```
// Week5
// class

// rectangle.cpp
#include "rectangle.h"

// Constructor definition
Rectangle::Rectangle(double l, double w) : length(l), width(w) {}

// another way to define constructor
// Rectangle::Rectangle(double l, double w) {
//     length = l;
//     width = w;
// }S
// Getter for length
double Rectangle::getLength() const {
    return length;
}
// Getter for width
double Rectangle::getWidth() const {
    return width;
}
// Setter for length
void Rectangle::setLength(double l) {
    length = l;
}
// Setter for width
void Rectangle::setWidth(double w) {
    width = w;
}
// Function to calculate area
double Rectangle::area() const {
    return length * width;
}
// Function to calculate perimeter
double Rectangle::perimeter() const {
    return 2 * (length + width);
}

// rectangle.h
#pragma once
class Rectangle {
private:
    double length;
    double width;
public:
    // Constructor
    Rectangle(double l, double w);
    // Getters
    double getLength() const;
    double getWidth() const;
    // Setters
    void setLength(double l);
    void setWidth(double w);
    // Function to calculate area
    double area() const;
    // Function to calculate perimeter
    double perimeter() const;
};

// week6
// throw, catch, try, runtime_error
#include <iostream>
#include <stdexcept> // For standard exception types

int divide(int a, int b) {
    if (b == 0) {
        throw std::runtime_error("Division by zero is not allowed.");
    }
    return a / b;
}
int main() {
    int x = 10;
    int y = 0;
    try {
        int result = divide(x, y);
        std::cout << "Result: " << result << std::endl;
    }
    catch (const std::runtime_error& e) {
        std::cerr << "Caught an exception: " << e.what() << std::endl;
    }
    std::cout << "Program continues after exception handling." << std::endl;
    return 0;
}

// week7

// complex example
// complex.cpp
#include "complex.h"
#include <cmath>

using namespace std;
double absolute_value(complex const & c) {
    return sqrt(c.real() * c.real() + c.imag() * c.imag());
}
bool complex::operator<(complex const & b) {
    double abs_a = absolute_value(*this);
    double abs_b = absolute_value(b);
    return abs_a < abs_b;
}
// complex.h
#pragma once
#include <iostream>

class complex{
private:
    double re, im;
public:
    complex(double r, double i) : re{r}, im{i} {}
    complex(double r) : re{r}, im{0} {}
    complex() : re{0}, im{0} {}
    complex(const complex& c) : re{c.re}, im{c.im} {}

    double real() const {
        return re;
    }
    void real(double r) {
        re = r;
    }
    double  imag() const {
        return im;
    }
    void imag(double i) {
        im = i;
    }
    bool operator<(complex const & b);
};

// complex.cpp
#include "complex.h"
#include <cmath>

using namespace std;

double absolute_value(complex const & c) {
    return sqrt(c.real() * c.real() + c.imag() * c.imag());
}

bool complex::operator<(complex const & b) {
    double abs_a = absolute_value(*this);
    double abs_b = absolute_value(b);
    return abs_a < abs_b;
}
```

```
// destructor
class DynamicArray {
private:
    int* data;
    int size;
public:
    // Constructor
    DynamicArray(int s) : size(s) {
        data = new int[size];  // Dynamically allocate memory
        cout << "Array of size " << size << " created." << endl;
    }
    // Destructor
    ~DynamicArray() {
        delete[] data;  // Free dynamically allocated memory
        cout << "Array of size " << size << " destroyed." << endl;
    }
    // Method to set a value
    void setValue(int index, int value) {
        if (index >= 0 && index < size) {
            data[index] = value;
        }
    }
    // Method to get a value
    int getValue(int index) const {
        if (index >= 0 && index < size) {
            return data[index];
        }
        return -1;  // Return -1 for invalid index
    }
};
// Stream I/O
// input function
Vector read(istream& is) {
    Vector v;
    for (double d; is >> d;) {
        v.push_back(d);
    }
    return v;
}
// output function
void write(ostream& os, const Vector& v) {
    for (int i = 0; i != v.size(); ++i) {
        os << v[i] << '\n';
    }
}

// week8

#include <bits/stdc++.h>

using namespace std;

int main(){
    // copy operation
    // copy constructor
    // copy assignment operator

    // copy constructor
    class_name(const class_name &obj){
        // copy all the data members
    }

    // copy assignment operator
    class_name& operator=(const class_name &obj){
        // copy all the data members
        return *this;
    }


    // move operation
    // move constructor
    // move assignment operator

    // move constructor
    class_name(class_name &&obj){
        // move all the data members
    }

    // move assignment operator
    class_name& operator=(class_name &&obj){
        // move all the data members
        return *this;
    }

    // rule of 3
    // destructor
    // copy constructor
    // copy assignment operator

    // rule of 5
    // destructor
