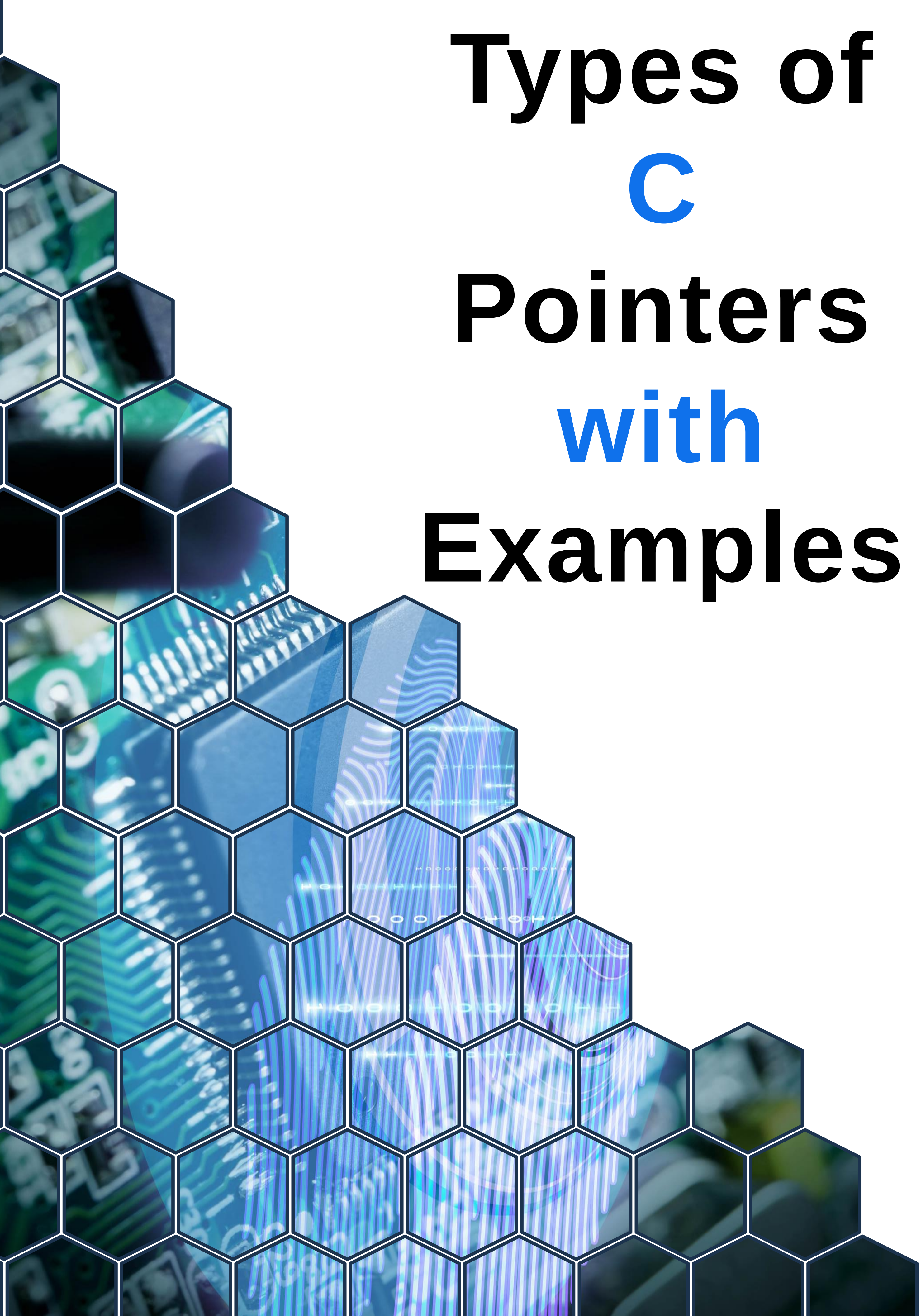
Types of C Pointers with Examples



