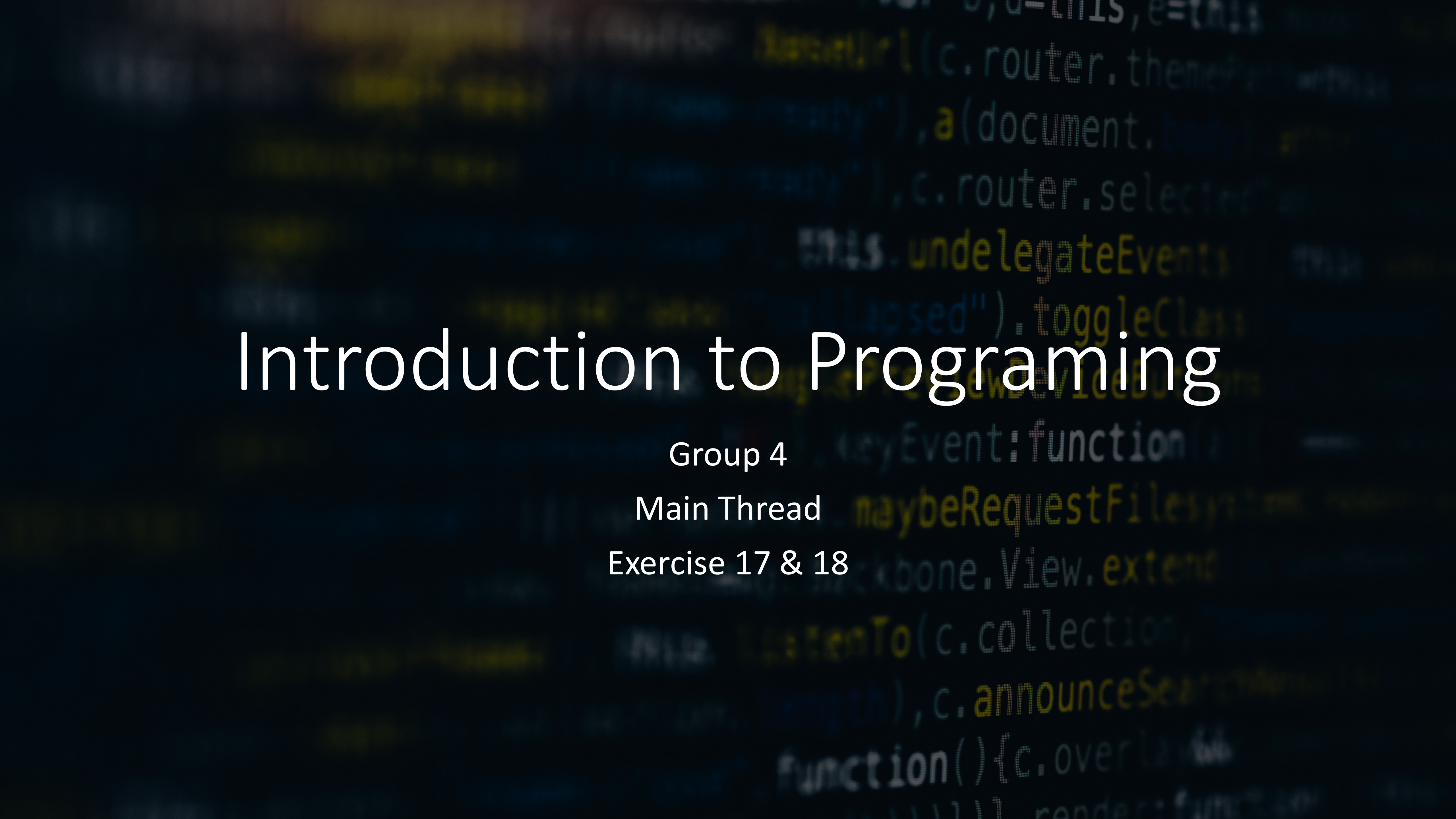


Introduction to Programing

Group 4

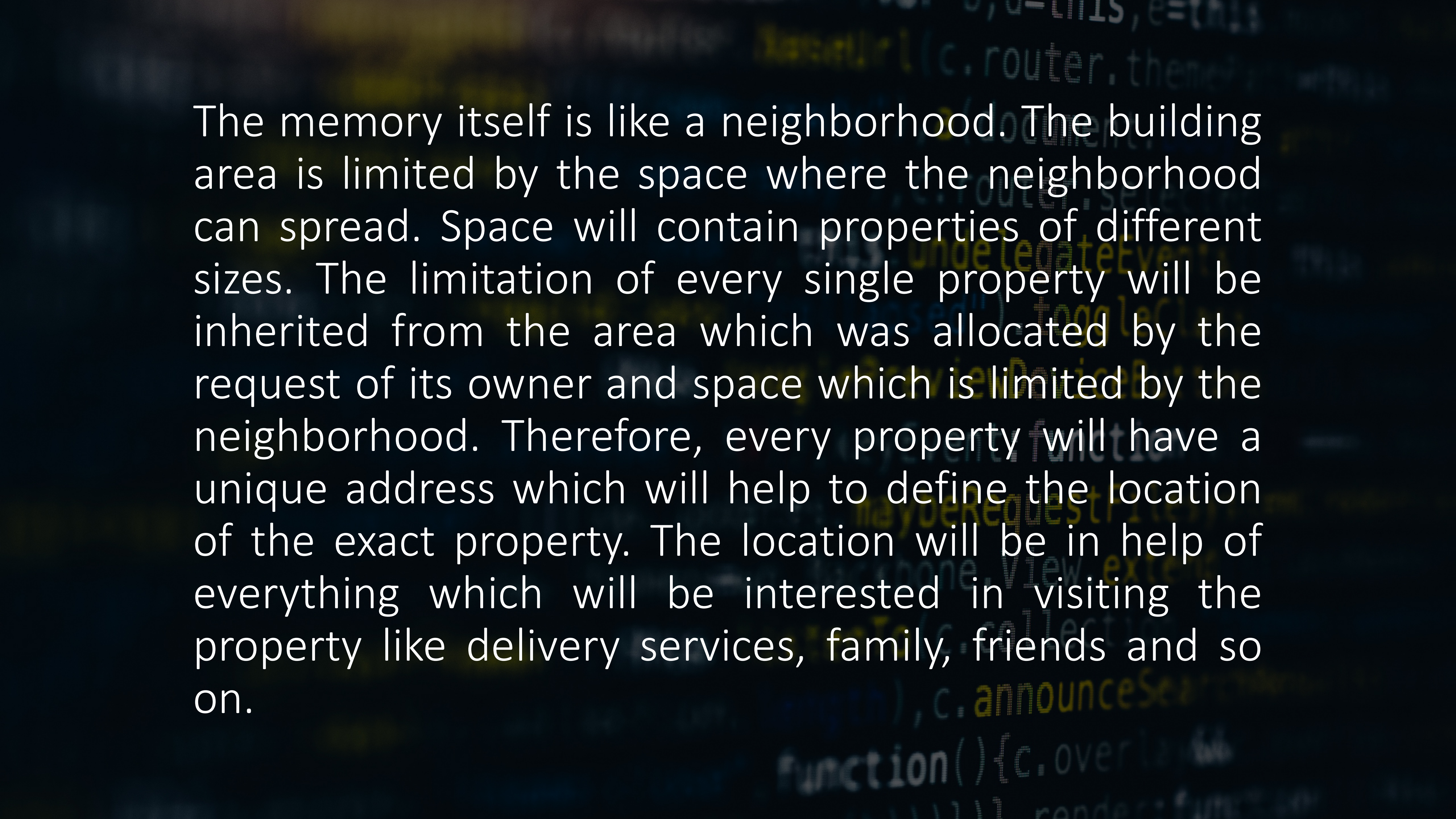
Main Thread

Exercise 17 & 18

POINTERS

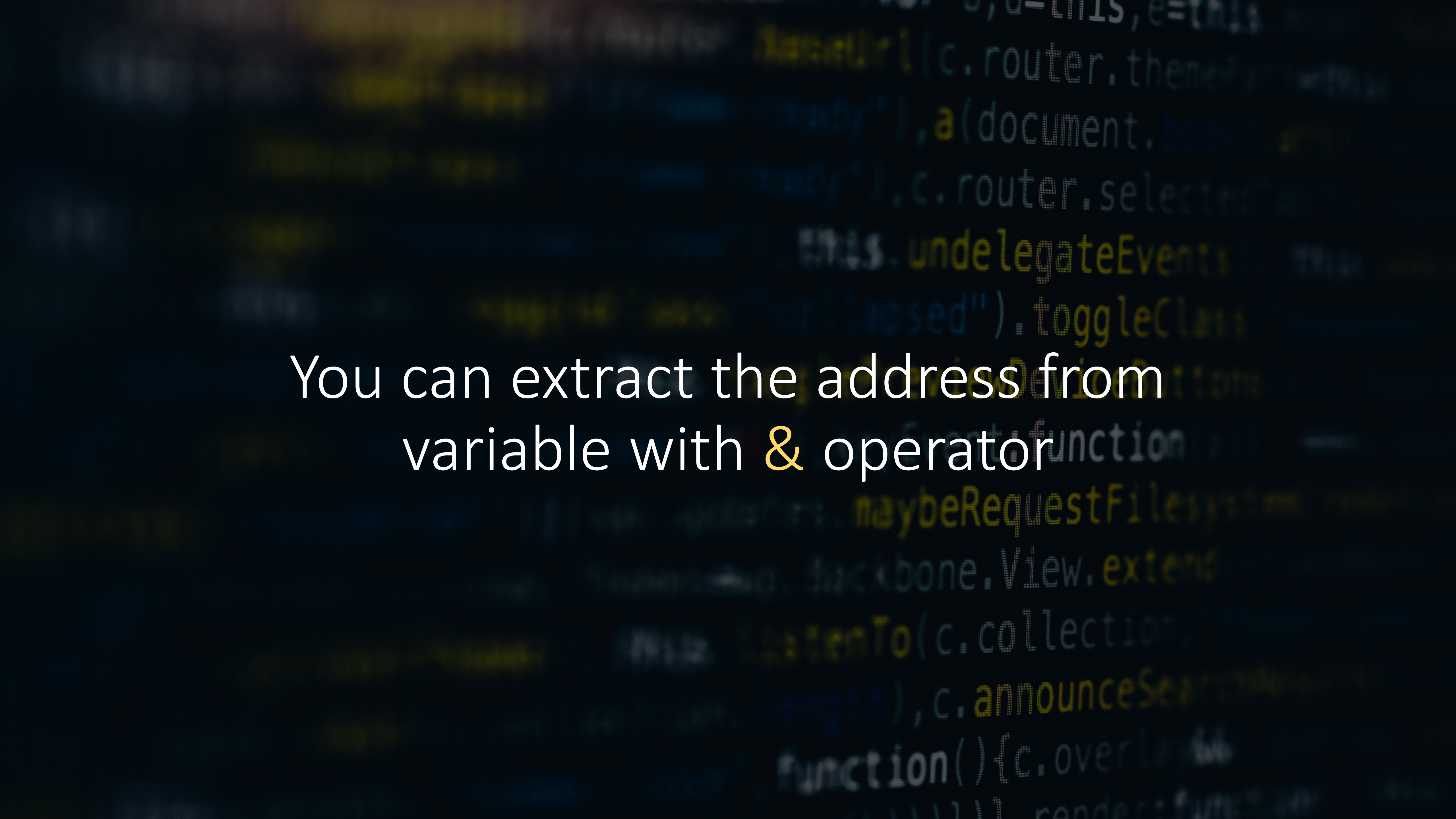
Pointer is a variable that stores the memory address as its value.

Memory management is the most important key in every program.

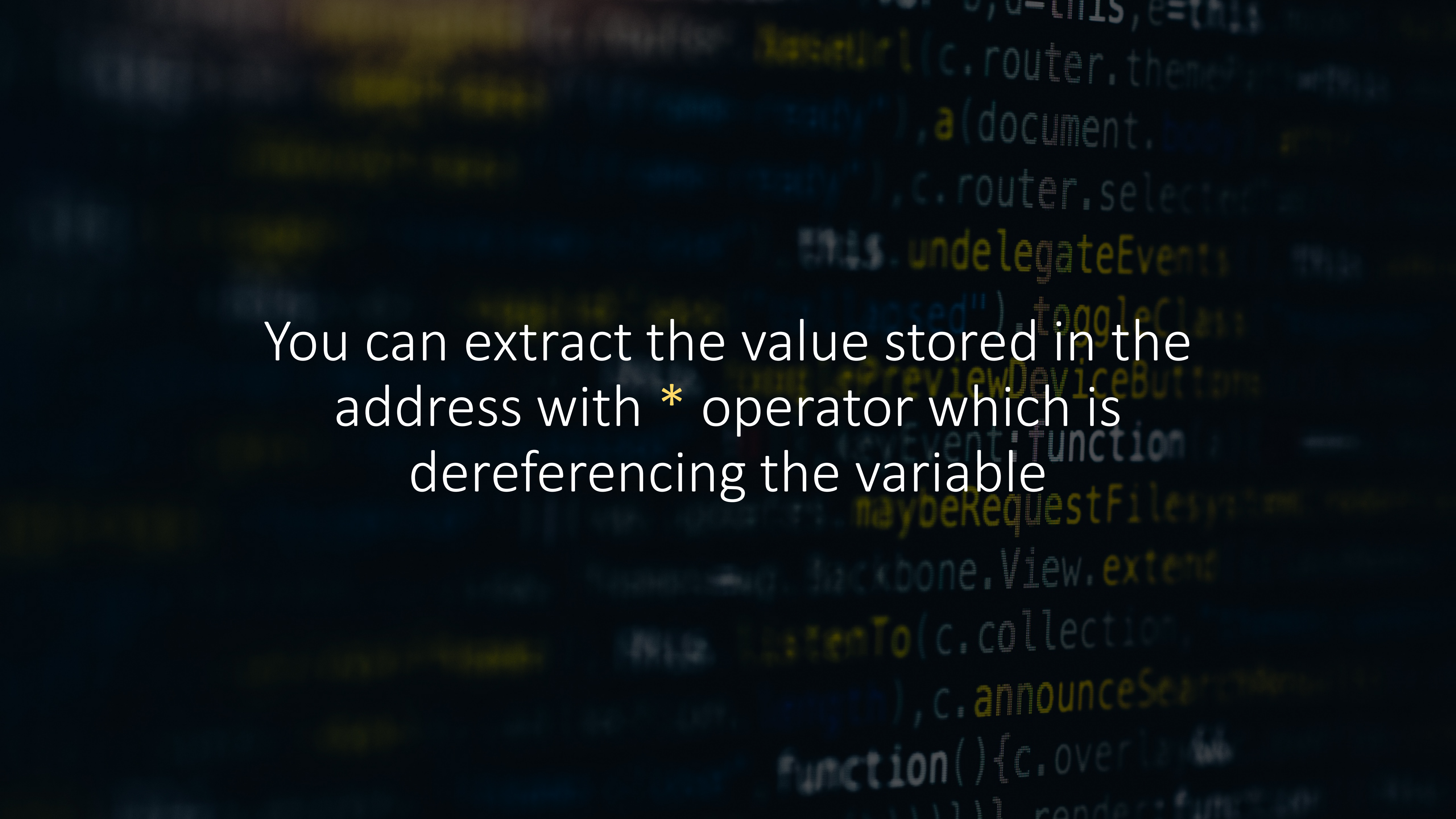


The memory itself is like a neighborhood. The building area is limited by the space where the neighborhood can spread. Space will contain properties of different