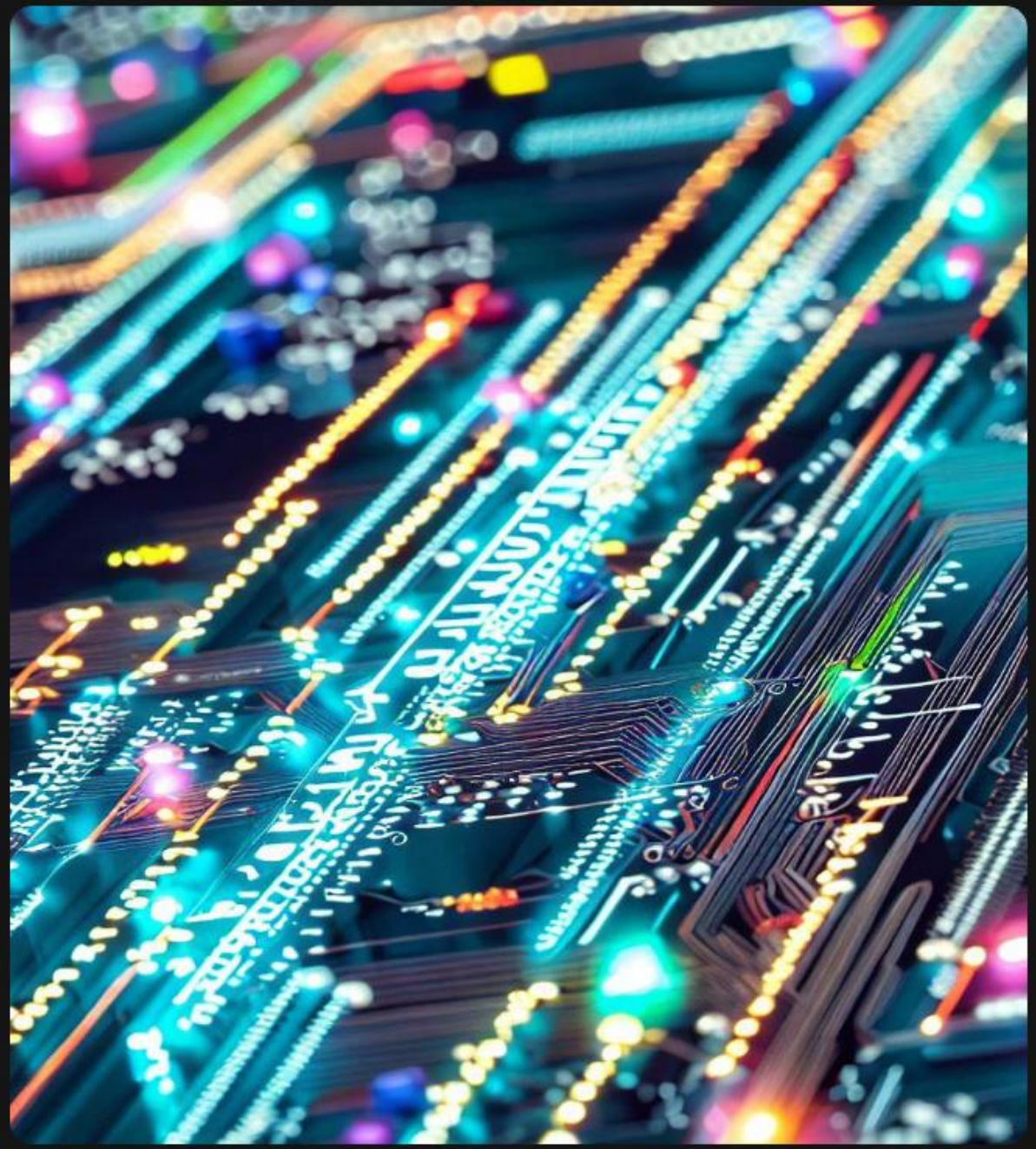
It ensures these resources are always properly initialized and cleaned up.



Benefits of Following the Rules of Thumb

By following the rule of three, rule of five, and rule of zero, we can create more robust and reliable code that is less prone to errors and bugs. These guidelines help us manage resources safely and efficiently, reducing the risk of memory leaks, dangling pointers, and other common issues.

