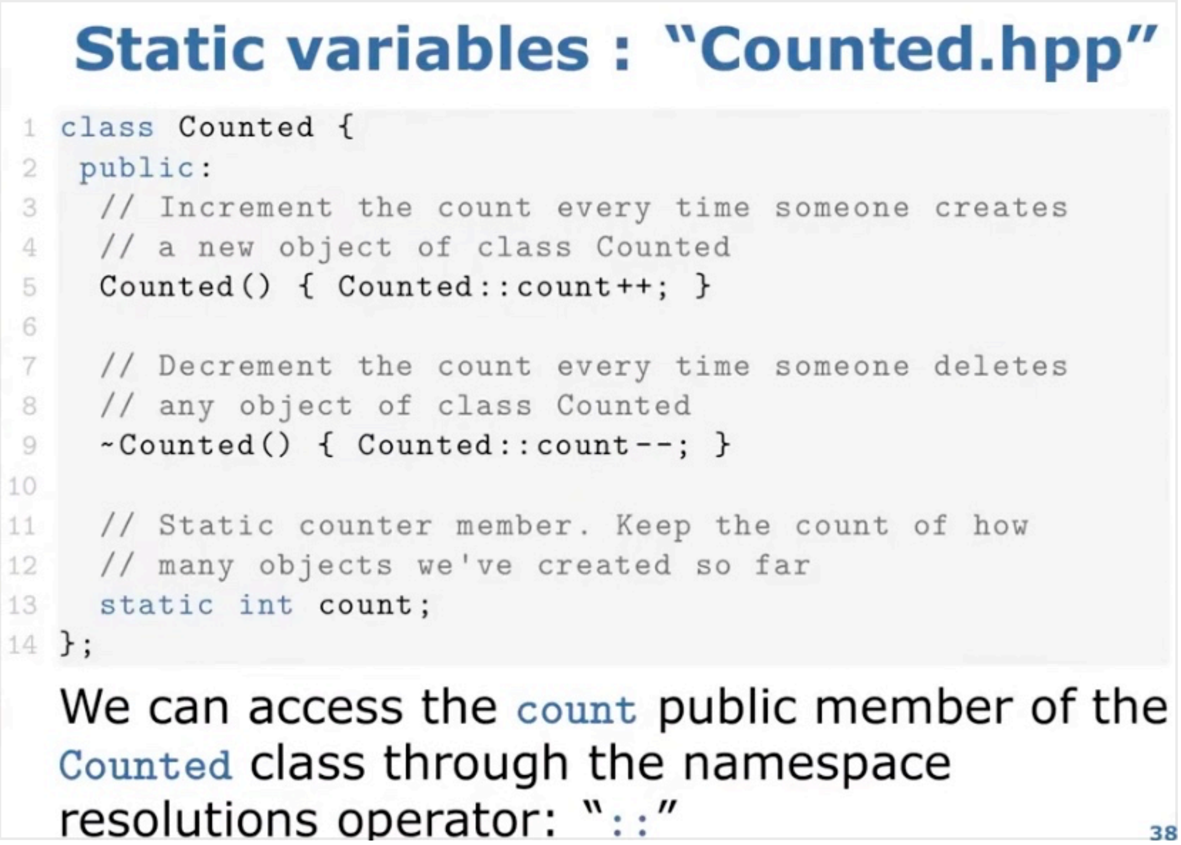
Static Variables and Methods

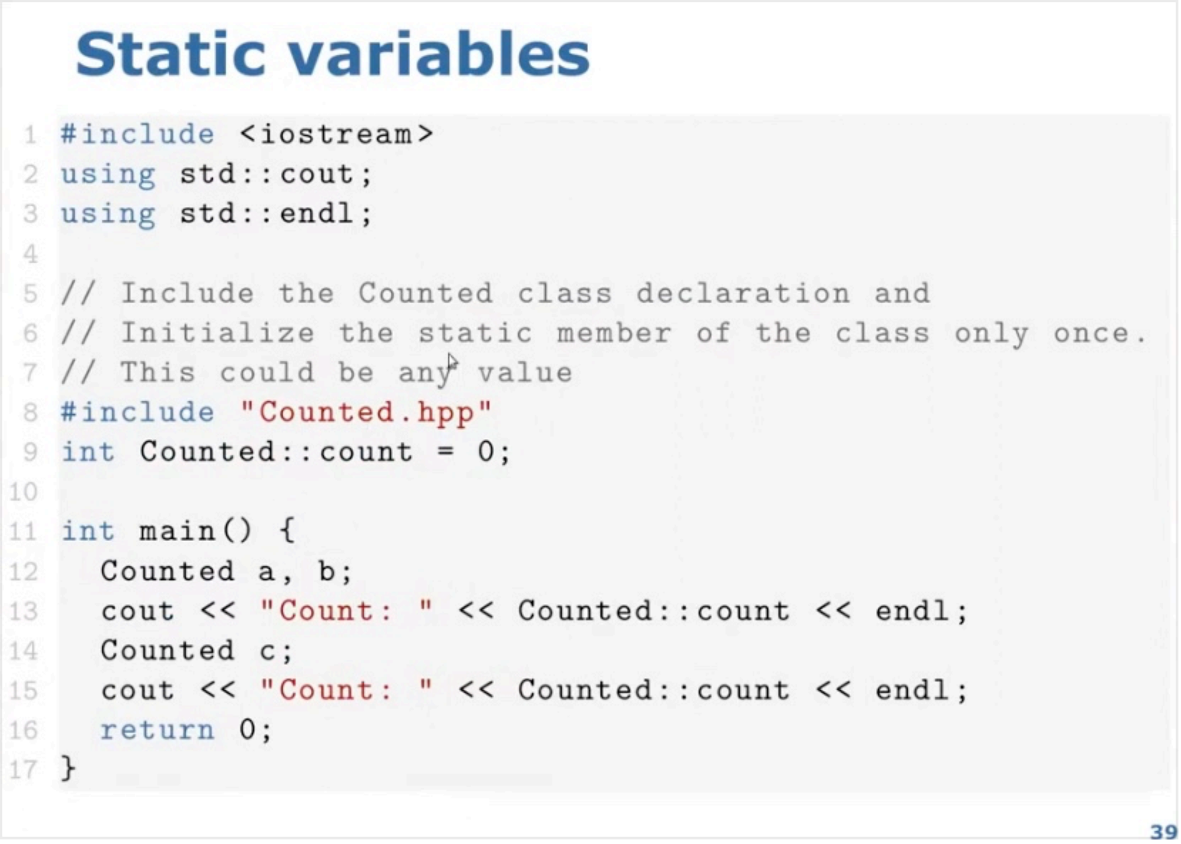
- Static member variables of a class
  - Exist exactly once per class, not per object.
  - The value is equal across all instances.
  - Must be defined in *.cpp fils (before C++17).
- Static member functions of a class
  - Do not need to access through an object of the class
  - Can access private members but need an object.
  - Syntax for calling: ClassName::MethodName(<params>)



Static variables : “Counted.hpp”