sizes. The limitation of every single property will be inherited from the area which was allocated by the request of its owner and space which is limited by the neighborhood. Therefore, every property will have a unique address which will help to define the location of the exact property. The location will be in help of everything which will be interested in visiting the property like delivery services, family, friends and so on.

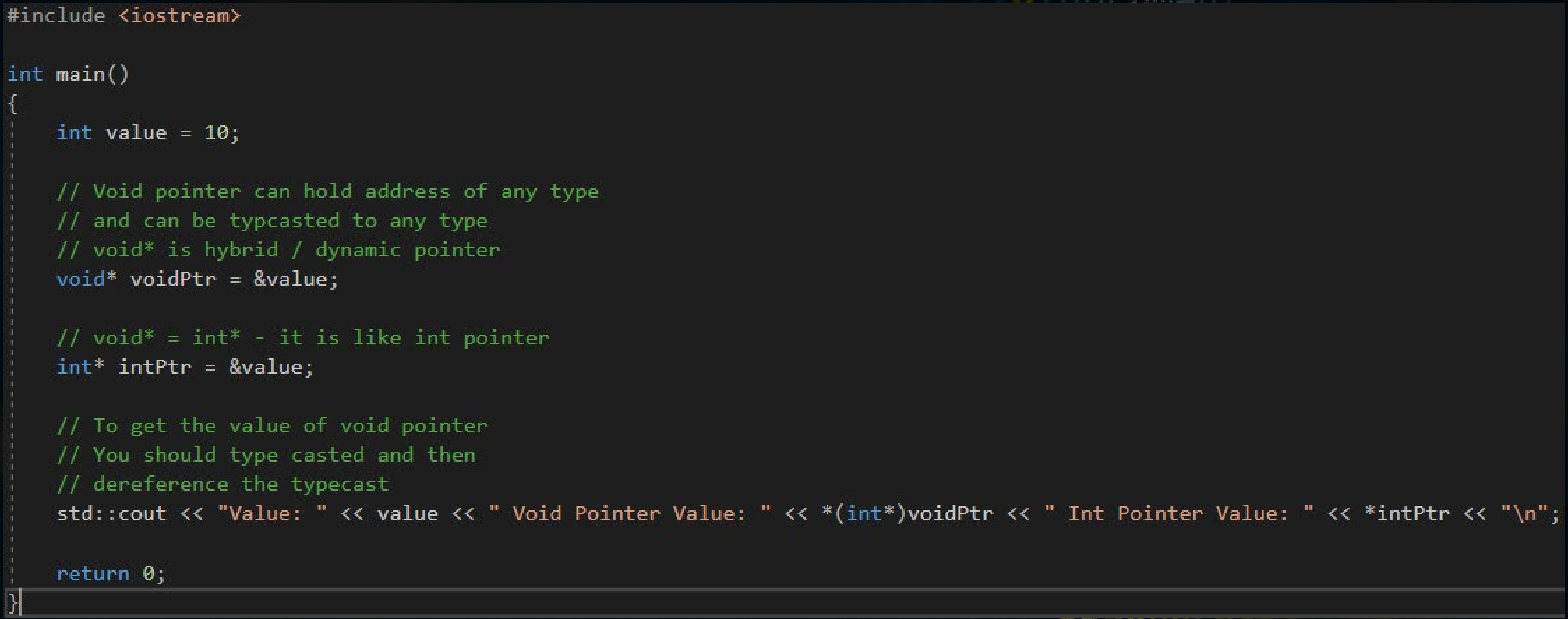
The address is used to pinpointing the location of memory



