2.

Sai: int \*ptr3 = &arr

Gán &arr cho int \*ptr3 có thể gây ra undefined behavior khi sử dụng ptr3.

&arr là địa chỉ của toàn bộ mảng, không phải của phần tử đầu tiên. Vì vậy, &arr là một con trỏ đến kiểu int[3] (một mảng có 3 phần tử kiểu int), không phải là int\*.

=> Kiểu của ptr3 không khớp với giá trị được gán.

3. Capture clause của một lambda expression được sử dụng để xác định cách các biến bên ngoài (trong phạm vi chứa lambda) được truyền vào và sử dụng bên trong lambda.

* Theo giá trị (=): lambda chỉ sử dụng bản sao, không ảnh hưởng đến biến gốc.
* Theo tham chiếu (&): Cho phép lambda truy cập và thao tác trực tiếp trên biến gốc bên ngoài lambda.

4.

Cách 1: Mảng arr được cấp phát động với kích thước 3 \* 4 => là một mảng 1 chiều chứa toàn bộ dữ liệu. Các phần tử trong mảng được truy cập bằng công thức: arr[i \* 4 + j].

Nhược điểm: Truy cập phần tử yêu cầu sử dụng công thức i \* 4 + j, khiến mã phức tạp hơn và dễ xảy ra lỗi (sai chỉ số hoặc tính toán sai offset). Kích thước cột phải cố định, nếu cần thay đổi số cột, cần sửa đổi toàn bộ logic tính chỉ số.

Cách 2:

Cấp phát một mảng con trỏ arr có kích thước 3. Mỗi phần tử của arr trỏ đến một mảng con có kích thước 4, tạo thành mảng 2 chiều. Truy cập phần tử tại hàng i và cột j thông qua arr[i][j].

Nhược điểm: Mỗi hàng yêu cầu một con trỏ, sẽ tốn thêm bộ nhớ cho các con trỏ này. Cần lặp qua từng hàng và gọi delete[] để giải phóng bộ nhớ của từng mảng con, sau đó mới gọi delete[] để giải phóng mảng con trỏ => phức tạp.

5. diff=10

(pe - pt) đo khoảng cách giữa hai con trỏ pe và pt theo số ô nhớ kiểu int.

pe trỏ đến a + 10 và pt trỏ đến a.

6. Gán con trỏ là nullptr sau khi giải phóng

- Tránh lỗi dangling pointer khi một con trỏ đã được giải phóng nhưng không được gán lại giá trị, nó vẫn giữ địa chỉ của vùng bộ nhớ cũ. Nếu sử dụng hoặc truy cập, sẽ gây undefined behavior...

- Giúp kiểm tra tính hợp lệ của con trỏ trước khi sử dụng.

- Tránh lỗi giải phóng vùng nhớ nhiều lần.

7. printf("item %d pointed to by gp: %s<br>", i, \*(char \*)gp);

Chuyển đổi gp sang char\* để in ra chuỗi, nhưng gp chứa địa chỉ của const char\*, không phải char\*. => Có thể gây nên undefined behavior.

8.

Inline expansion thường được sử dụng để tối ưu hóa hiệu suất của chương trình bằng cách thay thế các hàm hoặc phương thức bằng mã của chúng ngay tại chỗ mà chúng được gọi. Thay vì gọi hàm, trình biên dịch sẽ chèn mã của hàm đó vào vị trí gọi.

9.

Theo quy tắc trong C++ là nếu chỉ định tham số mặc định cho một tham số nào đó, tất cả các tham số ở bên phải của nó cũng phải có tham số mặc định.

void foo(int a, int b = 5, int c). Ở đây, tham số b có một giá trị mặc định là 5, nhưng tham số c không có giá trị mặc định => Không hợp lệ trong C++, vì c nằm bên phải b trong danh sách tham số.

10.

Hàm lồng nhau không được phép vì nếu định nghĩa hàm bên trong một hàm khác, sẽ rất khó để quản lý các hàm này => Dễ gây nhầm lẫn về cách gọi hàm, và làm cho mã khó đọc hơn.