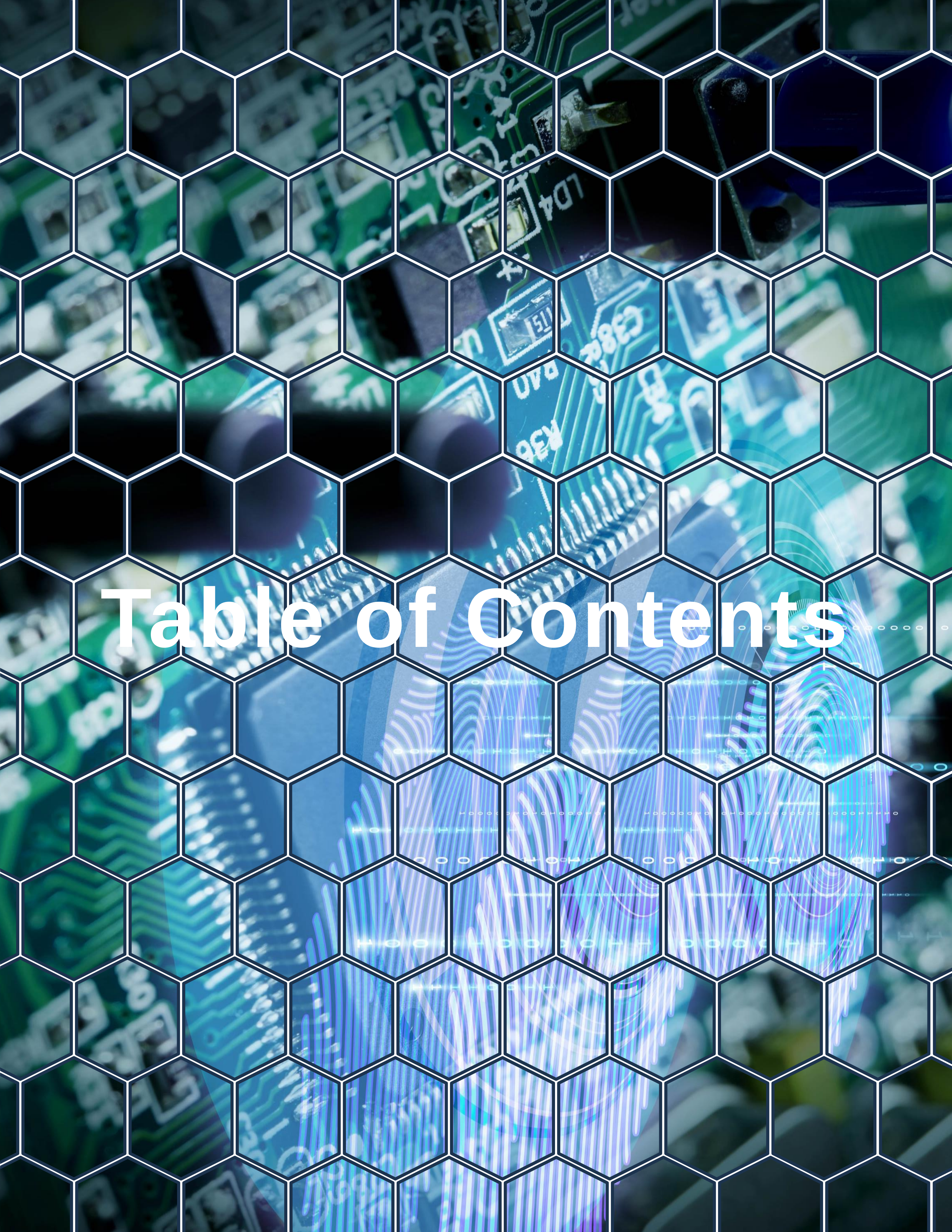
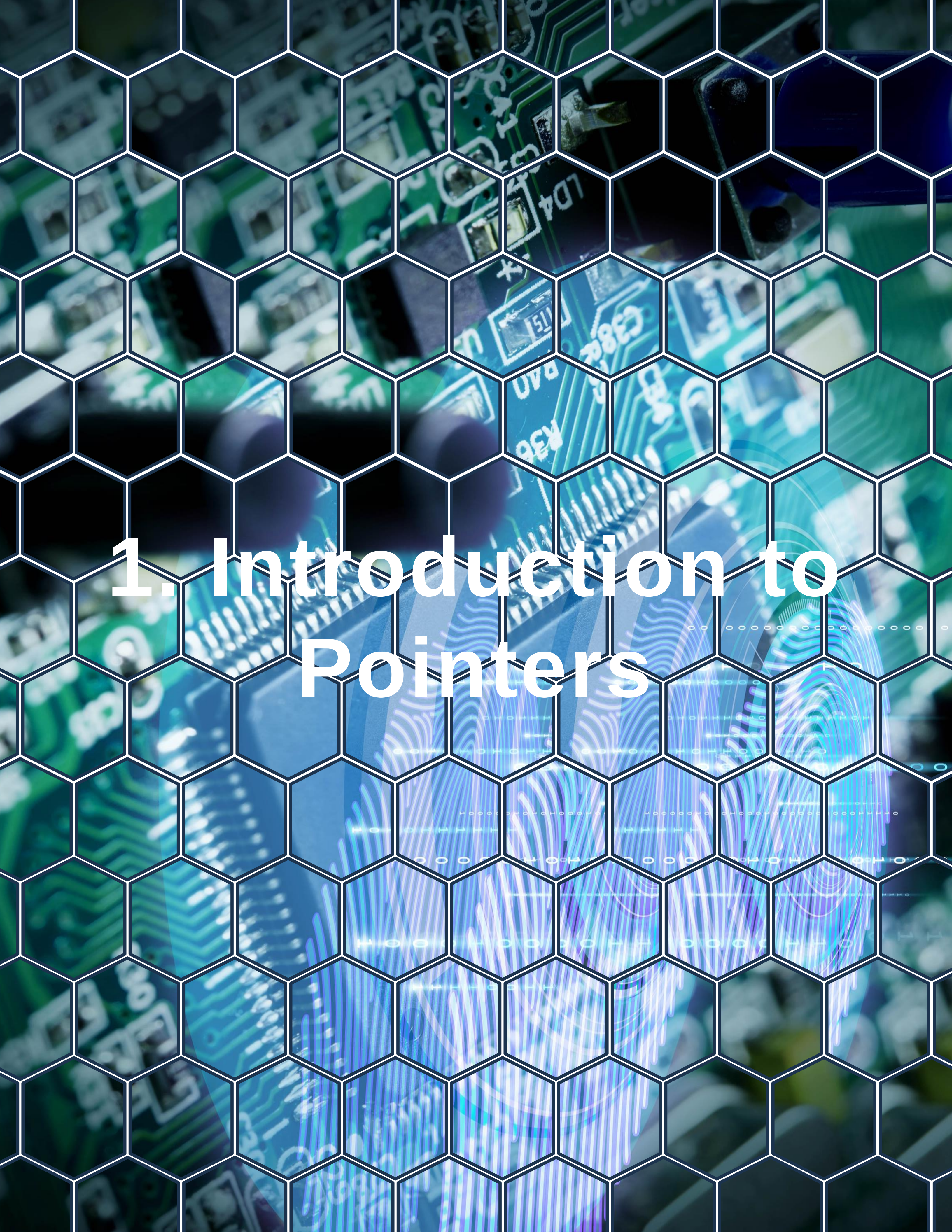