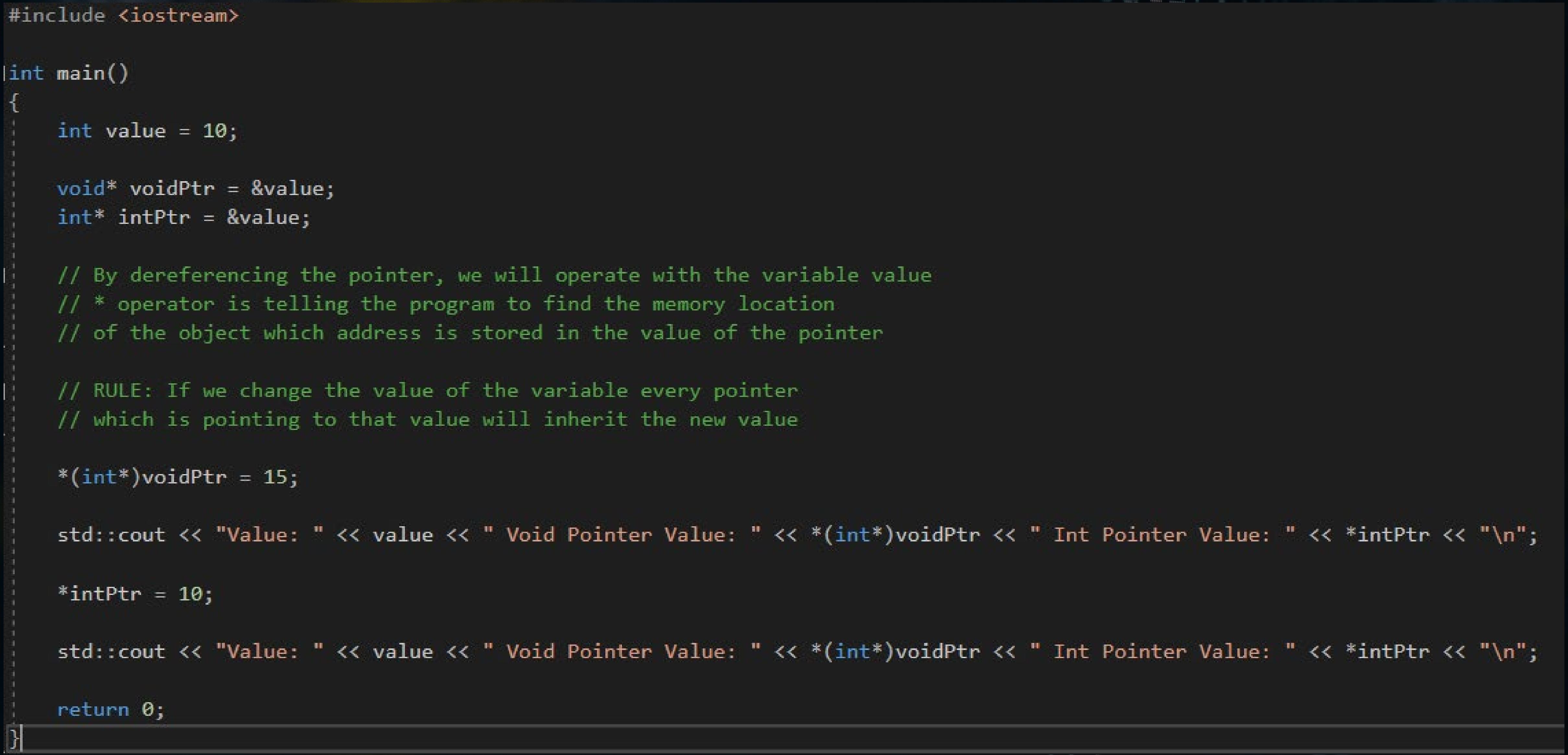
You can extract the address from variable with & operator



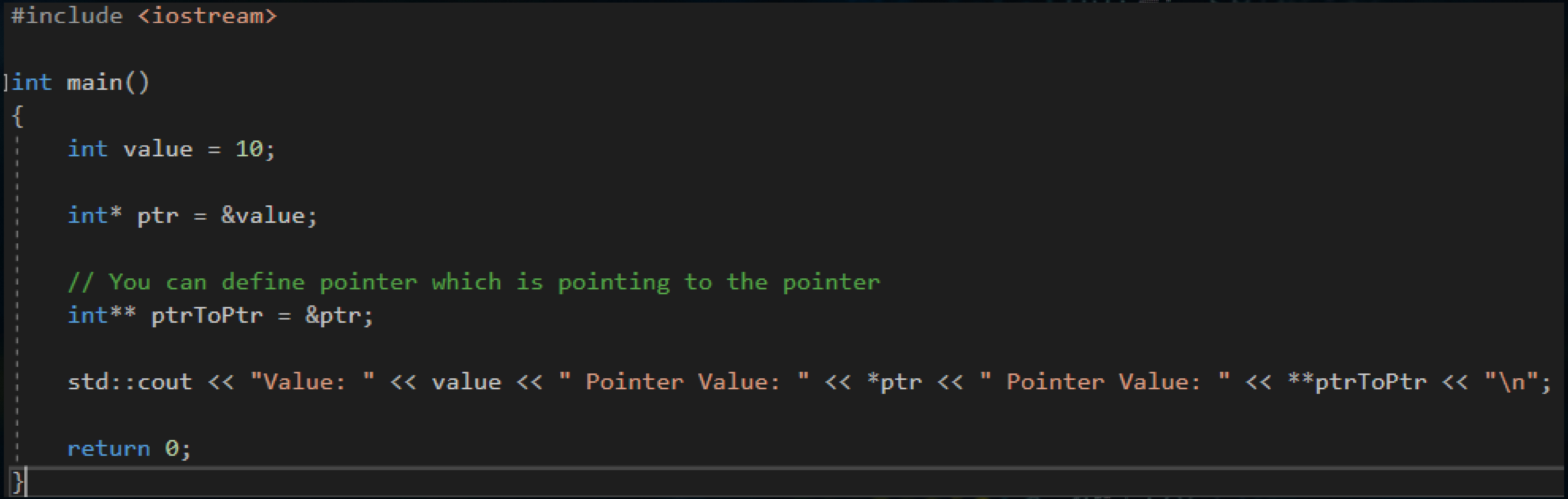
You can extract the value stored in the address with * operator which is dereferencing the variable



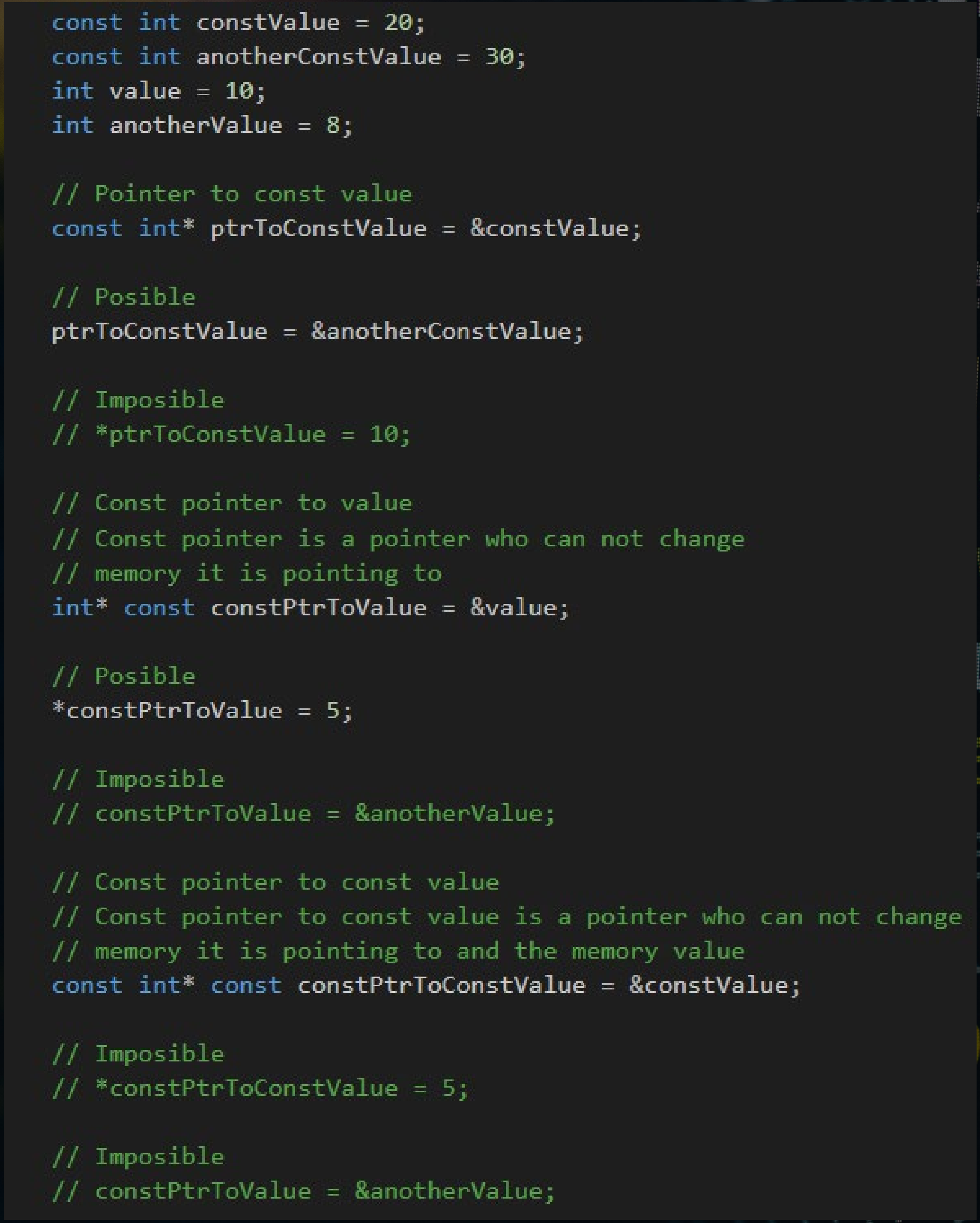
```cpp
#include <iostream>

int main()
{
    int value = 10;

    // Void pointer can hold address of any type
    // and can be typcasted to any type
    // void* is hybrid / dynamic pointer
    void* voidPtr = &value;

    // void* = int* - it is like int pointer
    int* intPtr = &value;

    // To get the value of void pointer
    // You should type casted and then
    // dereference the typecast
    std::cout << "Value: " << value << " Void Pointer Value: " << *(int*)voidPtr << " Int Pointer Value: " << *intPtr << "\n";

    return 0;
}
```