    // copy constructor
    // copy assignment operator
    // move constructor
    // move assignment operator

    // operator overloading

    // equality operator
    // relational operator
    // I/O operator

    // stream
    // stream operator

    // input stream operator
    // output stream operator

    // input stream operator
    istream & operator>>(istream & is, T & );

    // output stream operator
    ostream & operator<<(ostream & os, const T & );


    // fstream
    // ofstream
    ofstream ofs("file.txt");
    ofs << setw(4); // set width of the output
    ofs << setprecision(2); // set precision of the output
    ofs << "Age: " << 30 << endl;

    // ifstream
    ifstream ifs("file.txt");
    int x;
    ifs >> x;
    ifs >> noskipws; // do not skip white space
    ifs >> x;
    while(ifs >> x){
        cout << x << endl;
    }

    // stringstream
    // ostringstream
    ostringstream oss;
    oss << "Age: " << 30 << endl;
    string str = oss.str();
    // istringstream
    string word;
    char ch;
    istringstream iss{"Hello World"};
    iss >> word;
    iss.get(ch);
```

```
    iss.get(ch);
}


// week9

// friend function

//example
// login.h
#pragma once
#include <string>

class Login{

public:
    std::string username;
    Login(std::string const & username_, std::string const & password_)
        : username(username_), password(password_) {}

private:
    std::string password;
    friend bool operator==(const Login& l1, const Login& l2);
};
// login.cpp#include "login.h"

#include <string>

bool operator==(const Login& l1, const Login& l2){
    return l1.password == l2.password;
}




// week12

// random number generation

#include <bits/stdc++.h>

using namespace std;

int main(){
    // math function
    // accumulate
    vector<int> v = {1, 2, 3, 4, 5};
    cout << accumulate(v.begin(), v.end(), 0) << endl; // 15
    vector<double> v2 = {1.1, 2.2, 3.3, 4.4, 5.5};
    cout << accumulate(v2.begin(), v2.end(), double(1)) << endl; // 17.5

    // product of all elements in a vector
    std::vector<int> factors{2, 4, 29};
    double product = std::accumulate(
        factors.begin(), factors.end(), int{1},
        [](int a, int b) { return a * b; }
    );
    // Result: 232
    // starting from 1, 1*2 = 2, 2*4 = 8, 8*29 = 232

    // Concatenate
    std::string CommaSeparate(std::string const& left, std::string const& right) {
        return left + "," + right;
    }

    // ...
    std::vector<std::string> names{"Mal", "Kira", "Dax"};
    std::string line = std::accumulate(
        names.begin(), names.end(),
        std::string{"Josh"}, CommaSeparate
    );
    // Result: Josh,Mal,Kira,Dax
    // execution policy
    // parallel execution
    vector<int> v3(1000000, 1);
    cout << accumulate(v3.begin(), v3.end(), 0) << endl; // 1000000
    cout << accumulate(v3.begin(), v3.end(), 0, plus<int>(), execution::par) << endl; // 1000000

    // complex number
    std::vector<std::complex<int>> vec = {{2, 3}, {4, 5}, {10, -30}};
    int imag_sum = std::accumulate(
        vec.begin(), vec.end(), int{0},
        [](int x, std::complex<int> y) {
            return x + y.imag();
    });
    // Result: -22
    return 0;
}



#include <random>
#include <iostream>

int main() {
    // Create a random_device object to generate a random seed
    std::random_device rd;

    // Initialize the Mersenne Twister 64-bit random number generator with the seed
    std::mt19937_64 gen(rd());

    // Create a uniform integer distribution in the range [1, 6]
    std::uniform_int_distribution<> dist(1, 6);

    // Generate 20 random numbers using the distribution
    for (int i{0}; i < 20; ++i) {
        // Generate a random number and output it, followed by a space
        std::cout << dist(gen) << " ";  // Output a random number in the range [1, 6]
    }

    // Print a newline at the end
    std::cout << std::endl;

    // uniform_int_distribution (a,b) -> [a,b]
    // uniform_real_distribution (a,b) -> [a,b)
    // normal_distribution (mean, std_dev)
    // bernoulli_distribution (p) -> true with probability p
    // binomial_distribution (n, p) -> number of successes in n trials with probability p
    // poisson_distribution (mean) -> number of events in a fixed interval with mean rate
    // exponential_distribution (lambda) -> time between events with rate lambda
    // gamma_distribution (alpha, beta) -> sum of alpha exponential random variables with rate beta
    // weibull_distribution (a, b) -> time until event with shape a and scale b

    // valarray
    // valarray is a class template that represents an array of values.
    // It is designed to be a simpler and more efficient alternative to the vector class for some use cases.


    // c-style array
    int arr[5] = {1, 2, 3, 4, 5};
    int arr[5] = {1, 2, 3}; // {1, 2, 3, 0, 0}
    int arr[5] = {}; // {0, 0, 0, 0, 0}
    int arr[] = {1, 2, 3, 4, 5}; // size 5
    int arr[] = {1, 2, 3}; // size 3
    int arr[] = {}; // size 0
    int arr[5]; // uninitialized


    return 0;
}
```