Biểu thức lambda được phép do lambda cho phép tạo ra hàm tạm thời có thể sử dụng chúng ngay lập tức để thực hiện một số tác vụ nhỏ. Lambda có thể truy cập vào các biến từ hàm bên ngoài mà nó được định nghĩa => Giúp mã trở nên linh hoạt và dễ hiểu hơn.

#include <iostream>

#include <numeric> // for std::gcd

class Fraction {

private:

    int numerator;   // Tử số

    int denominator; // Mẫu số

    // Hàm rút gọn phân số

    void reduce() {

        int gcd = std::gcd(numerator, denominator);

        numerator /= gcd;

        denominator /= gcd;

        // Đảm bảo mẫu số luôn dương

        if (denominator < 0) {

            numerator = -numerator;

            denominator = -denominator;

        }

    }

public:

    // Constructor

    Fraction(int num = 0, int denom = 1) : numerator(num), denominator(denom) {

        if (denominator == 0) {

            throw std::invalid\_argument("Denominator cannot be zero");

        }

        reduce();

    }

    // Hàm in phân số

    void print() const {

        std::cout << numerator << "/" << denominator;

    }

    // Phép cộng

    Fraction operator+(const Fraction& other) const {

        return Fraction(numerator \* other.denominator + other.numerator \* denominator,

                        denominator \* other.denominator);

    }

    // Phép trừ

    Fraction operator-(const Fraction& other) const {

        return Fraction(numerator \* other.denominator - other.numerator \* denominator,

                        denominator \* other.denominator);

    }

    // Phép nhân

    Fraction operator\*(const Fraction& other) const {

        return Fraction(numerator \* other.numerator, denominator \* other.denominator);

    }

    // Phép chia

    Fraction operator/(const Fraction& other) const {

        if (other.numerator == 0) {

            throw std::invalid\_argument("Cannot divide by zero");

        }

        return Fraction(numerator \* other.denominator, denominator \* other.numerator);

    }

    // Toán tử tiền tố tăng (prefix)

    Fraction& operator++() {

        numerator += denominator; // Tăng tử số lên 1 đơn vị mẫu số

        reduce();

        return \*this;

    }

    // Toán tử hậu tố tăng (postfix)

    Fraction operator++(int) {

        Fraction temp = \*this; // Lưu trạng thái hiện tại

        ++(\*this);             // Gọi toán tử tiền tố

        return temp;          // Trả lại trạng thái cũ

    }

    // Toán tử tiền tố giảm (prefix)

    Fraction& operator--() {

        numerator -= denominator; // Giảm tử số xuống 1 đơn vị mẫu số

        reduce();

        return \*this;

    }

    // Toán tử hậu tố giảm (postfix)

    Fraction operator--(int) {

        Fraction temp = \*this; // Lưu trạng thái hiện tại

        --(\*this);             // Gọi toán tử tiền tố

        return temp;          // Trả lại trạng thái cũ

    }

};

int main() {

    try {

        Fraction frac(1, 2); // 1/2

        std::cout << "Initial Fraction: ";

        frac.print();

        std::cout << std::endl;

        // Sử dụng toán tử tăng

        ++frac;

        std::cout << "After prefix increment: ";

        frac.print();

        std::cout << std::endl;

        frac++;

        std::cout << "After postfix increment: ";

        frac.print();

        std::cout << std::endl;

        // Sử dụng toán tử giảm

        --frac;

        std::cout << "After prefix decrement: ";

        frac.print();

        std::cout << std::endl;

        frac--;

        std::cout << "After postfix decrement: ";

        frac.print();

        std::cout << std::endl;

    } catch (const std::exception& e) {

        std::cerr << "Error: " << e.what() << std::endl;

    }

    return 0;

}

#include <iostream>

#include <vector>

#include <functional>

#include <algorithm>

using namespace std;

enum FlipDir {

    kHorizontal,

    kVertical,

    kNothing,

};

struct Image {

    int width, height; // max width = max height = 10

    unsigned char \*data; // pointer to the first element of the array (0 - 255)