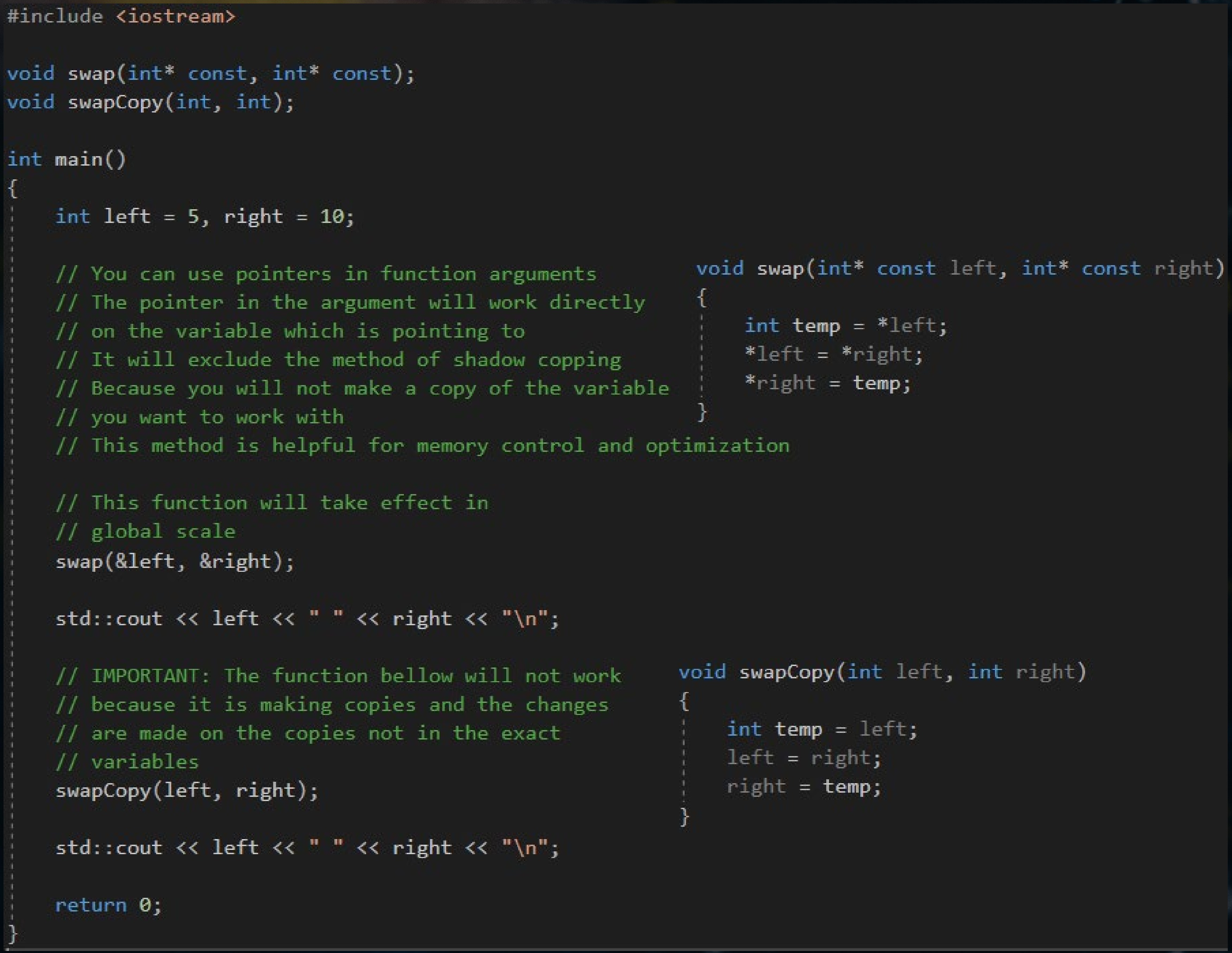
```cpp
#include <iostream>

int main()
{
    int value = 10;

    void* voidPtr = &value;
    int* intPtr = &value;

    // By dereferencing the pointer, we will operate with the variable value
    // * operator is telling the program to find the memory location
    // of the object which address is stored in the value of the pointer

    // RULE: If we change the value of the variable every pointer
    // which is pointing to that value will inherit the new value

    *(int*)voidPtr = 15;

    std::cout << "Value: " << value << " Void Pointer Value: " << *(int*)voidPtr << " Int Pointer Value: " << *intPtr << "\n";

    *intPtr = 10;

    std::cout << "Value: " << value << " Void Pointer Value: " << *(int*)voidPtr << " Int Pointer Value: " << *intPtr << "\n";

    return 0;
}
```



```cpp
#include <iostream>

int main()
{
    int value = 10;

    int* ptr = &value;

    // You can define pointer which is pointing to the pointer
    int** ptrToPtr = &ptr;

    std::cout << "Value: " << value << " Pointer Value: " << *ptr << " Pointer Value: " << **ptrToPtr << "\n";

    return 0;
}
```



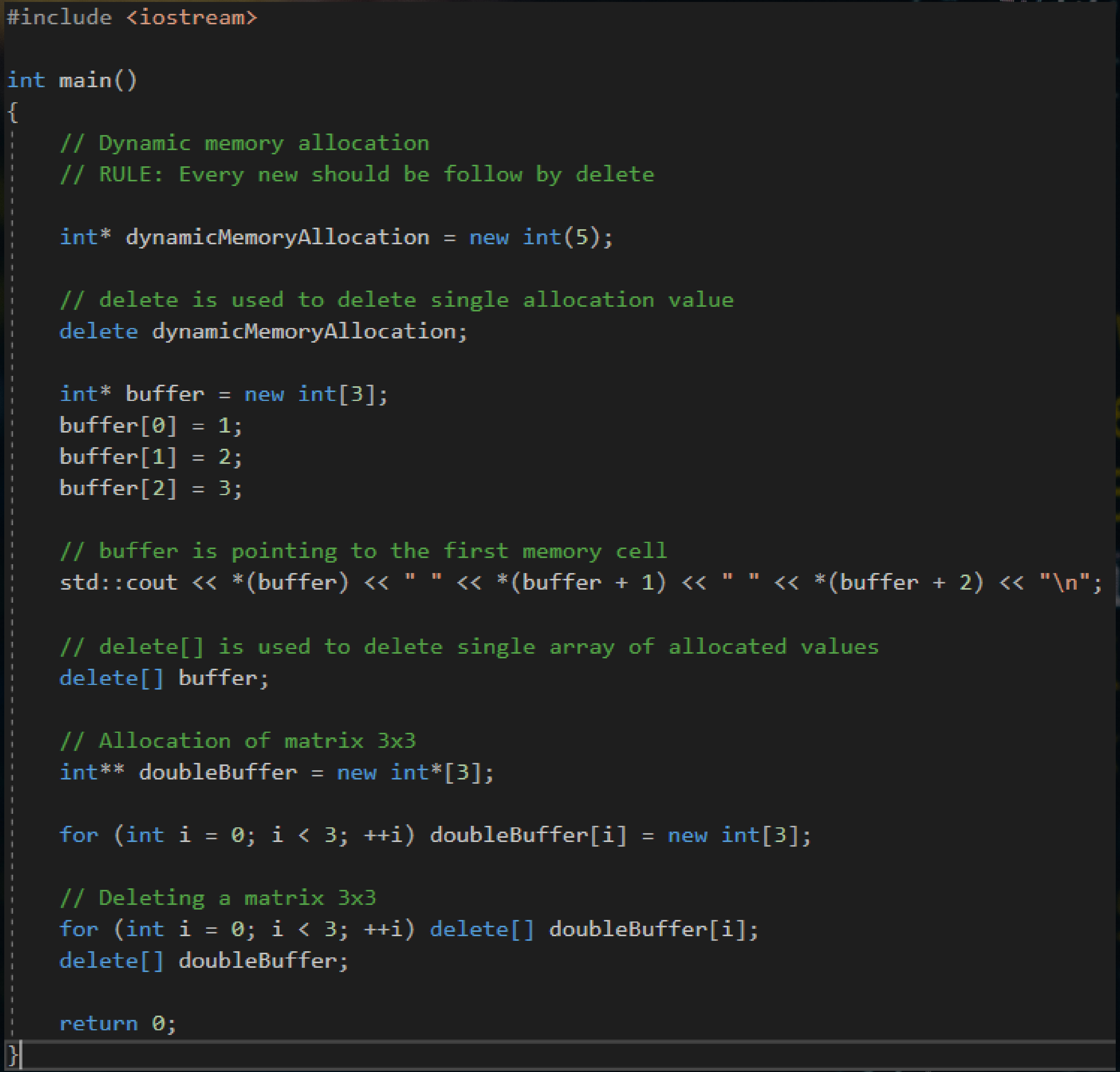
```cpp
const int constValue = 20;
const int anotherConstValue = 30;
int value = 10;
int anotherValue = 8;

// Pointer to const value
const int* ptrToConstValue = &constValue;

// Posible
ptrToConstValue = &anotherConstValue;

// Imposible
// *ptrToConstValue = 10;

// Const pointer to value
// Const pointer is a pointer who can not change
// memory it is pointing to
int* const constPtrToValue = &value;

// Posible
*constPtrToValue = 5;

// Imposible
// constPtrToValue = &anotherValue;

// Const pointer to const value
// Const pointer to const value is a pointer who can not change
// memory it is pointing to and the memory value
const int* const constPtrToConstValue = &constValue;

// Imposible
// *constPtrToConstValue = 5;

// Imposible
// constPtrToValue = &anotherValue;
```



