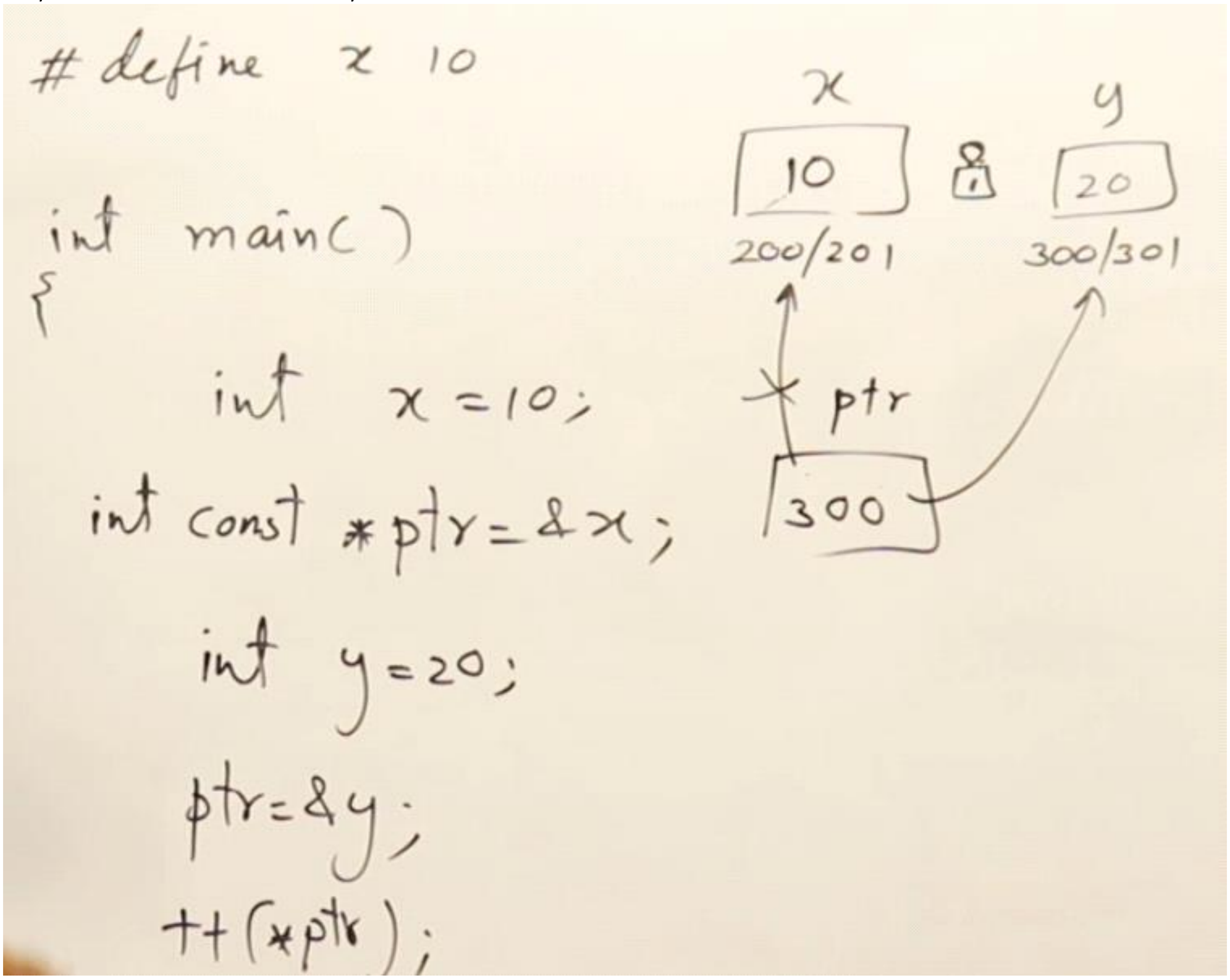
Constants, Preprocessor directives and Namespaces