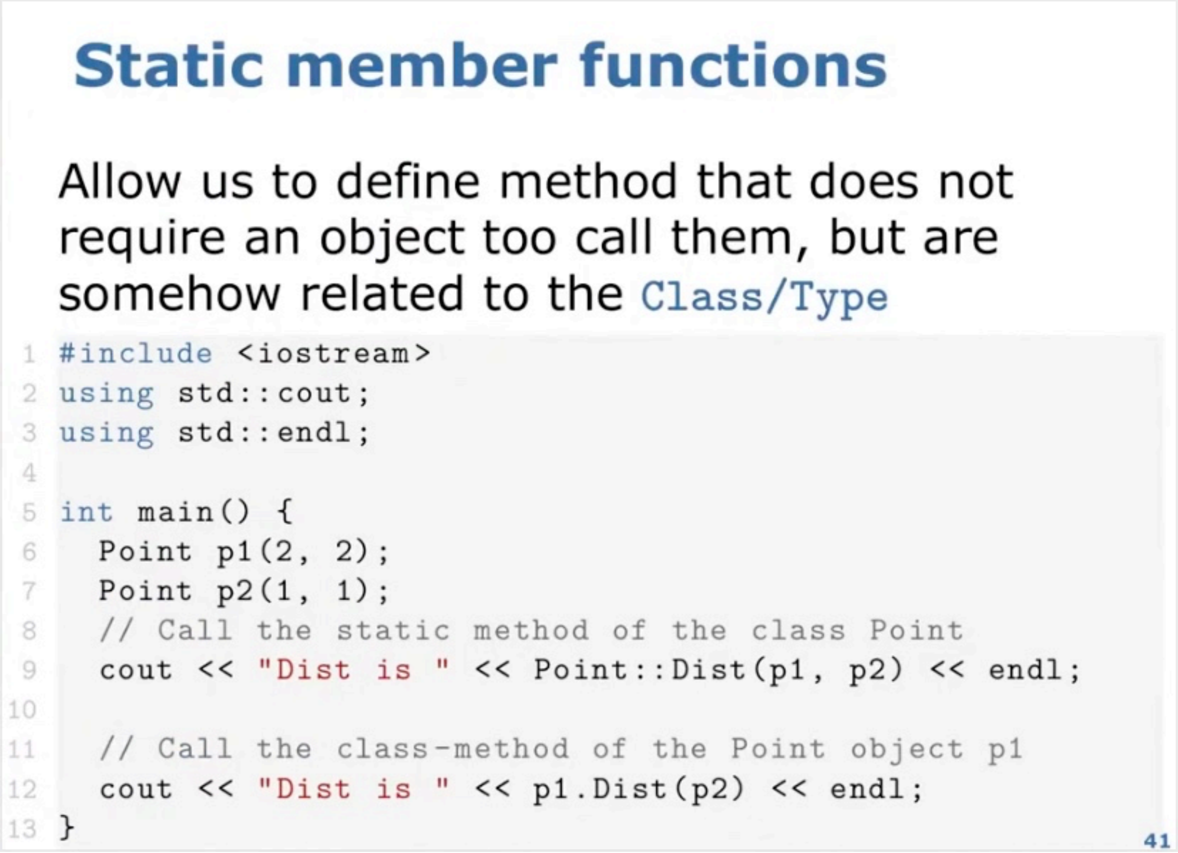
```cpp
class Counted {
 public:
  // Increment the count every time someone creates
  // a new object of class Counted
  Counted() { Counted::count++; }

  // Decrement the count every time someone deletes
  // any object of class Counted
  ~Counted() { Counted::count--; }

  // Static counter member. Keep the count of how
  // many objects we've created so far
  static int count;
};
```

We can access the count public member of the Counted class through the namespace resolutions operator: “::”