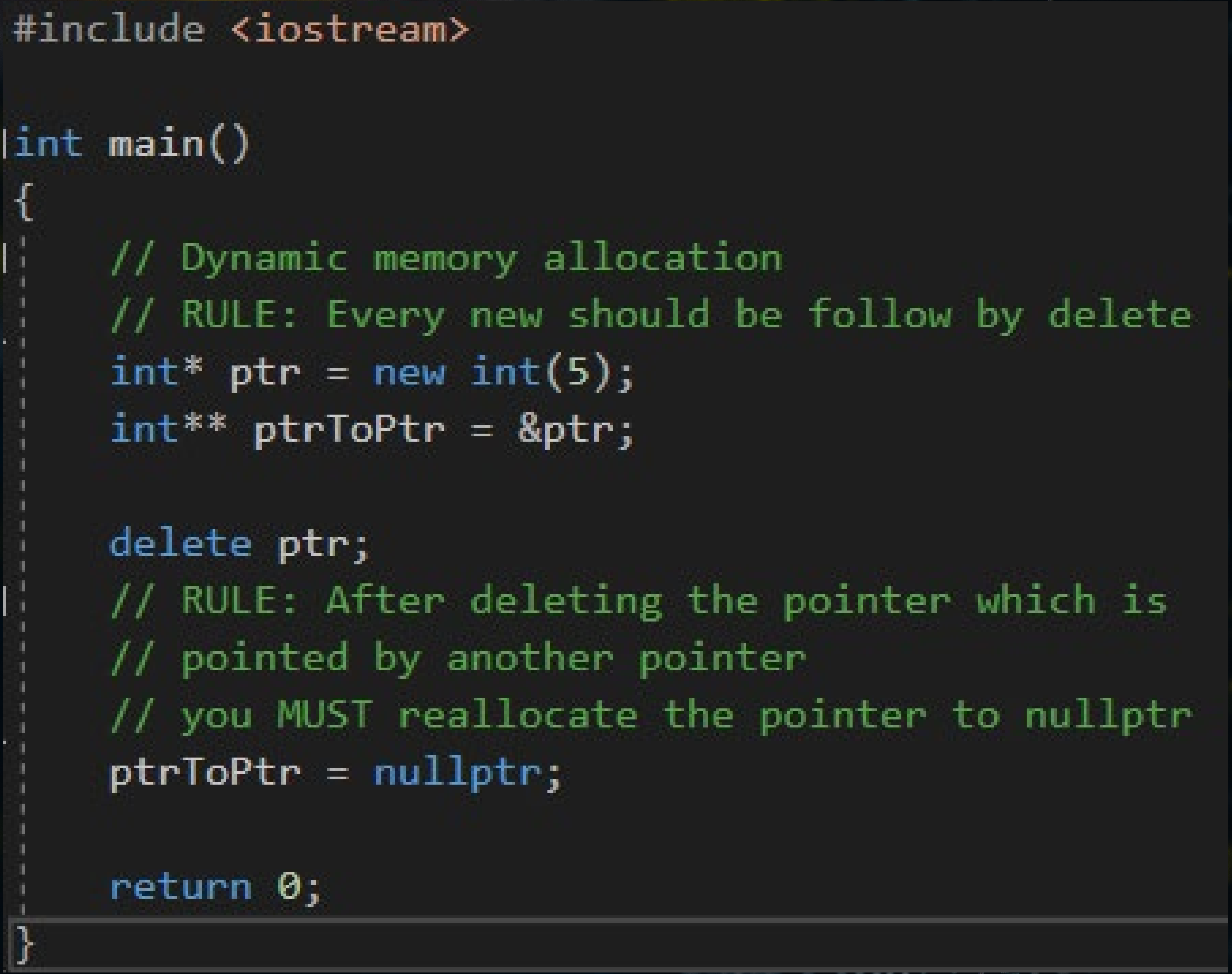
```cpp
#include <iostream>

void swap(int* const, int* const);
void swapCopy(int, int);

int main()
{
    int left = 5, right = 10;

    // You can use pointers in function arguments
    // The pointer in the argument will work directly
    // on the variable which is pointing to
    // It will exclude the method of shadow copping
    // Because you will not make a copy of the variable
    // you want to work with
    // This method is helpful for memory control and optimization

    // This function will take effect in
    // global scale
    swap(&left, &right);

    std::cout << left << " " << right << "\n";

    // IMPORTANT: The function bellow will not work
    // because it is making copies and the changes
    // are made on the copies not in the exact
    // variables
    swapCopy(left, right);

    std::cout << left << " " << right << "\n";

    return 0;
}
```

```cpp
void swap(int* const left, int* const right)
{
    int temp = *left;
    *left = *right;
    *right = temp;
}
```

```cpp
void swapCopy(int left, int right)
{
    int temp = left;
    left = right;
    right = temp;
}
```

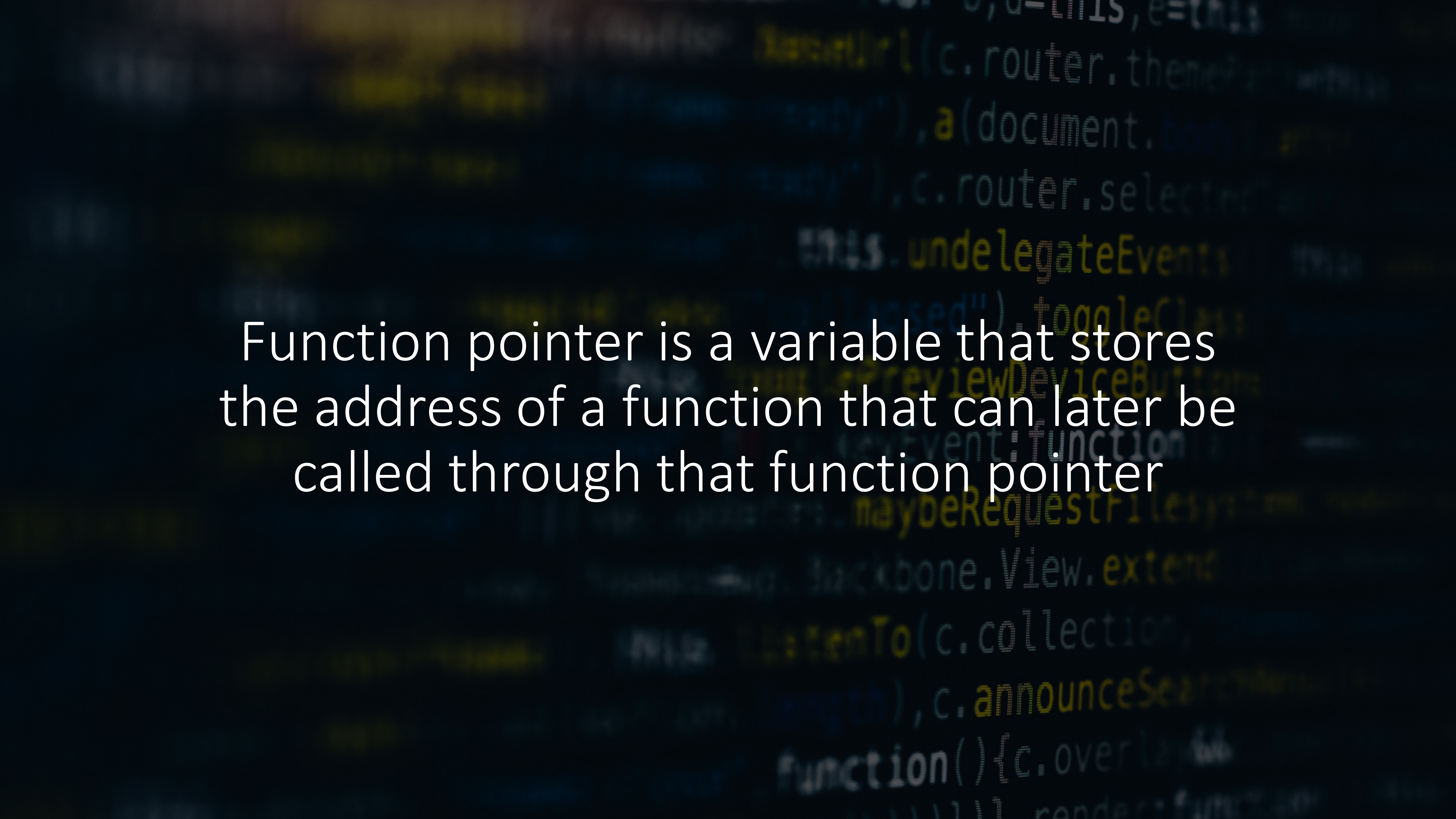


```cpp
#include <iostream>

int main()
{
    // Dynamic memory allocation
    // RULE: Every new should be follow by delete

    int* dynamicMemoryAllocation = new int(5);

    // delete is used to delete single allocation value
    delete dynamicMemoryAllocation;

    int* buffer = new int[3];
    buffer[0] = 1;
    buffer[1] = 2;
    buffer[2] = 3;

    // buffer is pointing to the first memory cell
    std::cout << *(buffer) << " " << *(buffer + 1) << " " << *(buffer + 2) << "\n";

    // delete[] is used to delete single array of allocated values
    delete[] buffer;

    // Allocation of matrix 3x3
    int** doubleBuffer = new int*[3];

    for (int i = 0; i < 3; ++i) doubleBuffer[i] = new int[3];

    // Deleting a matrix 3x3
    for (int i = 0; i < 3; ++i) delete[] doubleBuffer[i];
    delete[] doubleBuffer;

    return 0;
}
```