24 September 2024 19:14

Constants Qualifier

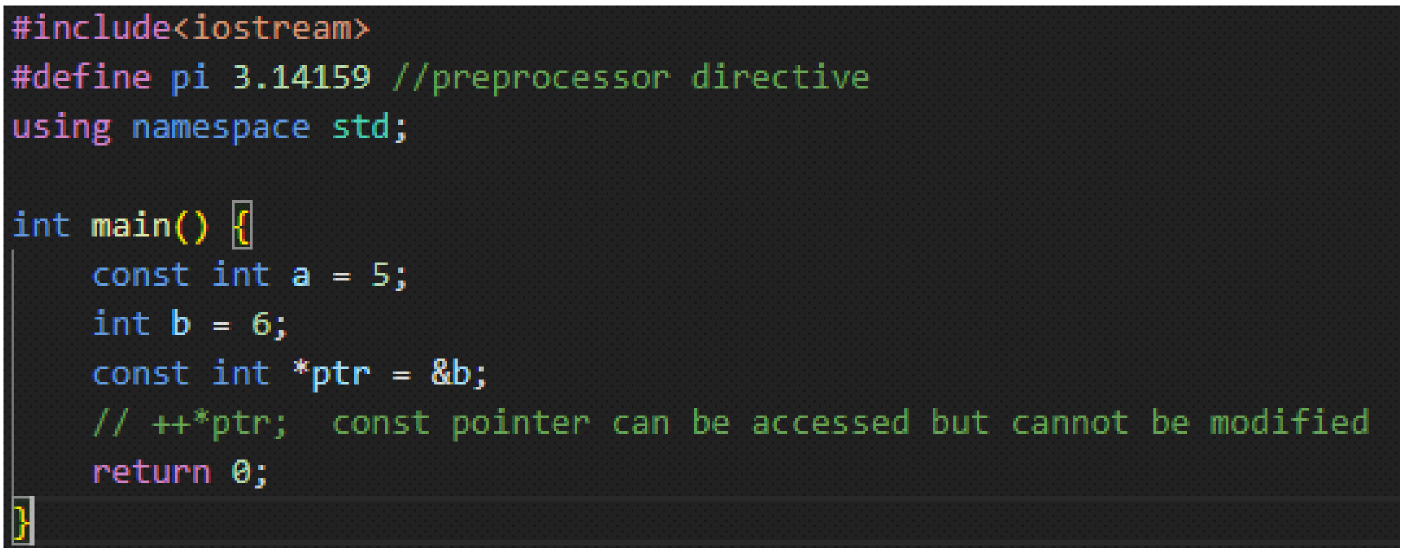
It refer to the use of the const keyword to mark variables, pointers, references, or member functions as immutable.

Const pointer can be read but cannot be modified. The const ptr can point at any data but cannot modify that data.



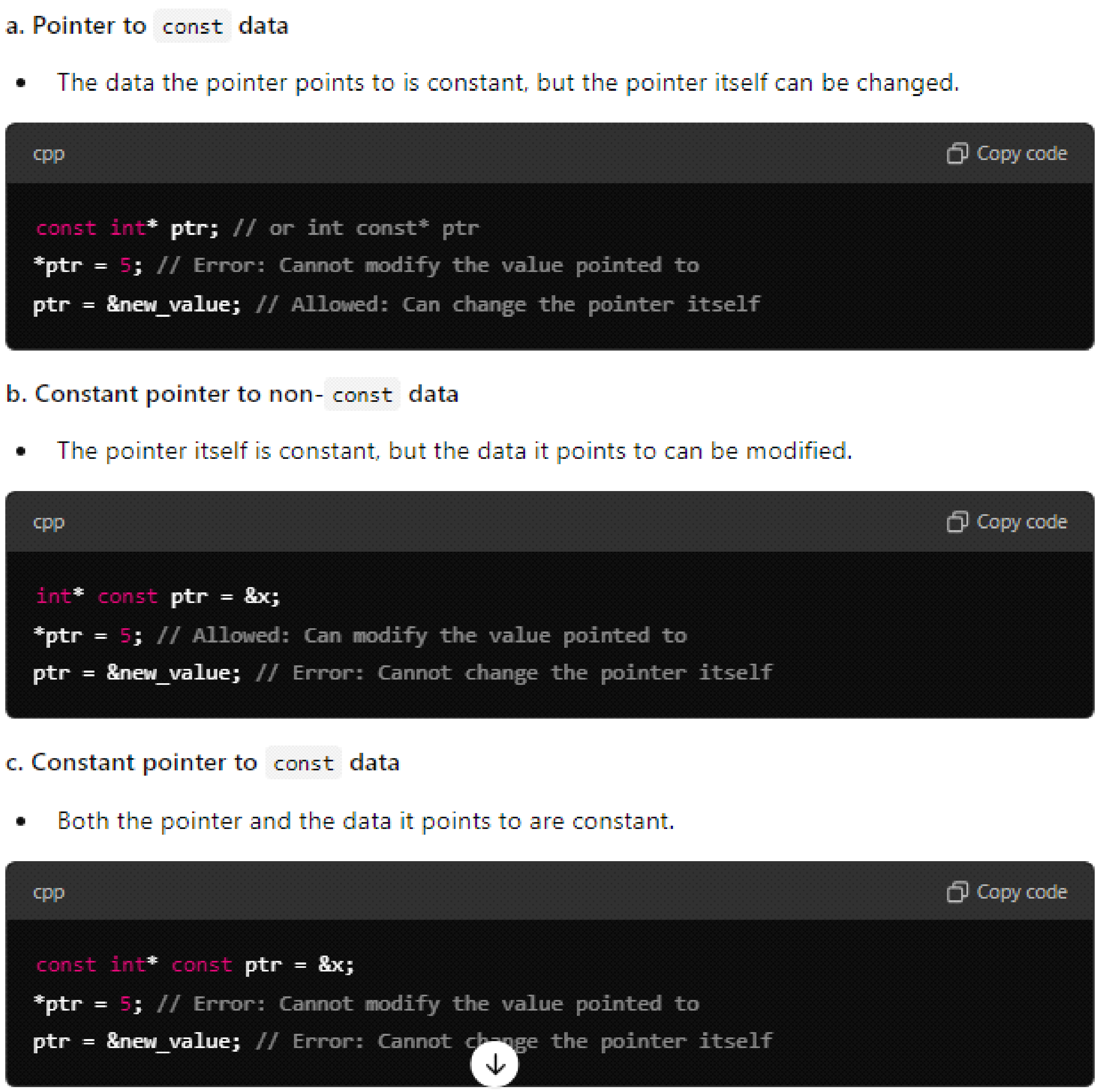
```
#define x 10

int main()
{
    int x=10;
  int const *ptr=&x;
    int y=20;
   ptr=&y;
  ++(*ptr);
```



```cpp
#include<iostream>
#define pi 3.14159 //preprocessor directive
using namespace std;

int main() {
    const int a = 5;
    int b = 6;
    const int *ptr = &b;
    // ++*ptr;  const pointer can be accessed but cannot be modified
    return 0;
}
```