Static variables

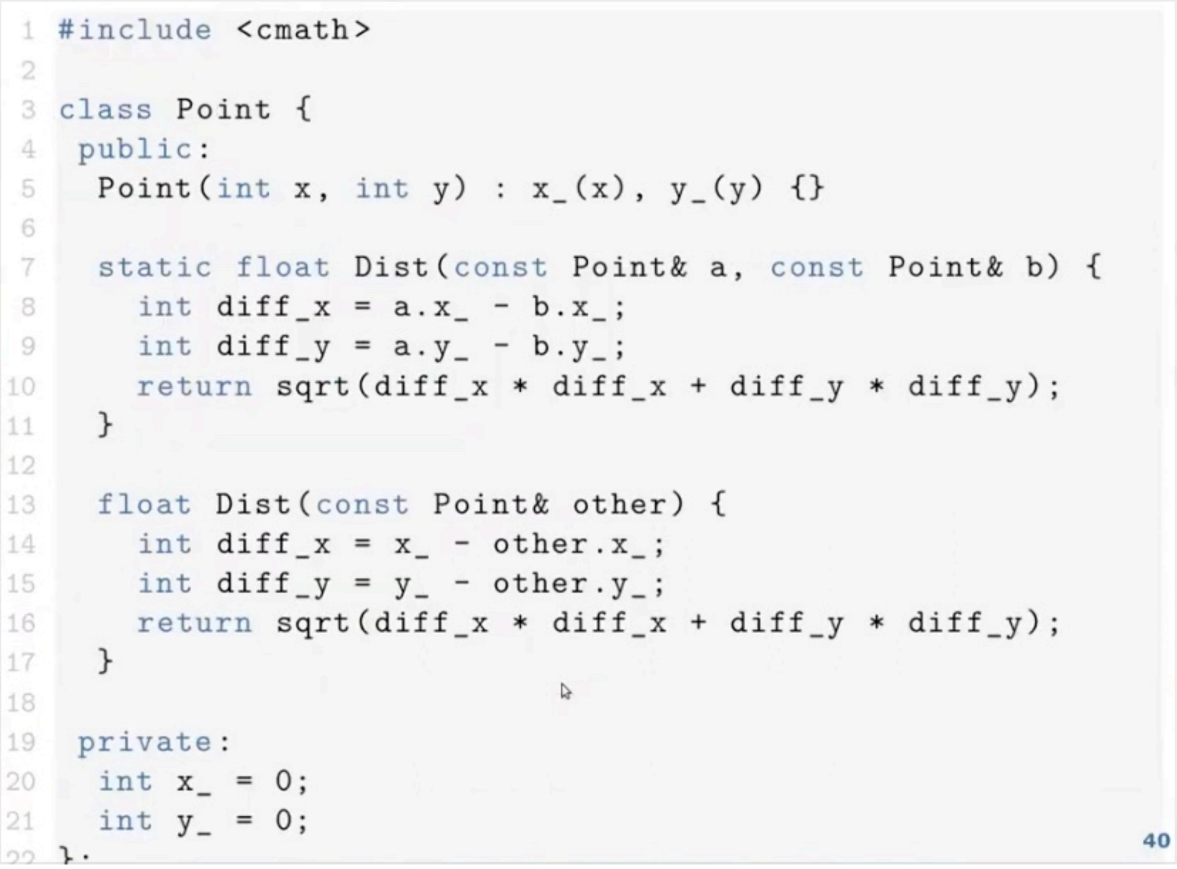
```cpp
#include <iostream>
using std::cout;
using std::endl;

// Include the Counted class declaration and
// Initialize the static member of the class only once.
// This could be any value
#include "Counted.hpp"
int Counted::count = 0;

int main() {
  Counted a, b;
  cout << "Count: " << Counted::count << endl;
  Counted c;
  cout << "Count: " << Counted::count << endl;
  return 0;
}
```



Static member functions

Allow us to define method that does not require an object too call them, but are somehow related to the `Class/Type`