Table of Contents

Table of Contents



1. Introduction to Pointers

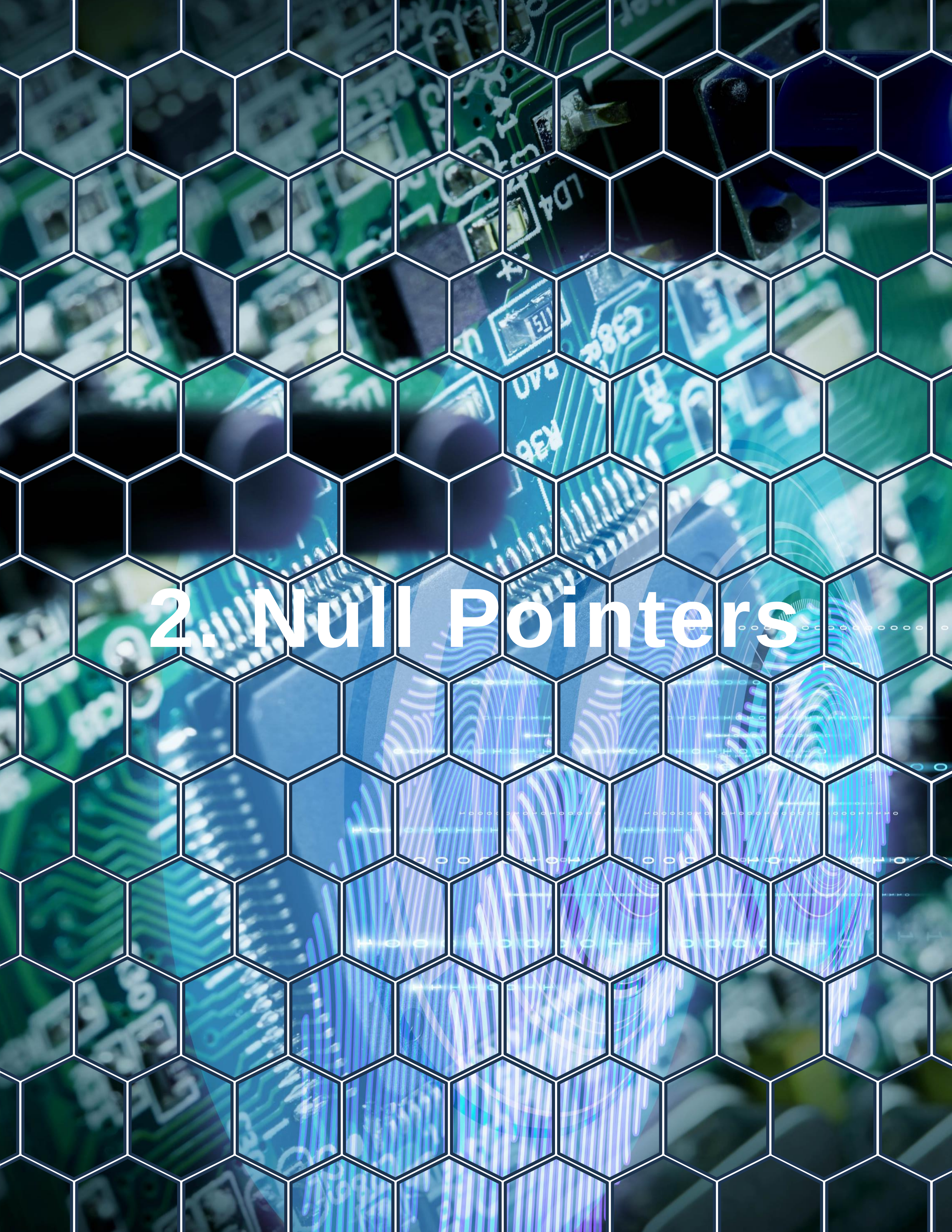
1. Introduction to Pointers

Pointers are one of the most powerful and critical features in the C programming language. They allow for efficient memory manipulation, dynamic memory allocation, and advanced data structures like linked lists and trees. However, pointers can also be a source of bugs and undefined behavior if not used properly. Understanding the different types of pointers in C is essential for writing efficient and safe code, especially in low-level and embedded systems development.

1. Introduction to Pointers

In C, a pointer is a variable that stores the memory address of another variable. The syntax involves the use of the asterisk (`*`) to declare a pointer, and the address-of operator (`&`) to get the memory address of a variable.

```c
int x = 10;
int *ptr = &x; // ptr points to x
```



2. Null Pointers

2. Null Pointers

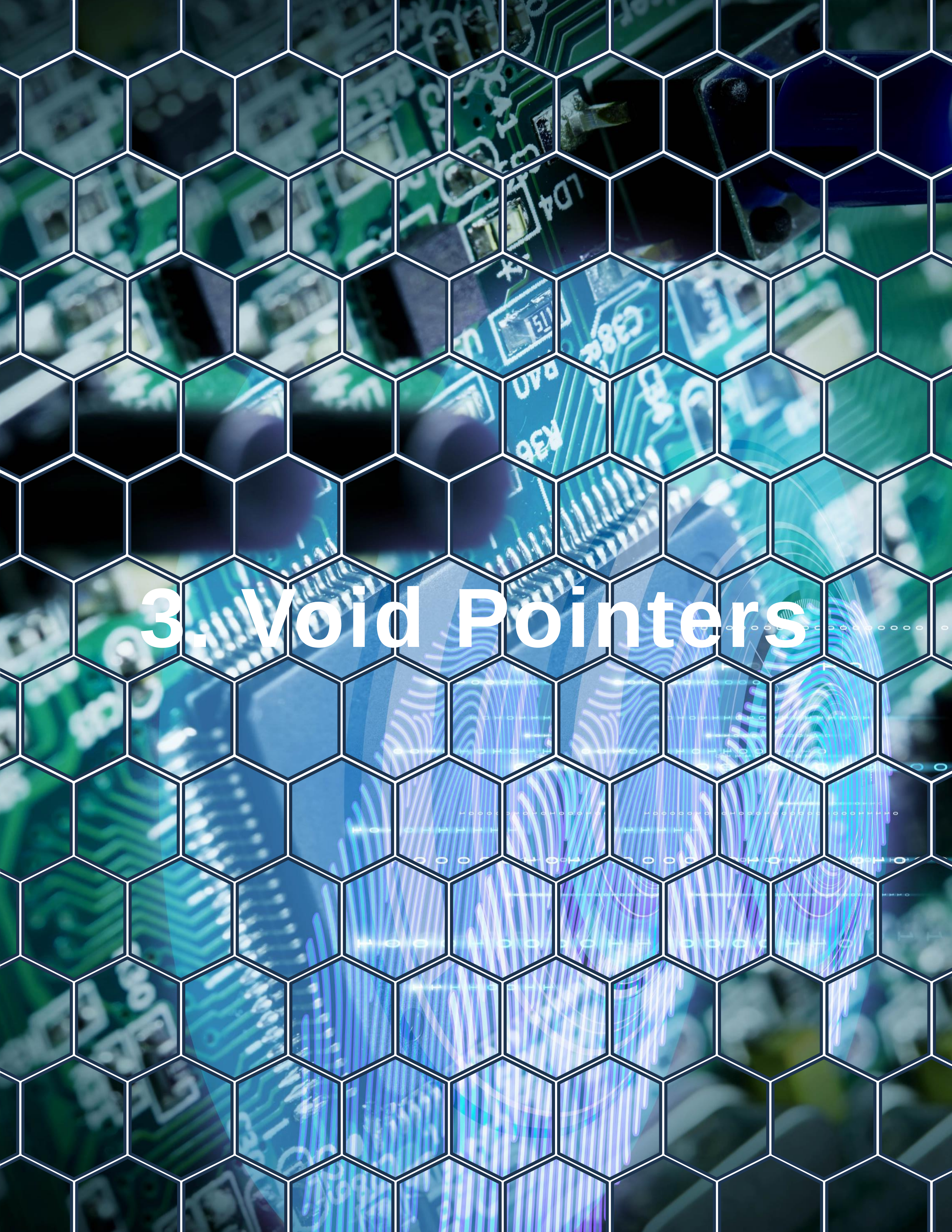
A **null pointer** is a pointer that does not point to any valid memory location. It is often used for initialization and error-checking.

```c
int *ptr = NULL;

if (ptr == NULL) {
    printf("Pointer is null.\n");
}
```