a. Pointer to `const` data

- The data the pointer points to is constant, but the pointer itself can be changed.

```cpp
const int* ptr; // or int const* ptr
*ptr = 5; // Error: Cannot modify the value pointed to
ptr = &new_value; // Allowed: Can change the pointer itself
```

b. Constant pointer to non-`const` data

- The pointer itself is constant, but the data it points to can be modified.

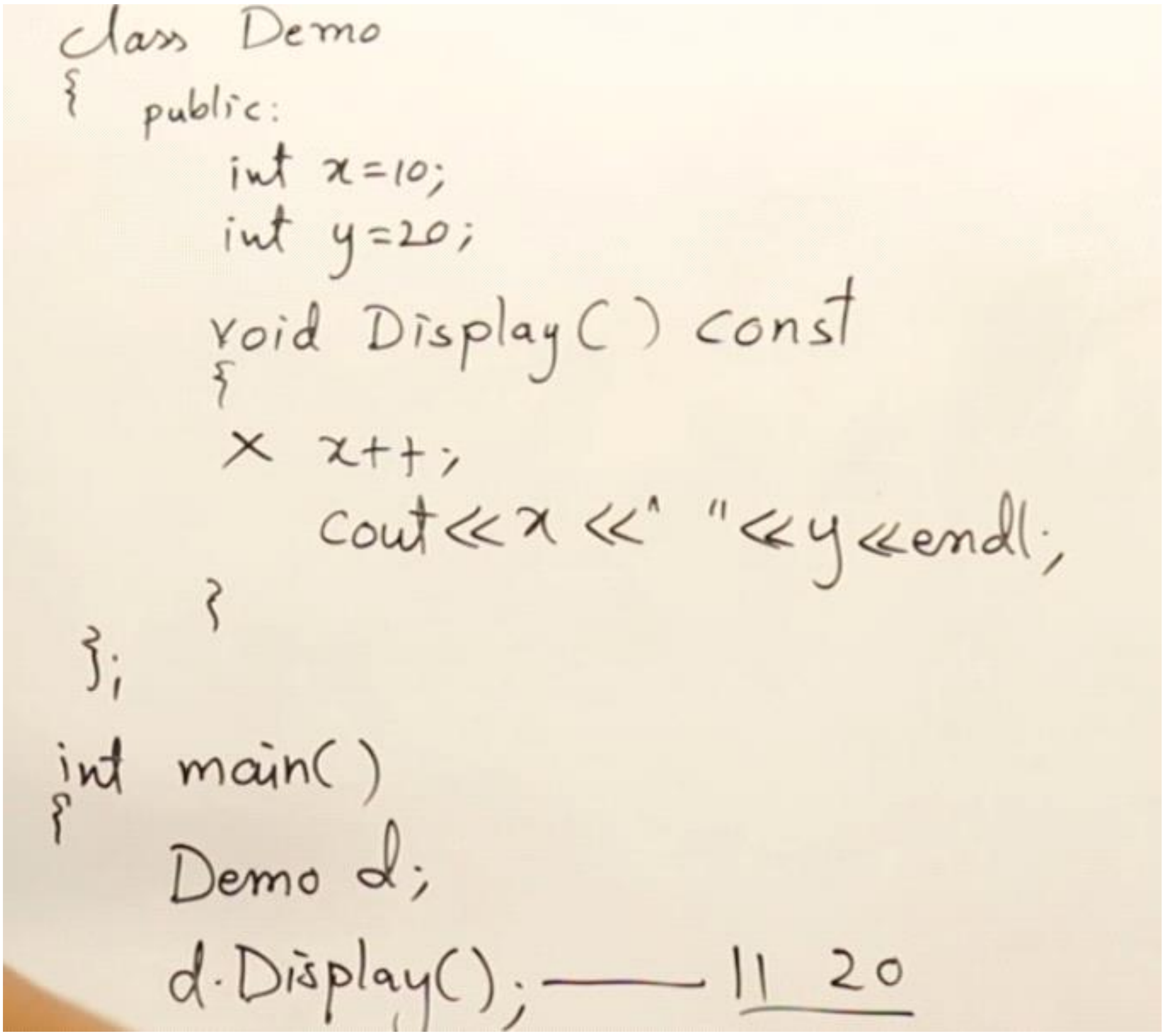
```cpp
int* const ptr = &x;
*ptr = 5; // Allowed: Can modify the value pointed to
ptr = &new_value; // Error: Cannot change the pointer itself
```