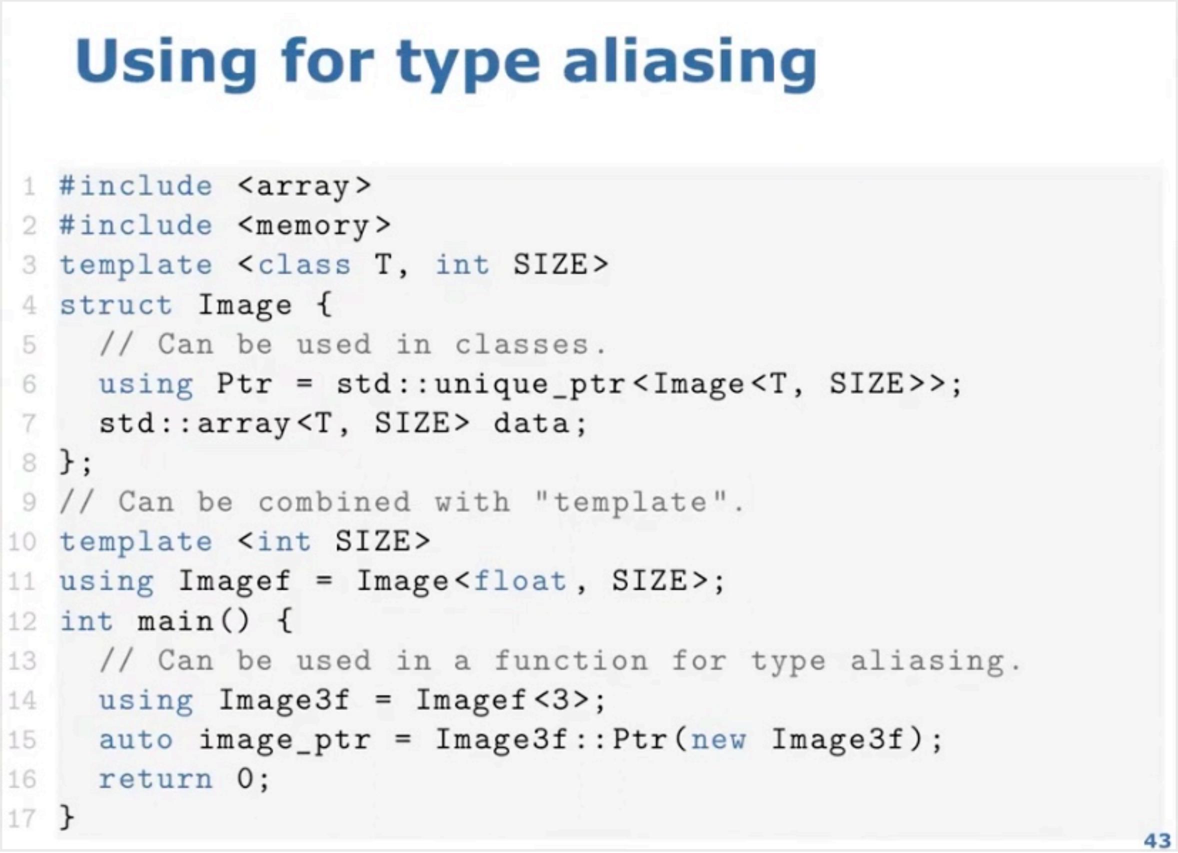
```cpp
#include <iostream>
using std::cout;
using std::endl;

int main() {
  Point p1(2, 2);
  Point p2(1, 1);
  // Call the static method of the class Point
  cout << "Dist is " << Point::Dist(p1, p2) << endl;

  // Call the class-method of the Point object p1
  cout << "Dist is " << p1.Dist(p2) << endl;
}
```



```cpp
#include <cmath>

class Point {
 public:
  Point(int x, int y) : x_(x), y_(y) {}

  static float Dist(const Point& a, const Point& b) {
    int diff_x = a.x_ - b.x_;
    int diff_y = a.y_ - b.y_;
    return sqrt(diff_x * diff_x + diff_y * diff_y);
  }

  float Dist(const Point& other) {
    int diff_x = x_ - other.x_;
    int diff_y = y_ - other.y_;
    return sqrt(diff_x * diff_x + diff_y * diff_y);
  }

 private:
  int x_ = 0;
  int y_ = 0;
};
```

Using for Type Aliasing

- Use word “using” to declare new types from existing and to create type aliases.
- Basic syntax:
  using NewType = OldType
- When used outside of functions declares a new type alias
- When used in function, creates an alias of a type available in the current scope.



Using for type aliasing

```cpp
#include <array>
#include <memory>
template <class T, int SIZE>
struct Image {
  // Can be used in classes.
  using Ptr = std::unique_ptr<Image<T, SIZE>>;
  std::array<T, SIZE> data;
};
// Can be combined with "template".
template <int SIZE>
using Imagef = Image<float, SIZE>;
int main() {
  // Can be used in a function for type aliasing.
  using Image3f = Imagef<3>;
  auto image_ptr = Image3f::Ptr(new Image3f);
  return 0;
}
```