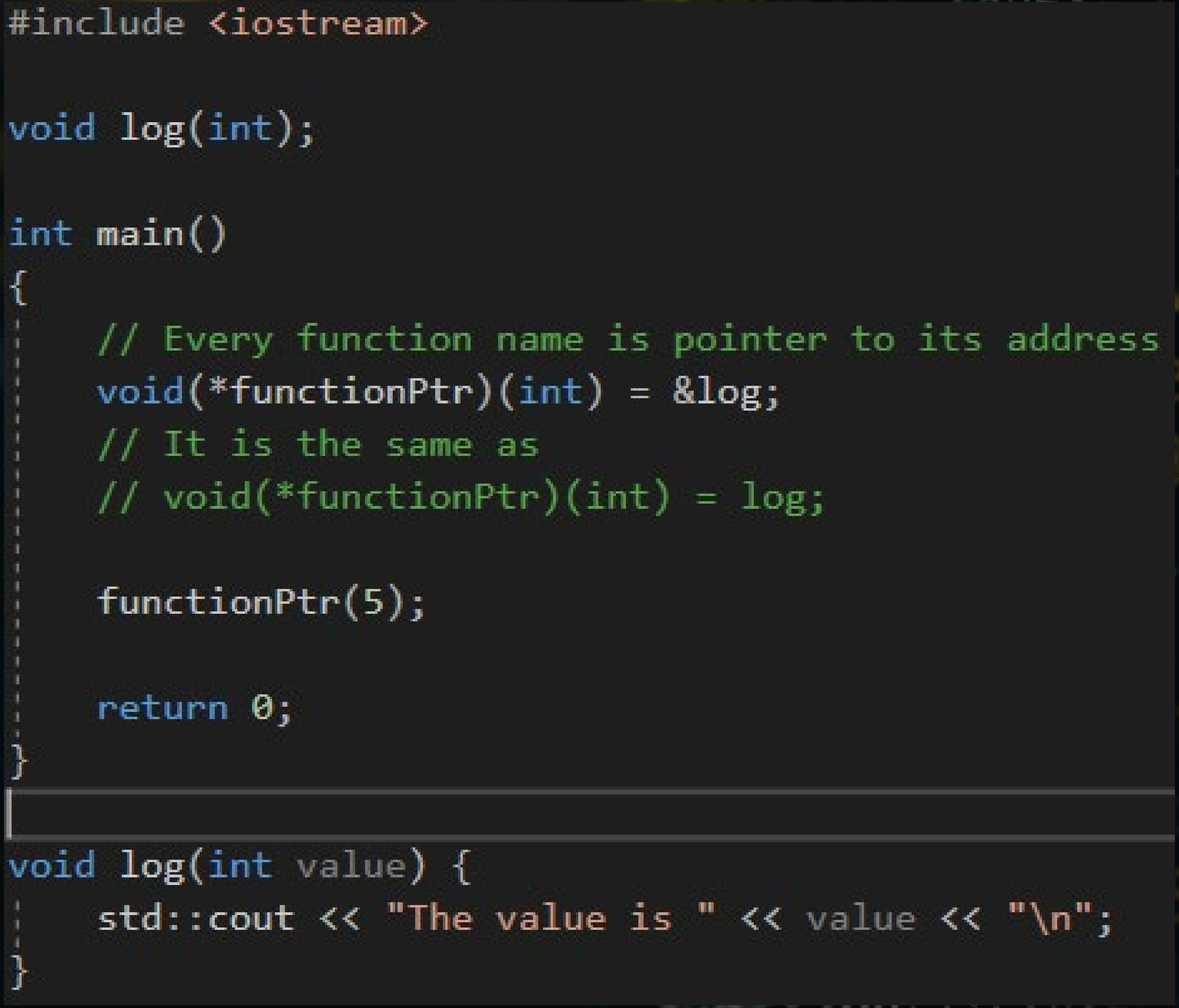


```cpp
#include <iostream>

int main()
{
    // Dynamic memory allocation
    // RULE: Every new should be follow by delete
    int* ptr = new int(5);
    int** ptrToPtr = &ptr;

    delete ptr;
    // RULE: After deleting the pointer which is
    // pointed by another pointer
    // you MUST reallocate the pointer to nullptr
    ptrToPtr = nullptr;

    return 0;
}
```

