Complex Numbers (C++) - Define a class Complex to represent complex numbers with member variables for real and imaginary parts. Overload the +, -, and \* operators for complex number addition, subtraction, and multiplication.

#include <iostream>

Using namespace std;

class Complex {

private:

double real;

double imag;

public:

Complex(double r = 0, double i = 0) : real(r), imag(i) {}

Complex operator + (const Complex& other) const {

return Complex(real + other.real, imag + other.imag);

}

Complex operator - (const Complex& other) const {

return Complex(real - other.real, imag - other.imag);

}

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

return Complex((real \* other.real - imag \* other.imag), (real \* other.imag + imag \* other.real));

}

void display() const {

if (imag < 0)

cout << real << " - " << -imag << "i" << std::endl;

else

cout << real << " + " << imag << "i" << std::endl;

}

};

int main() {

Complex c1(3, 2);

Complex c2(1, 7);

Complex c3 = c1 + c2;

Complex c4 = c1 - c2;

Complex c5 = c1 \* c2;

cout << "c1: ";

c1.display();

cout << "c2: ";

c2.display();

cout << "c1 + c2: ";

c3.display();

cout << "c1 - c2: ";

c4.display();

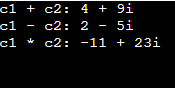
cout << "c1 \* c2: ";

c5.display();

return 0;

}

Output:



2) Point2D (C++) - Create a class Point2D with x and y coordinates. Overload the + operator to return a new Point2D object representing the sum of two points.

#include <iostream>

class Point2D {

private:

double x;

double y;

public:

Point2D(double x = 0, double y = 0) : x(x), y(y) {}

Point2D operator + (const Point2D& other) const {

return Point2D(x + other.x, y + other.y);

}

void display() const {

std::cout << "(" << x << ", " << y << ")" << std::endl;

}

};

int main() {

Point2D p1(3.5, 2.5);

Point2D p2(1.5, 4.0);

Point2D p3 = p1 + p2;

cout << "p1: ";

p1.display();

cout << "p2: ";

p2.display();

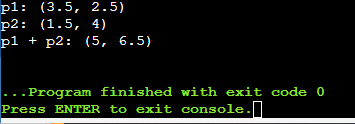
cout << "p1 + p2: ";

p3.display();

return 0;

}

Output:



3) Time (C++) - Design a class Time to store hours, minutes, and seconds. Overload the + operator to add two Time objects and return a new Time object with the combined duration.

#include <iostream>

class Time {

private:

int hours;

int minutes;

int seconds;

void normalize() {

if (seconds >= 60) {

minutes += seconds / 60;

seconds = seconds % 60;

}

if (minutes >= 60) {

hours += minutes / 60;

minutes = minutes % 60;

}

}

public:

Time(int h = 0, int m = 0, int s = 0) : hours(h), minutes(m), seconds(s) {

normalize();

}

Time operator + (const Time& other) const {

int newSeconds = seconds + other.seconds;

int newMinutes = minutes + other.minutes;

int newHours = hours + other.hours;

Time result(newHours, newMinutes, newSeconds);

result.normalize();

return result;

}

void display() const {

std::cout << (hours < 10 ? "0" : "") << hours << ":"

<< (minutes < 10 ? "0" : "") << minutes << ":"

<< (seconds < 10 ? "0" : "") << seconds << std::endl;

}

};

int main() {

Time t1(2, 45, 50);

Time t2(1, 20, 30);

Time t3 = t1 + t2;

std::cout << "t1: ";

t1.display();

std::cout << "t2: ";

t2.display();

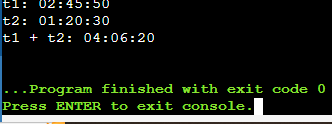
std::cout << "t1 + t2: ";

t3.display();

return 0;

}

Output:



4) Date (C++) - Implement a class Date with year, month, and day. Overload the comparison operators (== and !=) to compare two Date objects.