Use Case:

Safely indicate that a pointer is not currently pointing to anything valid.

3. Void Pointers



3. Void Pointers

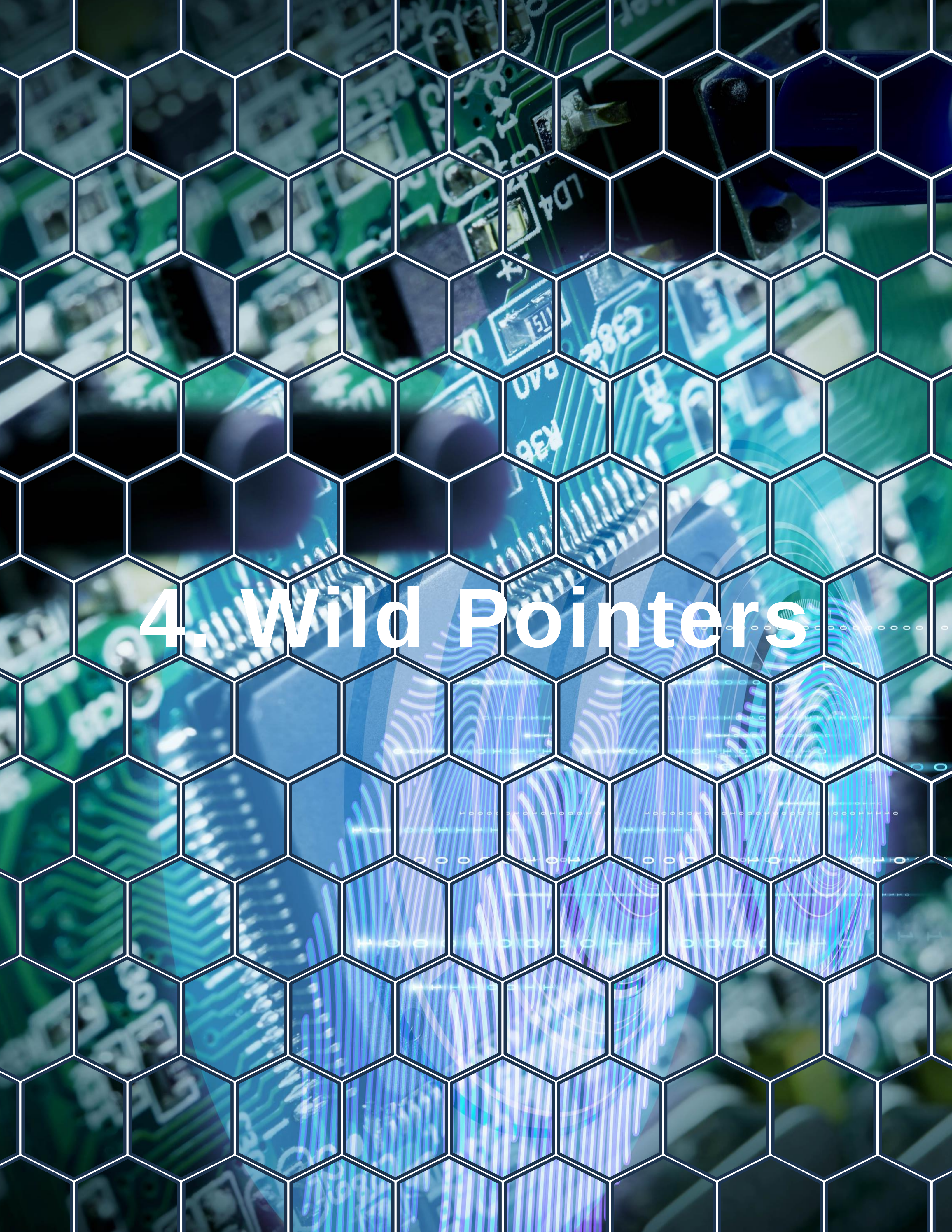
A void * **pointer** can point to any data type.

It's commonly used for generic programming, such as in memory allocation functions.

```
void *ptr;
int x = 100;
ptr = &x;

printf("Value of x through void pointer: %d\n", *(int *)ptr);
```

Use Case:

Useful in functions that deal with different data types.



4. Wild Pointers

4. Wild Pointers

Wild pointers are uninitialized pointers. Using them leads to undefined behavior.