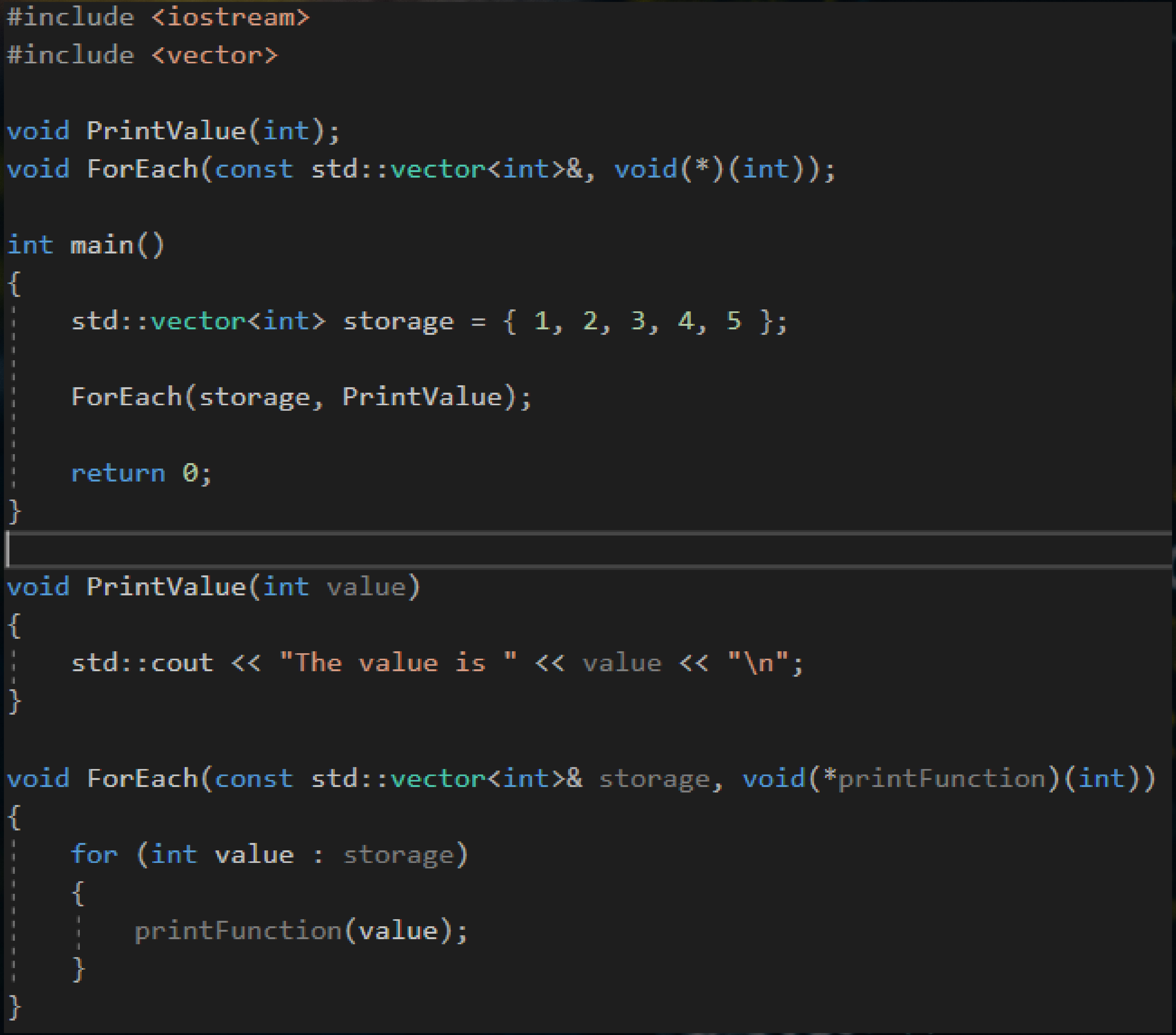
Function Pointers



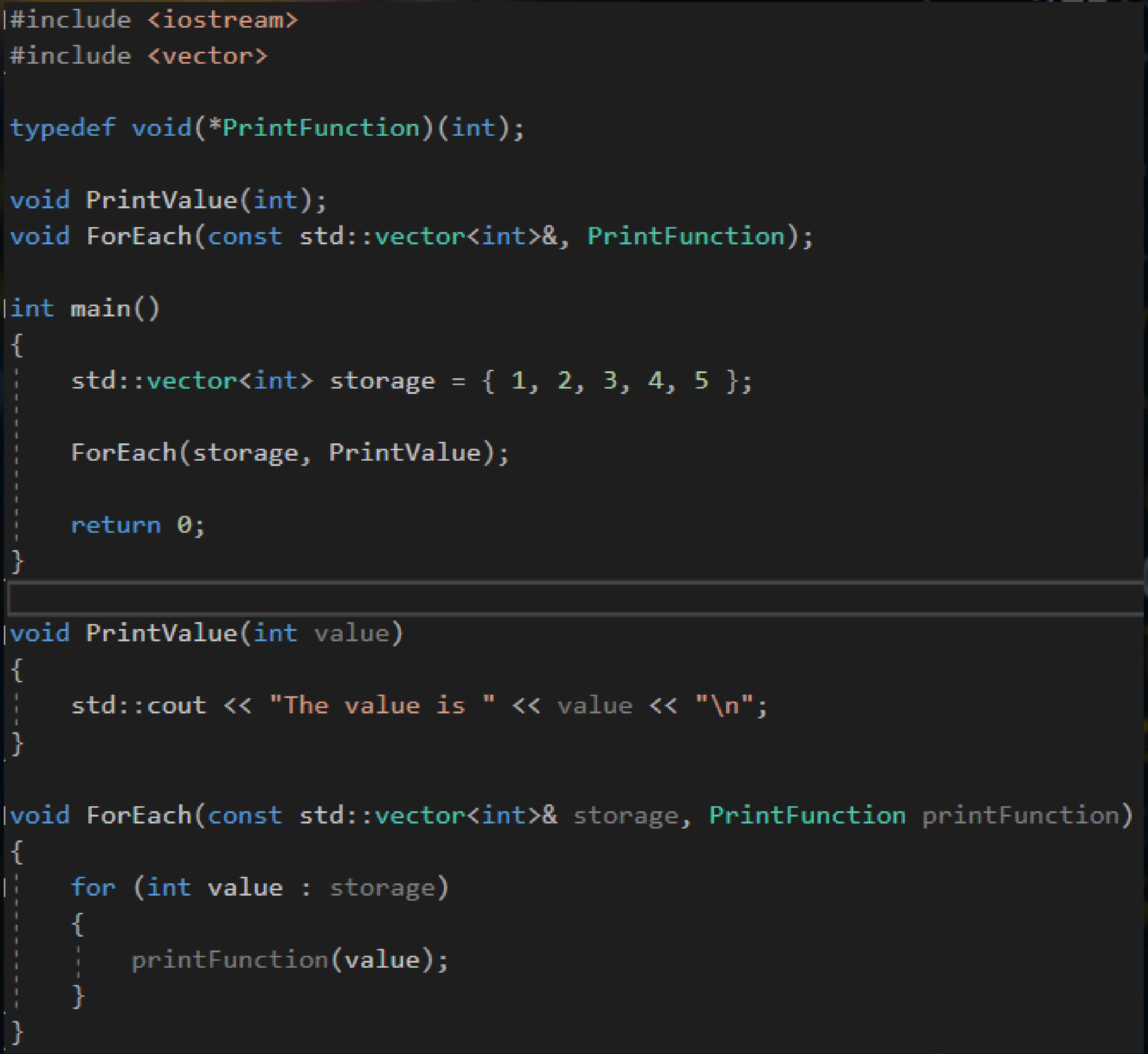
Function pointer is a variable that stores the address of a function that can later be called through that function pointer



```cpp
#include <iostream>

void log(int);

int main()
{
    // Every function name is pointer to its address
    void(*functionPtr)(int) = &log;
    // It is the same as
    // void(*functionPtr)(int) = log;

    functionPtr(5);

    return 0;
}

void log(int value) {
    std::cout << "The value is " << value << "\n";
}
```



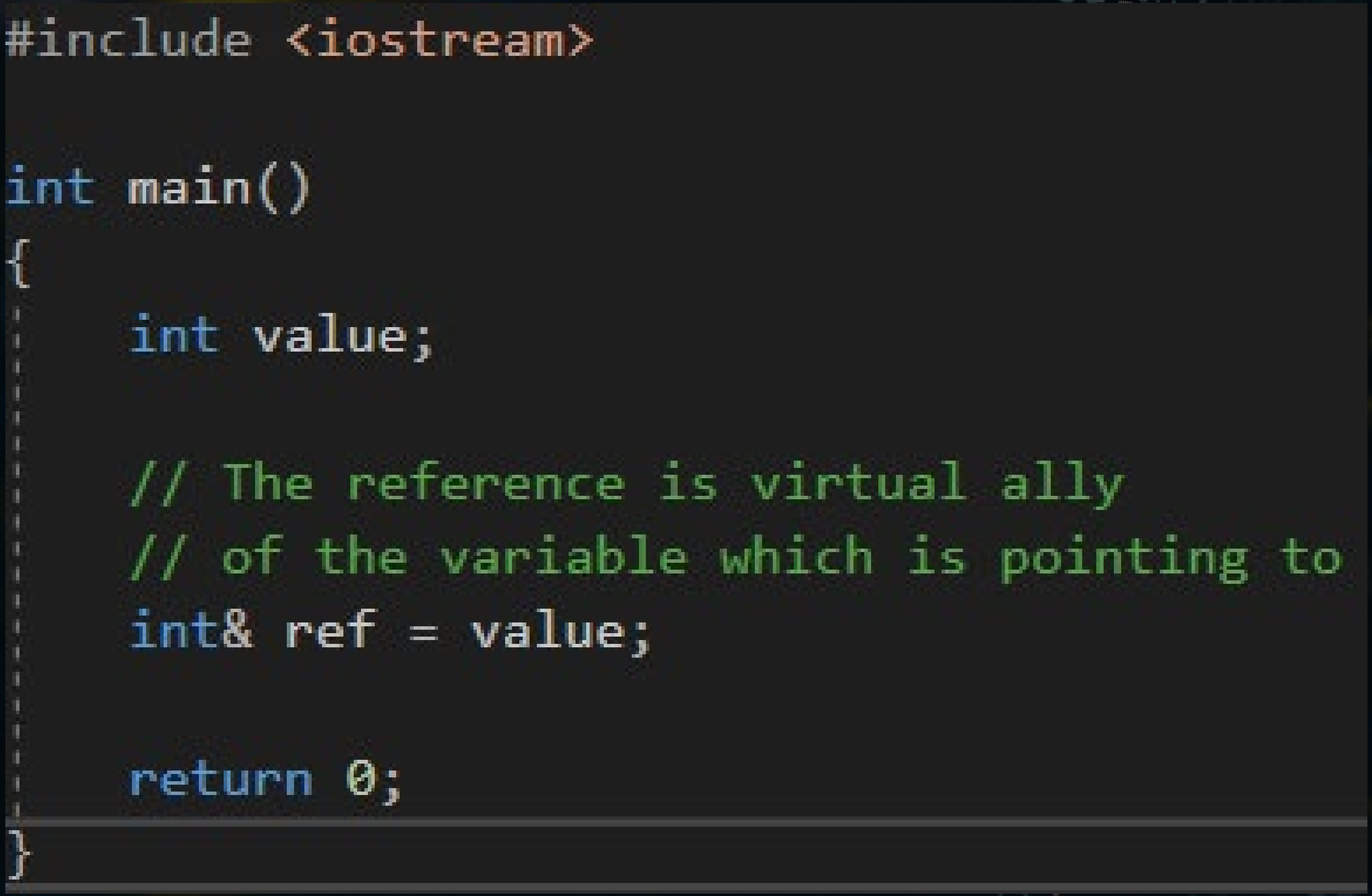
```cpp
#include <iostream>
#include <vector>

void PrintValue(int);
void ForEach(const std::vector<int>&, void(*)(int));

int main()
{
    std::vector<int> storage = { 1, 2, 3, 4, 5 };

    ForEach(storage, PrintValue);

    return 0;
}

void PrintValue(int value)
{
    std::cout << "The value is " << value << "\n";
}

void ForEach(const std::vector<int>& storage, void(*printFunction)(int))
{
    for (int value : storage)
    {
        printFunction(value);
    }
}
```


```cpp
#include <iostream>
#include <vector>

typedef void(*PrintFunction)(int);

void PrintValue(int);
void ForEach(const std::vector<int>&, PrintFunction);

int main()
{
    std::vector<int> storage = { 1, 2, 3, 4, 5 };

    ForEach(storage, PrintValue);

    return 0;
}

void PrintValue(int value)
{
    std::cout << "The value is " << value << "\n";
}

void ForEach(const std::vector<int>& storage, PrintFunction printFunction)
{
    for (int value : storage)
    {
        printFunction(value);
    }
}
```