};

using FilterFunc = function<void(Image&)>;

struct Pipeline {

    vector<FilterFunc> filters; // a list of filter functions to apply in sequence

};

ostream &operator<<(ostream &os, const Image &img) {

    os << "Image(" << img.width << ", " << img.height << ")\n";

    for (int i = 0; i < img.height; ++i) {

        for (int j = 0; j < img.width; ++j) {

            os << (int)img.data[i \* img.width + j] << " "; // Print pixel values

        }

        os << "\n";

    }

    return os;

}

void brightnessChange(Image &img, int delta) {

    for (int i = 0; i < img.width \* img.height; ++i) {

        img.data[i] = std::clamp((int)img.data[i] + delta, 0, 255);

    }

}

void rotate(Image &img, int angle) {

    if (angle == 90 || angle == 270) {

        unsigned char\* rotatedData = new unsigned char[img.width \* img.height];

        for (int i = 0; i < img.height; ++i) {

            for (int j = 0; j < img.width; ++j) {

                if (angle == 90) {

                    rotatedData[j \* img.height + (img.height - 1 - i)] = img.data[i \* img.width + j];

                } else { // angle == 270

                    rotatedData[(img.width - 1 - j) \* img.height + i] = img.data[i \* img.width + j];

                }

            }

        }

        // Update image dimensions and data

        swap(img.width, img.height);

        delete[] img.data;

        img.data = rotatedData;

    } else if (angle == 180) {

        for (int i = 0; i < img.width \* img.height; ++i) {

            img.data[i] = img.data[img.width \* img.height - 1 - i];

        }

    }

}

void flip(Image &img, FlipDir dir) {

    if (dir == kHorizontal) {

        for (int i = 0; i < img.height; ++i) {

            for (int j = 0; j < img.width / 2; ++j) {

                swap(img.data[i \* img.width + j], img.data[i \* img.width + (img.width - 1 - j)]);

            }

        }

    } else if (dir == kVertical) {

        for (int i = 0; i < img.width; ++i) {

            for (int j = 0; j < img.height / 2; ++j) {

                swap(img.data[j \* img.width + i], img.data[(img.height - 1 - j) \* img.width + i]);

            }

        }

    }

}

void applyFilter(Image &img, int delta, int angle, FlipDir flipDir) {

    Pipeline pipeline;

    if (delta != 0) {

        pipeline.filters.push\_back([delta](Image &img) { brightnessChange(img, delta); });

    }

    if (angle == 90 || angle == 180 || angle == 270) {

        pipeline.filters.push\_back([angle](Image &img) { rotate(img, angle); });

    }

    if (flipDir != kNothing) {

        pipeline.filters.push\_back([flipDir](Image &img) { flip(img, flipDir); });

    }

    for (auto filter : pipeline.filters) {

        filter(img);

    }

}

int main() {

    Image image;

    image.width = 3;

    image.height = 3;

    image.data = new unsigned char[image.width \* image.height];

    // Random initialization of pixel values

    for (int i = 0; i < image.width \* image.height; ++i) {

        image.data[i] = rand() % 256; // random pixel value between 0 and 255

    }

    cout << "Original Image:\n" << image;

    applyFilter(image, 0, 0, kNothing); // This means no filter is applied

    cout << "After no filters:\n" << image;

    applyFilter(image, 10, 0, kHorizontal); // Increase brightness by 10 and flip horizontally

    cout << "After brightness +10 and horizontal flip:\n" << image;

    applyFilter(image, -5, 90, kVertical); // Decrease brightness by 5 and rotate 90 degrees

    cout << "After brightness -5 and 90 degree rotation:\n" << image;

    delete[] image.data;

    return 0;

}

#include <iostream>

#include <vector>

#include <functional>

#include <algorithm>

#include <iomanip>

using namespace std;

enum FlipDir {

    kHorizontal,

    kVertical,

    kNothing,

};

struct Image {

    int width, height; // max width = max height = 10

    unsigned char \*data; // pointer to the first element of the array (0 - 255)

};

using FilterFunc = function<void(Image&)>;

struct Pipeline {

    vector<FilterFunc> filters; // a list of filter functions to apply in sequence

};

ostream &operator<<(ostream &os, const Image &img) {

    os << "Image(" << img.width << ", " << img.height << ")\n";

    for (int i = 0; i < img.height; ++i) {

        for (int j = 0; j < img.width; ++j) {

            os << static\_cast<int>(img.data[i \* img.width + j]) << " ";