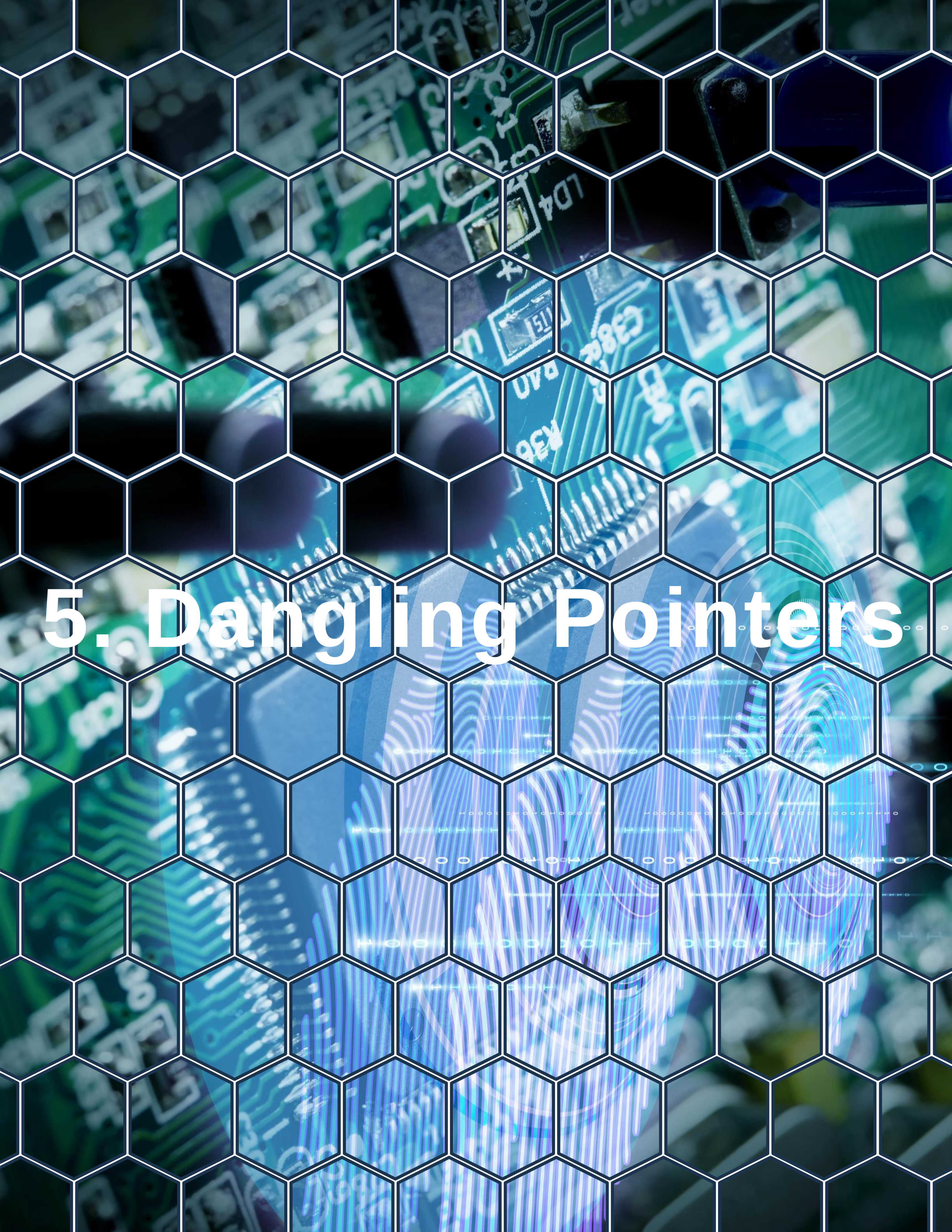
```
int *ptr; // Uninitialized - wild pointer
*ptr = 5; // Dangerous!
```

Tip:

Always initialize pointers before use:

```
int *ptr = NULL;
```

5. Dangling Pointers



5. Dangling Pointers

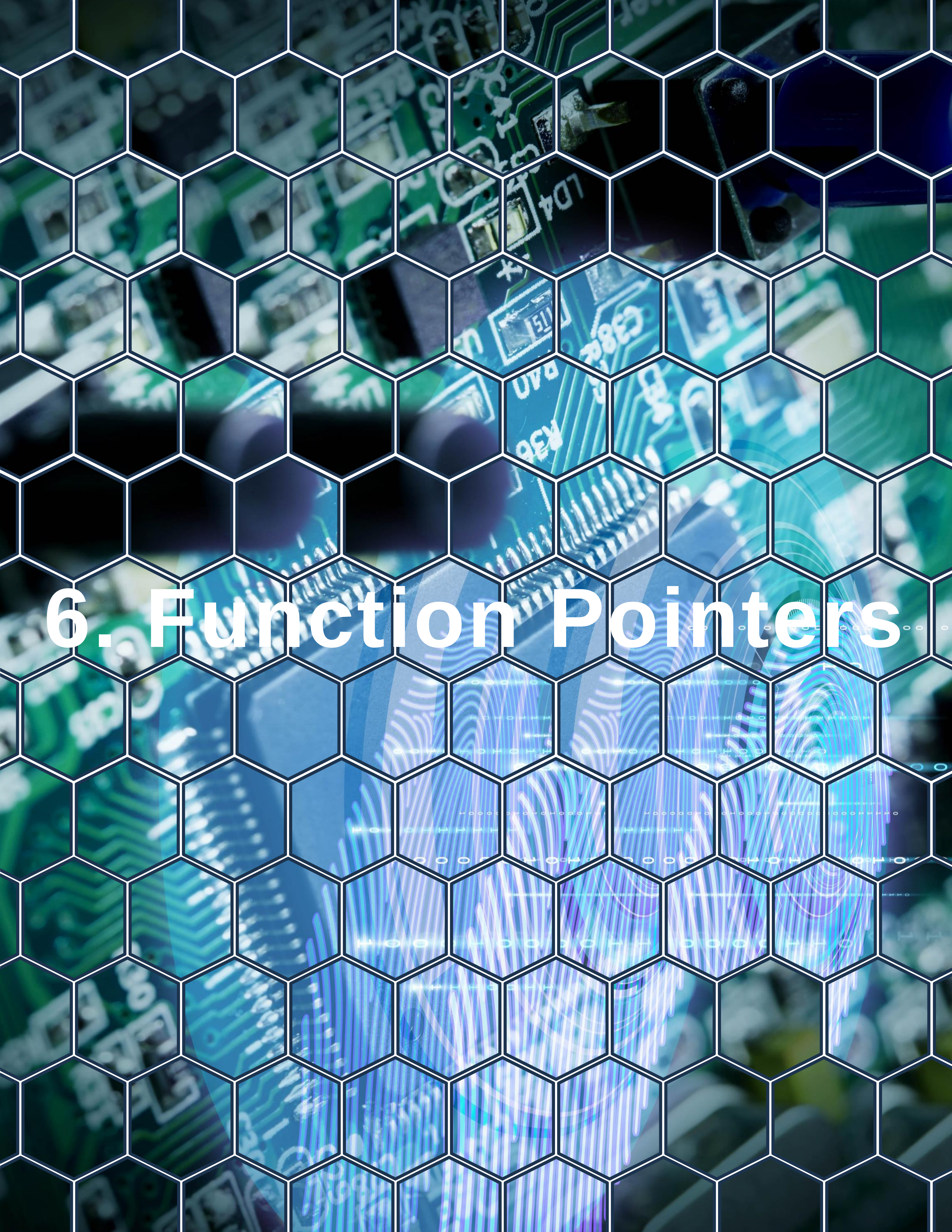
A **dangling pointer** points to a memory location that has been freed or is out of scope.

```c
int *ptr;
{
    int temp = 42;
    ptr = &temp;
}
// temp is out of scope here; ptr is now dangling
```

