```cpp
//#include "stdafx.h"

#include <cstdint>
#include <random>
#include <functional>
#include <algorithm>
#include <opencv2/opencv.hpp>

using namespace cv;

// Lambda function test

void lambda_tests()
{
    int N = 1000000;
    double mu = 0, sigma = 10;
    double min_z = 0, max_z = 8; // Inlier range:[min_z, max_z]

    // Generate N random numbers in N[mu,sigma]

    std::vector<Vec3d> pts(N);

    std::default_random_engine rgen;
    std::normal_distribution<double> P(mu, sigma);
    auto rng = bind(P, rgen);

    // A predefined lambda function, using the local object 'rng'.
    auto rng_vec3d = [&]()
    {
        return Vec3d(rng(), rng(), rng());
    };
    // Using the predefined function
    std::generate(begin(pts), end(pts), rng_vec3d);

    printf("Generated %d points\n", pts.size());

    // Remove outliers - vectors with z-values outside inlier range

    // Using a lambda function defined inline
    auto p_end = std::remove_if(begin(pts), end(pts), [&](const Vec3d& p) {
        return (p(2) < min_z || p(2) > max_z);
    });
    pts.erase(p_end, end(pts));

    printf("Removed outliers. %d points left. (%.0f%%)\n", pts.size(), 100.0f * pts.size() /
    N);
}


// Rvalue reference test

/// Object with move semantics
struct MovableObject
{
    std::string name;
    size_t N;
    char *data;
```

```
    MovableObject(std::string name)
        : name(name)
    {
        printf("Creating '%s'\n", name.c_str());
        N = 1000000;
        data = new char[N];
    }

    ~MovableObject()
    {
        printf("Destroying %s'%s'\n", data == nullptr ? "empty " : "", name.c_str());
        delete data;
    }

    MovableObject& operator=(const MovableObject& rhs)
    {
        printf("Copying content of '%s' to '%s'\n", rhs.name.c_str(), name.c_str());

        delete data;
        N = rhs.N;
        data = new char[N];
        std::copy(rhs.data, rhs.data + N, data);
        return *this;
    }

    MovableObject& operator=(MovableObject&& rhs)
    {
        printf("Moving content of %s to %s\n", rhs.name.c_str(), name.c_str());

        data = rhs.data;
        N = rhs.N;
        rhs.data = 0;
        return *this;
    }
};

MovableObject operator+(const MovableObject& x, const MovableObject& y)
{
    MovableObject z(x.name + "+" + y.name);

    printf("%s = %s + %s\n", z.name.c_str(), x.name.c_str(), y.name.c_str());

    for (size_t i = 0; i < x.N; i++) {
        z.data[i] = x.data[i] + y.data[i];
    }
    return z;
}

// Rvalue reference test

/// Object without move semantics
struct CopyableObject
{
    std::string name;
    size_t N;
    char *data;
```

```
    CopyableObject(std::string name)
        : name(name)
    {
        printf("Creating '%s'\n", name.c_str());
        N = 1000000;
        data = new char[N];
    }

    ~CopyableObject()
    {
        printf("Destroying %s'%s'\n", data == nullptr ? "empty " : "", name.c_str());
        delete data;
    }

    CopyableObject& operator=(const CopyableObject& rhs)
    {
        printf("Copying content of '%s' to '%s'\n", rhs.name.c_str(), name.c_str());

        delete data;
        N = rhs.N;
        data = new char[N];
        std::copy(rhs.data, rhs.data + N, data);
        return *this;
    }

    // No operator=() taking an rvalue reference is defined in this class
};

CopyableObject operator+(const CopyableObject& x, const CopyableObject& y)
{
    CopyableObject z(x.name + "+" + y.name);

    printf("%s = %s + %s\n", z.name.c_str(), x.name.c_str(), y.name.c_str());

    for (size_t i = 0; i < x.N; i++) {
        z.data[i] = x.data[i] + y.data[i];
    }
    return z;
}

void move_semantics_test()
{
    {
        printf("Without move semantics\n\n");

        CopyableObject a("a");
        CopyableObject b("b");
        CopyableObject c("c");

        printf("...\n");

        c = a + b;

        printf("...\n");
    }
```

```
    {
        printf("\nWith move semantics\n\n");

        MovableObject a("a");
        MovableObject b("b");
        MovableObject c("c");

        printf("...\n");

        c = a + b;

        printf("...\n");
    }
}

// Output from move_semantics_test()
//
// Without move semantics
//
// Creating 'a'
// Creating 'b'
// Creating 'c'
// ...
// Creating 'a+b'
// a + b = a + b
// Copying content of 'a+b' to 'c'  <= Copying object here
// Destroying 'a+b'
// ...
// Destroying 'c'
// Destroying 'b'
// Destroying 'a'
//
// With move semantics
//
// Creating 'a'
// Creating 'b'
// Creating 'c'
// ...
// Creating 'a+b'
// a + b = a + b
// Moving content of a + b to c   <= Moving object here
// Destroying empty 'a+b'
// ...
// Destroying 'c'
// Destroying 'b'
// Destroying 'a'

int main(int argc, char* argv[])
{
    move_semantics_test();
    //lambda_tests();
    exit(0);
}
```