        }

        os << "\n";

    }

    return os;

}

void brightnessChange(Image &img, int delta) {

    for (int i = 0; i < img.height; ++i) {

        for (int j = 0; j < img.width; ++j) {

            int index = i \* img.width + j;

            int newValue = static\_cast<int>(img.data[index]) + delta;

            img.data[index] = static\_cast<unsigned char>(clamp(newValue, 0, 255));

        }

    }

}

void rotate(Image &img, int angle) {

    if (angle == 0) return;

    Image tempImage;

    if (angle == 90) {

        tempImage.width = img.height;

        tempImage.height = img.width;

        tempImage.data = new unsigned char[tempImage.width \* tempImage.height];

        for (int i = 0; i < img.height; ++i) {

            for (int j = 0; j < img.width; ++j) {

                tempImage.data[j \* tempImage.width + (tempImage.width - 1 - i)] = img.data[i \* img.width + j];

            }

        }

    } else if (angle == 180) {

        tempImage.width = img.width;

        tempImage.height = img.height;

        tempImage.data = new unsigned char[tempImage.width \* tempImage.height];

        for (int i = 0; i < img.height; ++i) {

            for (int j = 0; j < img.width; ++j) {

                tempImage.data[(img.height - 1 - i) \* img.width + (img.width - 1 - j)] = img.data[i \* img.width + j];

            }

        }

    } else if (angle == 270) {

        tempImage.width = img.height;

        tempImage.height = img.width;

        tempImage.data = new unsigned char[tempImage.width \* tempImage.height];

        for (int i = 0; i < img.height; ++i) {

            for (int j = 0; j < img.width; ++j) {

                tempImage.data[(tempImage.height - 1 - j) \* tempImage.width + i] = img.data[i \* img.width + j];

            }

        }

    }

    delete[] img.data; // Release the old image data

    img = tempImage; // Replace with the rotated image

}

void flip(Image &img, FlipDir dir) {

    if (dir == kNothing) return;

    for (int i = 0; i < img.height; ++i) {

        for (int j = 0; j < img.width / 2; ++j) {

            if (dir == kHorizontal) {

                swap(img.data[i \* img.width + j], img.data[i \* img.width + (img.width - 1 - j)]);

            } else if (dir == kVertical) {

                swap(img.data[j \* img.width + i], img.data[(img.height - 1 - j) \* img.width + i]);

            }

        }

    }

}

void applyFilters(Image &img, int delta, int angle, FlipDir flipDir) {

    Pipeline pipeline;

    if (delta != 0) {

        pipeline.filters.push\_back([delta](Image &img) { brightnessChange(img, delta); });

    }

    if (angle != 0) {

        pipeline.filters.push\_back([angle](Image &img) { rotate(img, angle); });

    }

    if (flipDir != kNothing) {

        pipeline.filters.push\_back([flipDir](Image &img) { flip(img, flipDir); });

    }

    for (auto &filter : pipeline.filters) {

        filter(img);

    }

}

int main() {

    Image image;

    image.width = 3;

    image.height = 3;

    image.data = new unsigned char[image.width \* image.height];

    for (int i = 0; i < image.width \* image.height; ++i) {

        image.data[i] = rand() % 256; // random Initialization

    }

    cout << "Original Image:\n" << image << endl;

    applyFilters(image, 0, 0, kNothing); // This means no filter is applied

    cout << "After no filters:\n" << image << endl;

    applyFilters(image, 10, 0, kHorizontal); // Increase brightness by 10 and flip horizontally

    cout << "After brightness +10 and flip horizontally:\n" << image << endl;

    applyFilters(image, 0, 90, kNothing); // Rotate by 90 degrees

    cout << "After rotating 90 degrees:\n" << image << endl;

    applyFilters(image, -20, 270, kVertical); // Decrease brightness by 20 and rotate by 270 degrees

    cout << "After brightness -20 and rotating 270 degrees:\n" << image << endl;

    delete[] image.data; // Clean up dynamically allocated memory

    return 0;

}