Use Case:

Avoid using stack memory outside its valid scope. Nullify the pointer after free().

6. Function Pointers



6. Function Pointers

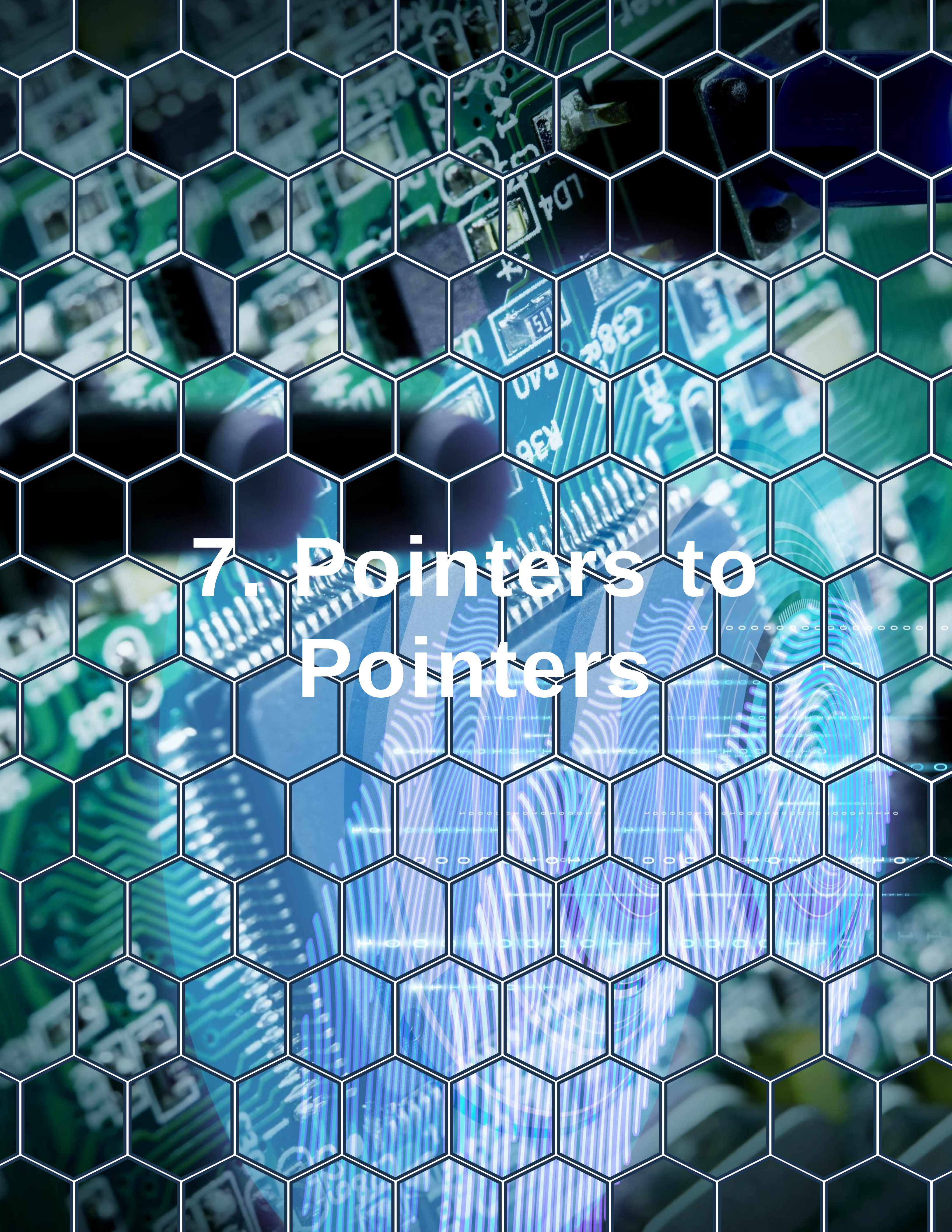
Function pointers store the address of a function. They're useful for callback implementations and building jump tables.

```c
#include <stdio.h>

void greet() {
    printf("Hello!\n");
}

int main() {
    void (*func_ptr)() = greet;
    func_ptr(); // Calls greet
    return 0;
}
```

Use Case:

Used in state machines, callback mechanisms, and hardware abstraction layers.