c. Constant pointer to `const` data

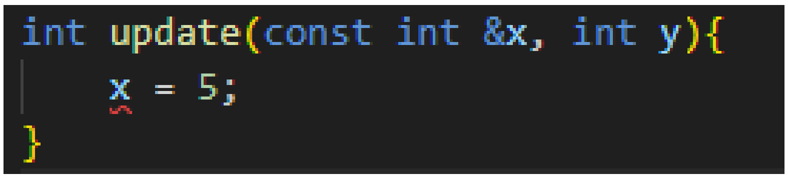
- Both the pointer and the data it points to are constant.

```cpp
const int* const ptr = &x;
*ptr = 5; // Error: Cannot modify the value pointed to
ptr = &new_value; // Error: Cannot change the pointer itself
```

If we know that we are not going to update any data we can use const in a function. And if we by mistake update any data the compiler will give us an error.



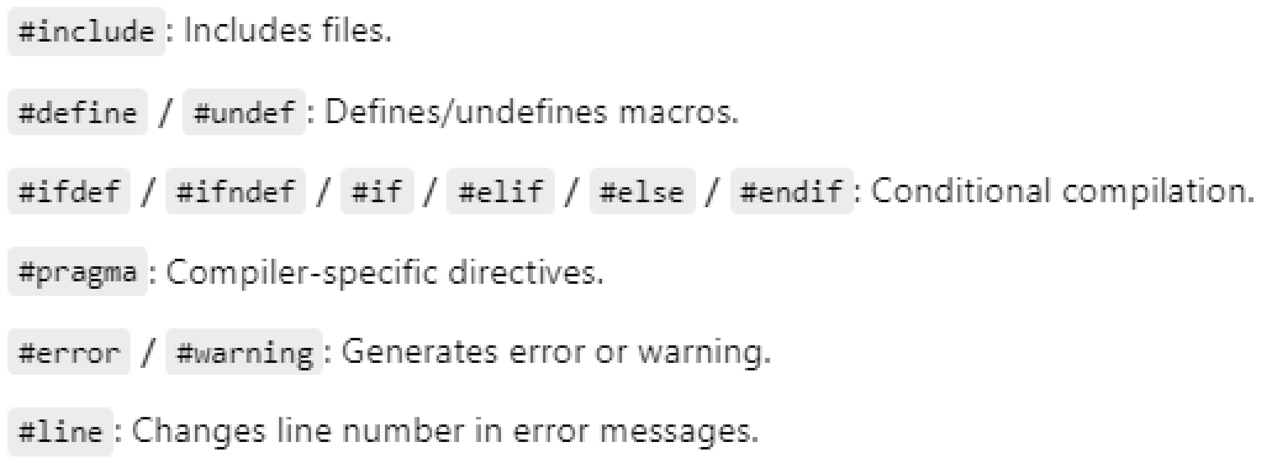
```
class Demo
{  public:
       int x=10;
       int y=20;
       void Display() const
       {
       X  x++;
           cout<<x<<" "<<y<<endl;
       }
};
int main()
{
     Demo d;
     d.Display(); —— 11 20
```



```
int update(const int &x, int y){
    x = 5;
}
```

Preprocessor directives

They are the commands that we give to our compiler.