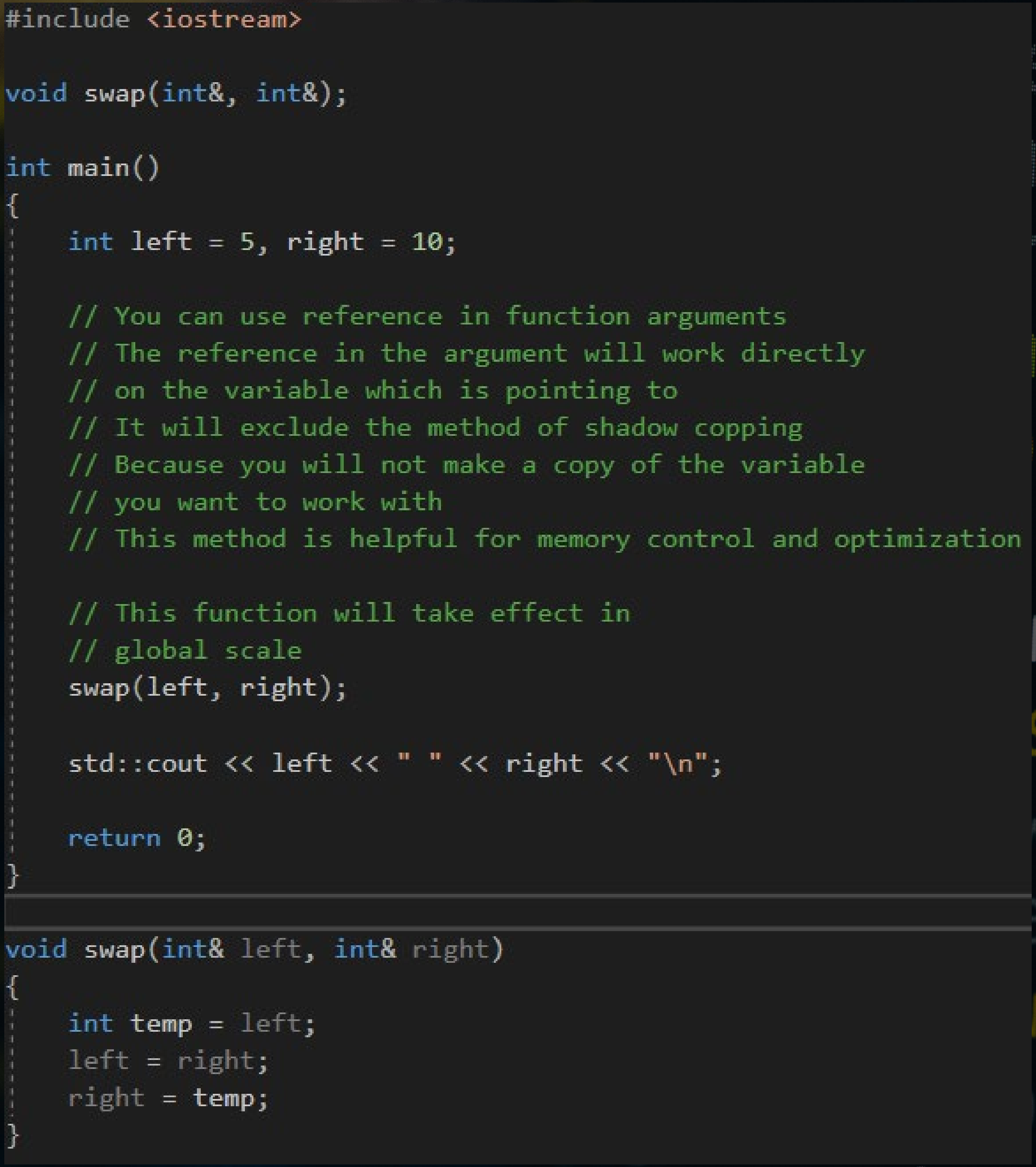
REFERENCE

Reference is alternative name for an existing variable

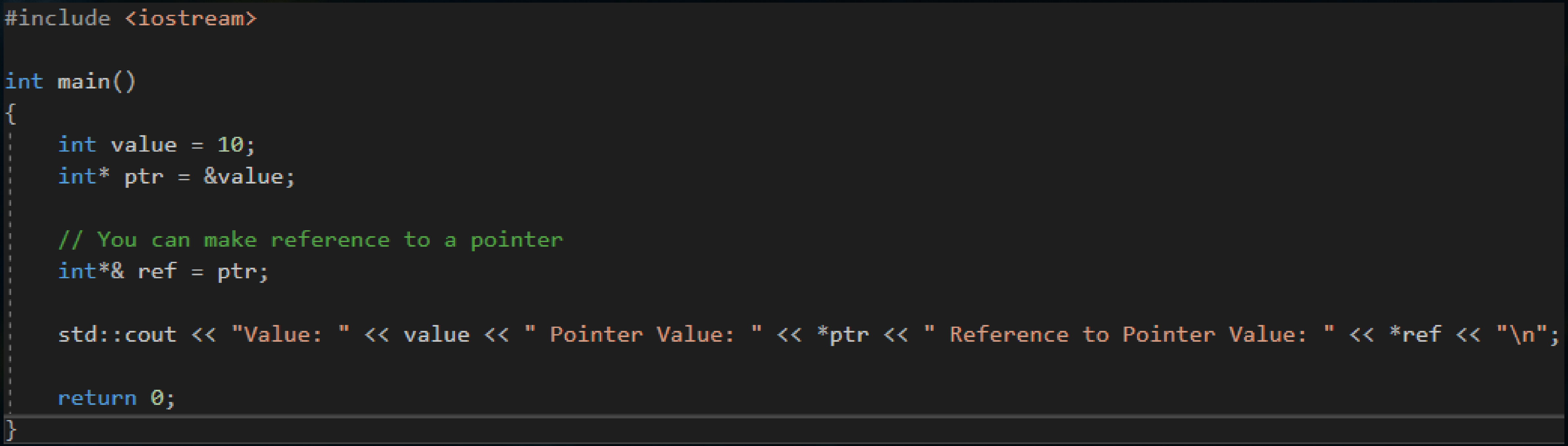
Reference is extension of pointers. The main difference is that reference will not occupy memory



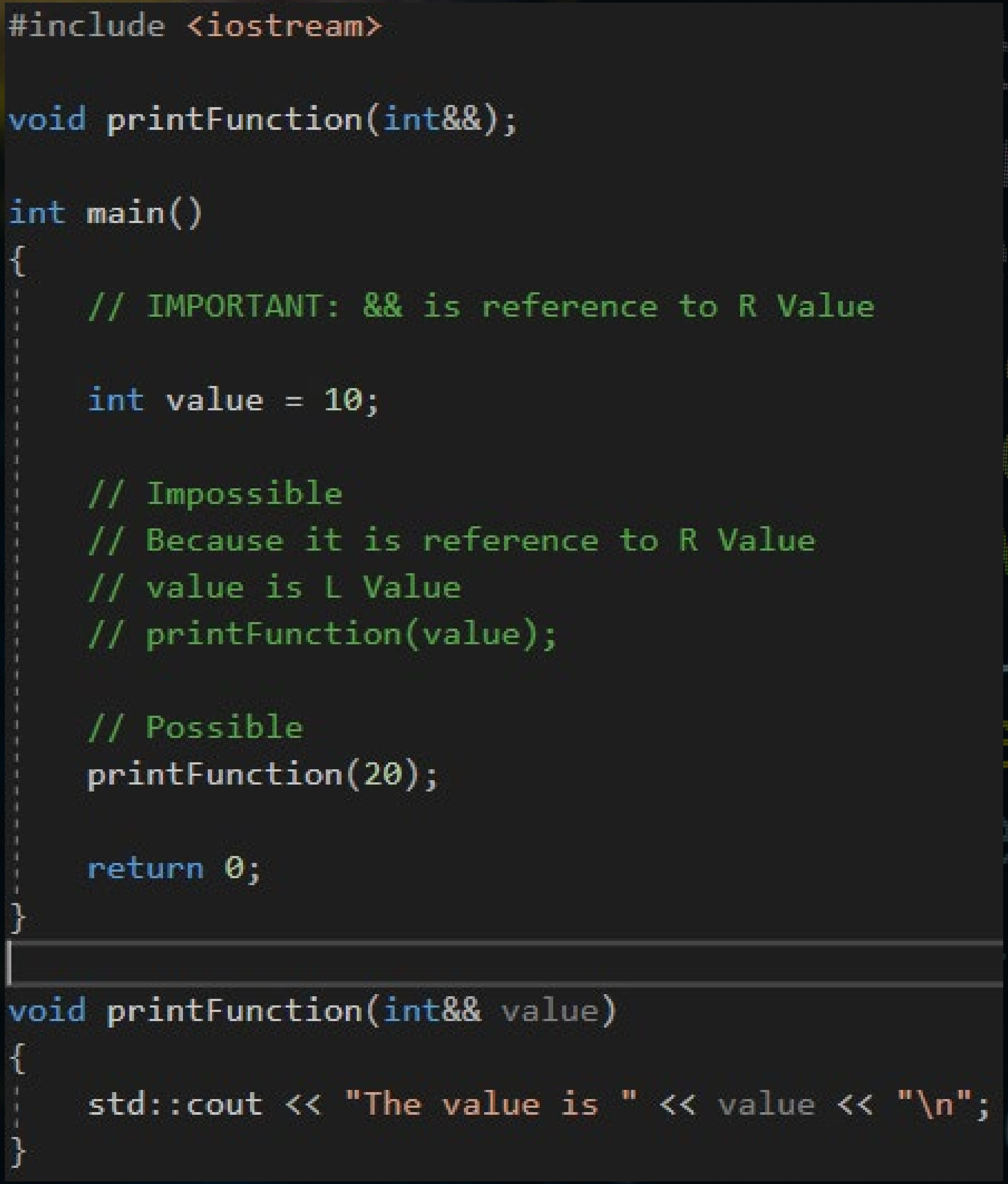
```
#include <iostream>

int main()
{
    int value;

    // The reference is virtual ally
    // of the variable which is pointing to
    int& ref = value;

    return 0;
}
```



```cpp
#include <iostream>

void swap(int&, int&);

int main()
{
    int left = 5, right = 10;

    // You can use reference in function arguments
    // The reference in the argument will work directly
    // on the variable which is pointing to
    // It will exclude the method of shadow copping
    // Because you will not make a copy of the variable
    // you want to work with
    // This method is helpful for memory control and optimization

    // This function will take effect in
    // global scale
    swap(left, right);

    std::cout << left << " " << right << "\n";

    return 0;
}

void swap(int& left, int& right)
{
    int temp = left;
    left = right;
    right = temp;
}
```



```cpp
#include <iostream>

int main()
{
    int value = 10;
    int* ptr = &value;

    // You can make reference to a pointer
    int*& ref = ptr;

    std::cout << "Value: " << value << " Pointer Value: " << *ptr << " Reference to Pointer Value: " << *ref << "\n";

    return 0;
}
```



```cpp
#include <iostream>

void printFunction(int&&);

int main()
{
    // IMPORTANT: && is reference to R Value

    int value = 10;

    // Impossible
    // Because it is reference to R Value
    // value is L Value
    // printFunction(value);

    // Possible
    printFunction(20);

    return 0;
}

void printFunction(int&& value)
{
    std::cout << "The value is " << value << "\n";
}
```

Thank you for watching