7. Pointers to Pointers



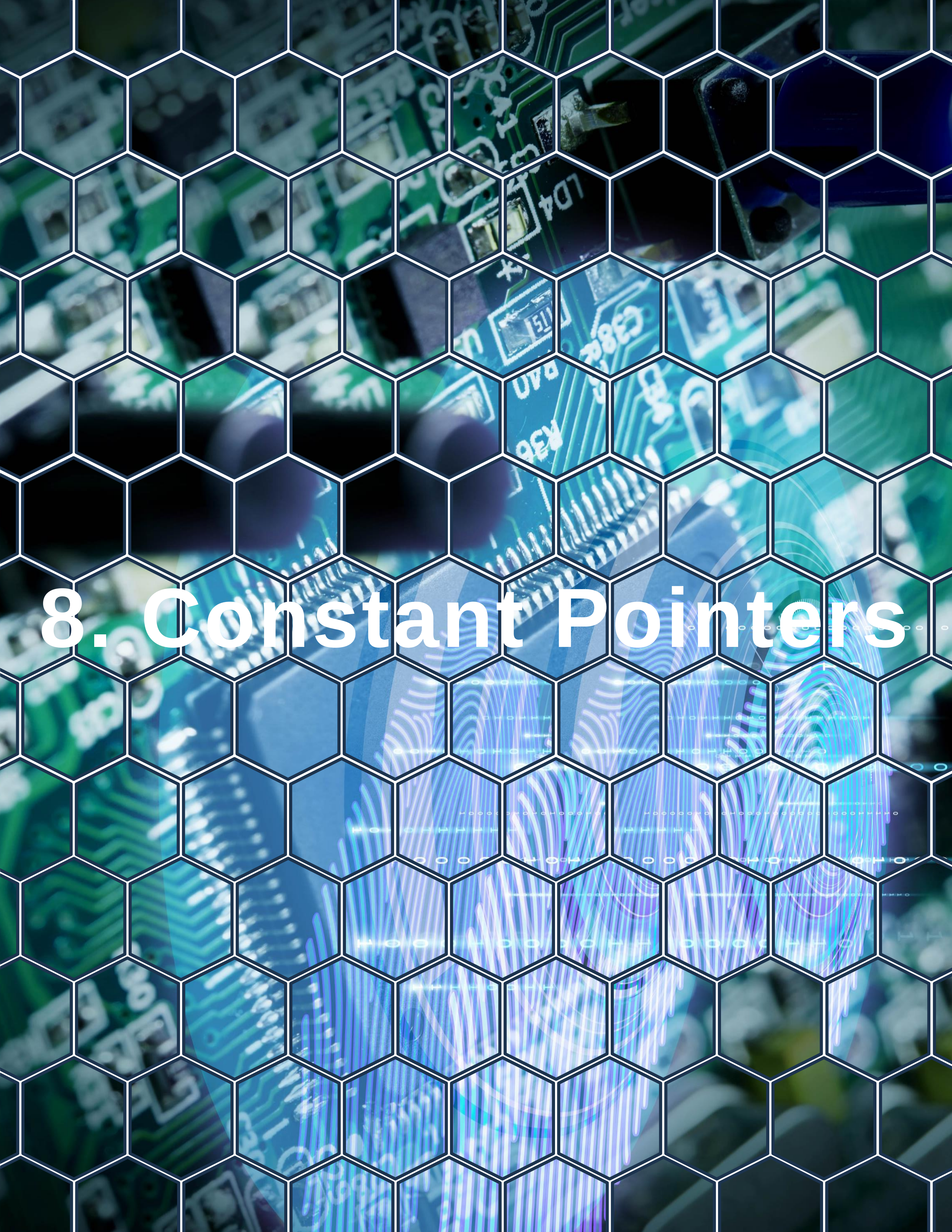
7. Pointers to Pointers

This is a pointer that points to another pointer. It's often used in dynamic memory allocation and passing multi-dimensional arrays.

```c
#include <stdio.h>

int main() {
    int x = 10;
    // ptr is a pointer to int
    int *ptr = &x;
    // pptr is a pointer to a pointer to int
    int **pptr = &ptr;
    // Access the value using double dereference
    printf("Value of x     : %d\n", x);       // Direct access
    printf("Value via *ptr : %d\n", *ptr);    // Access via pointer
    printf("Value via **pptr: %d\n", **pptr); // Access via pointer
                                              // to pointer
    return 0;
}
```

Use Case:

Used in advanced data structures and memory management.