#include <iostream>

using namespace std;

class Date {

private:

int year;

int month;

int day;

public:

Date(int y = 0, int m = 0, int d = 0) : year(y), month(m), day(d) {}

bool operator == (const Date& other) const {

return (year == other.year && month == other.month && day == other.day);

}

bool operator != (const Date& other) const {

return !(\*this == other);

}

void display() const {

std::cout << year << "-" << (month < 10 ? "0" : "") << month << "-" << (day < 10 ? "0" : "") << day << std::endl;

}

};

int main() {

Date d1(2024, 6, 27);

Date d2(2023, 6, 27);

Date d3(2024, 6, 27);

cout << "d1: ";

d1.display();

cout << "d2: ";

d2.display();

cout << "d3: ";

d3.display();

if (d1 == d2) {

cout << "d1 is equal to d2" << endl;

} else {

cout << "d1 is not equal to d2" << endl;

}

if (d1 == d3) {

cout << "d1 is equal to d3" << endl;

} else {

cout << "d1 is not equal to d3" << endl;

}

if (d1 != d2) {

cout << "d1 is not equal to d2" << endl;

} else {

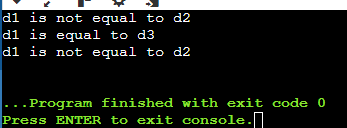
cout << "d1 is equal to d2" << endl;

}

return 0;

}

Output:



5) String Equality (C++) - Overload the equality operator (==) for a custom String class to compare string contents (not just memory addresses).

#include <iostream>

#include <cstring>

class String {

private:

char\* str;

public:

String() : str(nullptr) {}

String(const char\* s) {

if (s) {

str = new char[std::strlen(s) + 1];

std::strcpy(str, s);

} else {

str = nullptr;

}

}

String(const String& other) {

if (other.str) {

str = new char[std::strlen(other.str) + 1];

std::strcpy(str, other.str);

} else {

str = nullptr;

}

}

String(String&& other) noexcept : str(other.str) {

other.str = nullptr;

}

~String() {

delete[] str;

}

String& operator=(const String& other) {

if (this != &other) {

delete[] str;

if (other.str) {

str = new char[std::strlen(other.str) + 1];

std::strcpy(str, other.str);

} else {

str = nullptr;

}

}

return \*this;

}

String& operator=(String&& other) noexcept {

if (this != &other) {

delete[] str;

str = other.str;

other.str = nullptr;

}

return \*this;

}

bool operator==(const String& other) const {

if (str == nullptr && other.str == nullptr) {

return true;

}

if (str == nullptr || other.str == nullptr) {

return false;

}

return std::strcmp(str, other.str) == 0;

}

void print() const {

if (str) {

std::cout << str << std::endl;

} else {

std::cout << "null" << std::endl;

}

}

};

int main() {

String s1("Hello");

String s2("Hello");

String s3("World");

if (s1 == s2) {

std::cout << "s1 and s2 are equal" << std::endl;

} else {

std::cout << "s1 and s2 are not equal" << std::endl;

}

if (s1 == s3) {

std::cout << "s1 and s3 are equal" << std::endl;

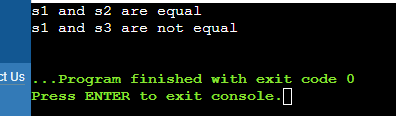
} else {

std::cout << "s1 and s3 are not equal" << std::endl;

}

return 0;

}



6) Area Calculation (C++) - Create a function calculateArea that can handle different shapes (e.g., rectangle, circle) by overloading it with parameters like width, height, or radius.

|  |  |  |
| --- | --- | --- |
| #include <iostream>  #include <cmath>  using namespace std;  double calculateArea(double width, double height) {  return width \* height;  }  double calculateArea(double radius) {  return M\_PI \* radius \* radius;  }  int main() {  double rectangleWidth = 5.0;  double rectangleHeight = 10.0;  double rectangleArea = calculateArea(rectangleWidth, rectangleHeight);  cout << "Area of the rectangle: " << rectangleArea << std::endl;  double circleRadius = 7.0;  double circleArea = calculateArea(circleRadius);  cout << "Area of the circle: " << circleArea <<endl;  return 0;  }  Output: |  |  |
|  |  |  |

7) Unit Conversion (Python) - Design a function convert that takes a value and a unit (e.g., meters, feet, Celsius, Fahrenheit) and converts it to another unit using appropriate conversion factors.