`#include` : Includes files.

`#define` / `#undef` : Defines/undefines macros.

`#ifdef` / `#ifndef` / `#if` / `#elif` / `#else` / `#endif` : Conditional compilation.

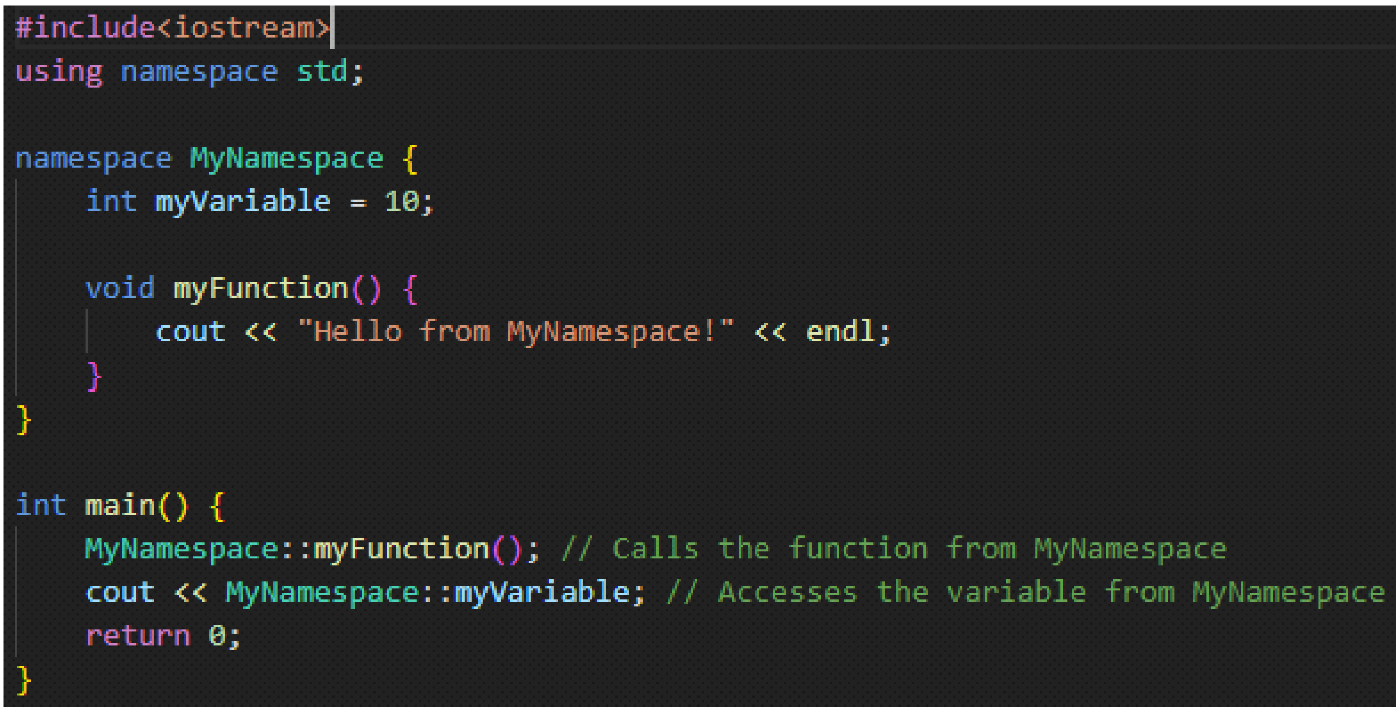
`#pragma` : Compiler-specific directives.

`#error` / `#warning` : Generates error or warning.

`#line` : Changes line number in error messages.

Namespace

namespace is a feature that allows us to organize code into logical groups and avoid name collisions, especially when your code base includes multiple libraries. Namespaces provide a way to group related classes, functions, variables, and other identifiers.



```cpp
#include<iostream>
using namespace std;

namespace MyNamespace {
    int myVariable = 10;

    void myFunction() {
        cout << "Hello from MyNamespace!" << endl;
    }
}

int main() {
    MyNamespace::myFunction(); // Calls the function from MyNamespace
    cout << MyNamespace::myVariable; // Accesses the variable from MyNamespace
    return 0;
}
```