8. Constant Pointers

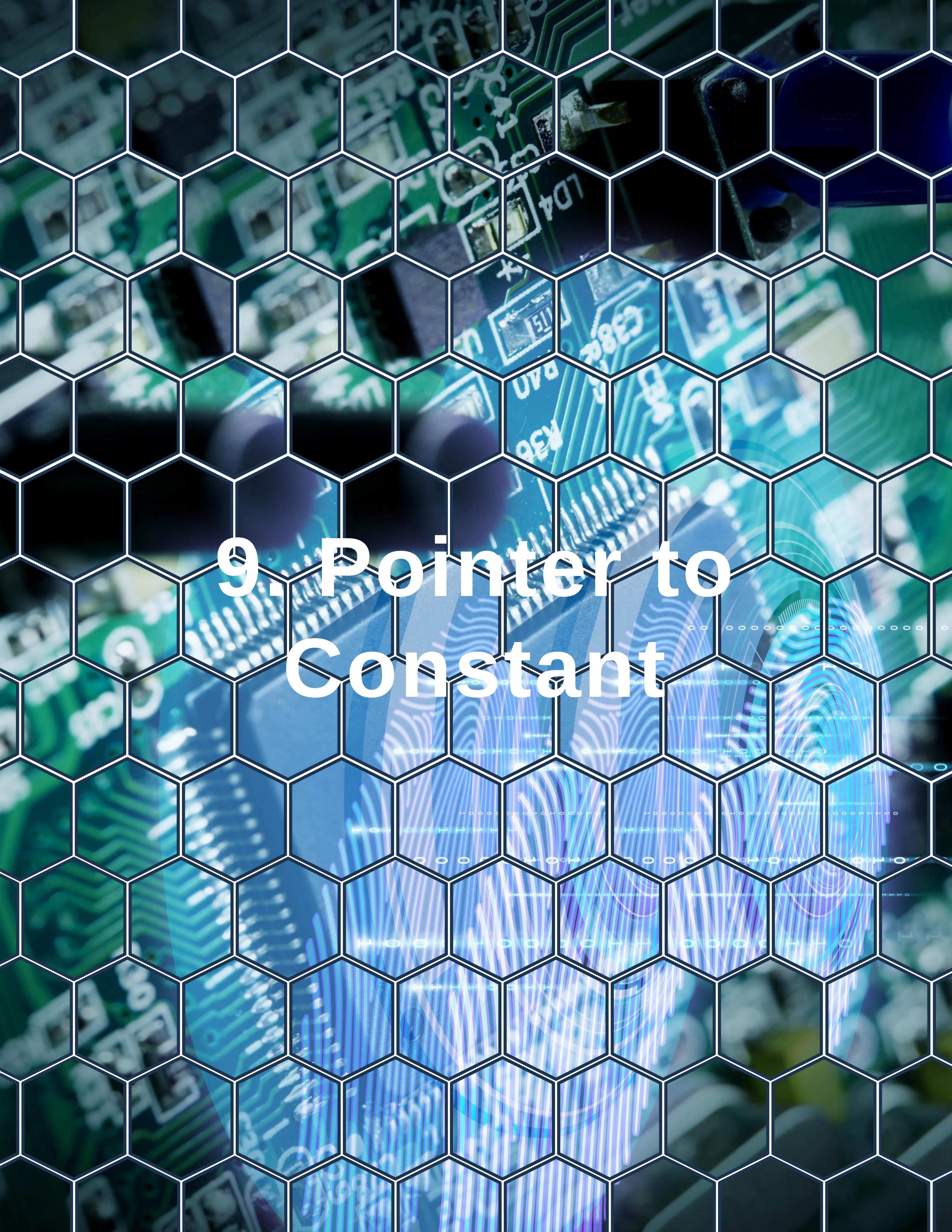
8. Constant Pointers

A **constant pointer** cannot point to another variable after initialization.

```
int x = 10, y = 20;
int *const ptr = &x;
// ptr = &y; // Error
*ptr = 30; // Allowed
```

Use Case:

When a pointer must always refer to the same memory location.



9. Pointer to Constant

9. Pointer to Constant

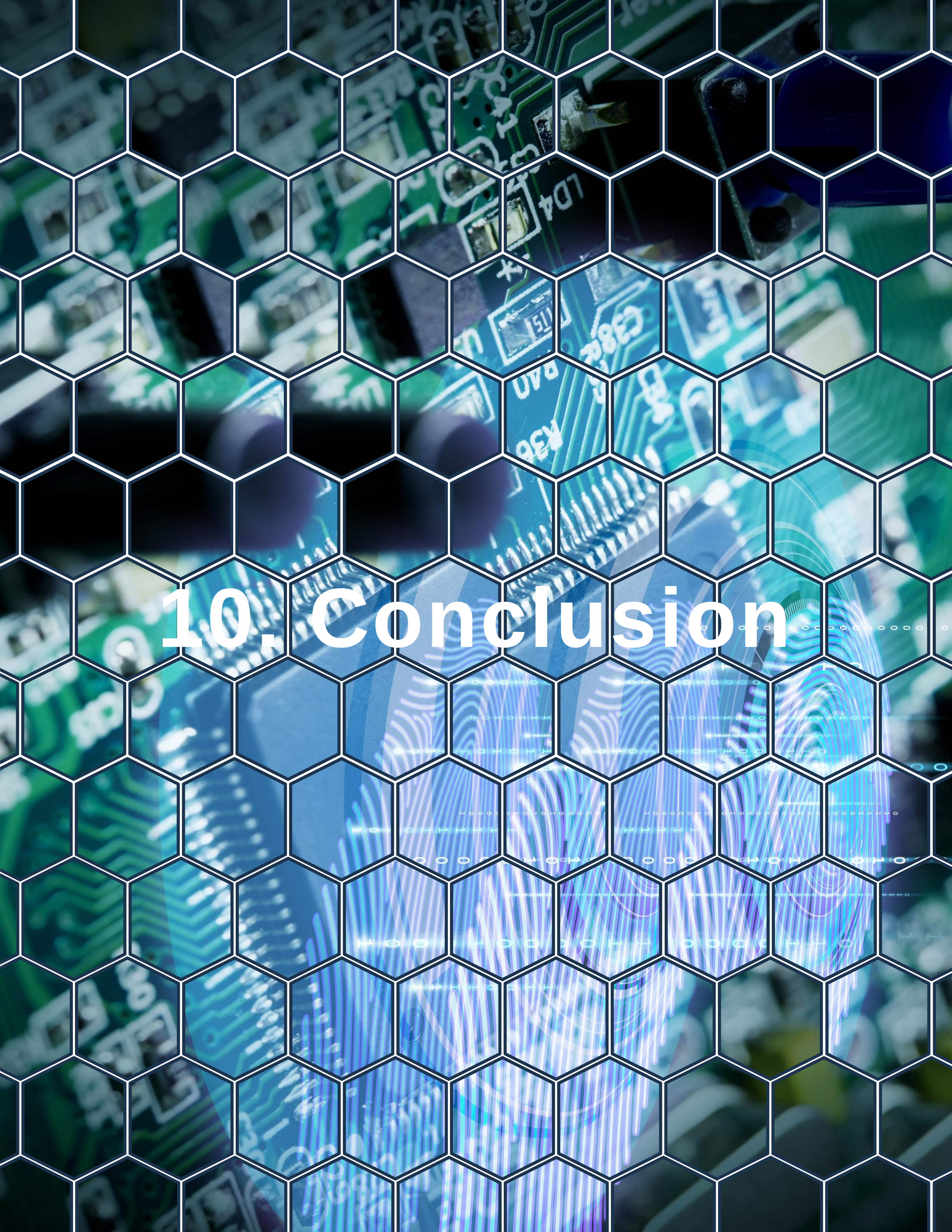
A pointer to a constant means the value being pointed to cannot be changed via the pointer.

```
int x = 10;
const int *ptr = &x;
// *ptr = 20; // Error
x = 20; // Allowed directly
```

Use Case:

Used to enforce read-only access to the pointed data.

10. Conclusion



10. Conclusion

Pointers are fundamental to system-level programming and mastering them is critical for writing high-performance C code. From simple address referencing to function callbacks and memory safety practices, the many types of pointers in C empower developers with flexibility and control—but demand responsibility and discipline. Always use pointers wisely: initialize them, avoid wild and dangling pointers, and use const where appropriate.