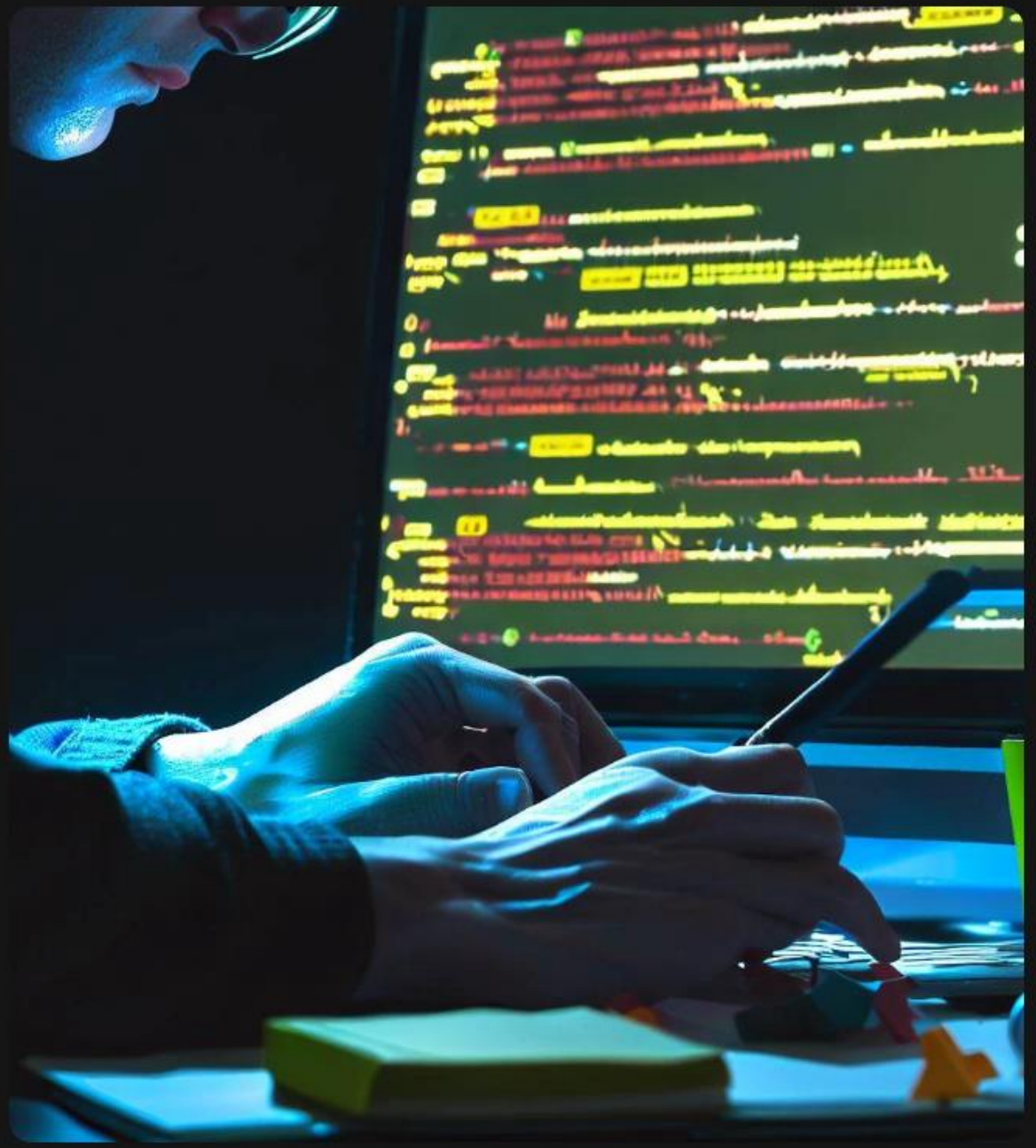
In addition, following these rules can make our code more readable and maintainable, as they provide a clear structure and organization for our classes. By defining the special member functions in a systematic way, we can ensure that our classes behave consistently and predictably, making them easier to understand and modify.



Common Pitfalls to Avoid

While the rule of three, rule of five, and rule of zero can be powerful tools for creating exception-safe code, there are also some common pitfalls to avoid. One common mistake is forgetting to define one or more of the special member functions, which can lead to undefined behavior and memory leaks.

Another pitfall is relying too heavily on raw pointers or manual memory management, which can make our code more error-prone and difficult to maintain. To avoid these issues, we should always use smart pointers and containers whenever possible, and carefully manage the lifetime of our resources.



Conclusion

In conclusion, the rule of three, rule of five, and rule of zero are important guidelines for building exception-safe code and managing resources in C++. By following these rules, we can create more robust and reliable software that is easier to understand and maintain.

While there are some common pitfalls to avoid, such as forgetting to define special member functions or relying too heavily on manual memory management, by using smart pointers and containers and carefully managing our resources, we can ensure that our code is both efficient and safe.

Thanks.