#include <iostream>

double convert(double value, const std::string& unit\_from, const std::string& unit\_to) {

if (unit\_from == "meters" && unit\_to == "feet") {

return value \* 3.28084;

} else if (unit\_from == "feet" && unit\_to == "meters") {

return value / 3.28084;

} else if (unit\_from == "Celsius" && unit\_to == "Fahrenheit") {

return (value \* 9/5) + 32;

} else if (unit\_from == "Fahrenheit" && unit\_to == "Celsius") {

return (value - 32) \* 5/9;

} else {

throw std::invalid\_argument("Unsupported conversion");

}

}

int main() {

double value = 10.0;

double converted\_value = convert(value, "meters", "feet");

std::cout << value << " meters is " << converted\_value << " feet" << std::endl;

double temperature = 25.0;

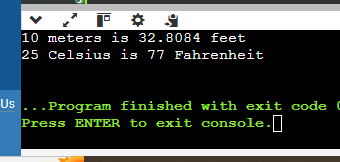
double converted\_temp = convert(temperature, "Celsius", "Fahrenheit");

std::cout << temperature << " Celsius is " << converted\_temp << " Fahrenheit" << std::endl;

return 0;

}

Output:



8) Statistics (C++) - Implement functions average, minimum, and maximum that can take an array of integers or doubles as input, depending on the function call.

#include <iostream>

#include <vector>

#include <algorithm>

double average(const std::vector<int>& nums) {

int sum = 0;

for (int num : nums) {

sum += num;

}

return static\_cast<double>(sum) / nums.size();

}

double average(const std::vector<double>& nums) {

double sum = 0.0;

for (double num : nums) {

sum += num;

}

return sum / nums.size();

}

int minimum(const std::vector<int>& nums) {

return \*std::min\_element(nums.begin(), nums.end());

}

double minimum(const std::vector<double>& nums) {

return \*std::min\_element(nums.begin(), nums.end());

}

int maximum(const std::vector<int>& nums) {

return \*std::max\_element(nums.begin(), nums.end());

}

double maximum(const std::vector<double>& nums) {

return \*std::max\_element(nums.begin(), nums.end());

}

int main() {

std::vector<int> int\_nums = {3, 1, 4, 1, 5, 9, 2, 6};

std::vector<double> double\_nums = {3.5, 1.2, 4.7, 2.1, 5.0, 1.8};

std::cout << "Average of integers: " << average(int\_nums) << std::endl;

std::cout << "Minimum of integers: " << minimum(int\_nums) << std::endl;

std::cout << "Maximum of integers: " << maximum(int\_nums) << std::endl;

std::cout << "Average of doubles: " << average(double\_nums) << std::endl;

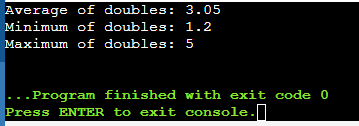
std::cout << "Minimum of doubles: " << minimum(double\_nums) << std::endl;

std::cout << "Maximum of doubles: " << maximum(double\_nums) << std::endl;

return 0;

}

Output:



9) String Formatting (C++) - Write overloaded functions formatString that can take a format string and different data types (e.g., int, double, string) to create formatted output strings.

#include <iostream>

#include <sstream>

#include <iomanip>

std::string formatString(const std::string& format, int value) {

std::ostringstream oss;

oss << format << " " << value;

return oss.str();

}

std::string formatString(const std::string& format, double value) {

std::ostringstream oss;

oss << std::fixed << std::setprecision(2) << format << " " << value;

return oss.str();

}

std::string formatString(const std::string& format, const std::string& value) {

return format + " " + value;

}

int main() {

int intValue = 42;

double doubleValue = 3.14;

std::string stringValue = "Hello, World!";

std::cout << formatString("Integer value:", intValue) << std::endl;

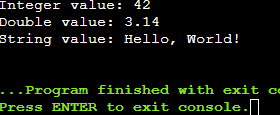
std::cout << formatString("Double value:", doubleValue) << std::endl;

std::cout << formatString("String value:", stringValue) << std::endl;

return 0;

}

Output:



10) Math Functions (C++) - Create overloaded functions factorial and power that can handle integer and floating-point input for calculating factorials and raising a number to a power.

#include <iostream>

#include <cmath>

// Factorial for integers

int factorial(int n) {

return (n == 0 || n == 1) ? 1 : n \* factorial(n - 1);

}

// Factorial for floats (approximation)

double factorial(double n) {

return tgamma(n + 1);

}

// Power for integers

int power(int base, int exp) {

return std::pow(base, exp);

}

// Power for floats

double power(double base, double exp) {

return std::pow(base, exp);

}

int main() {

// Factorial of integers

std::cout << "Factorial of 5: " << factorial(5) << std::endl;

// Factorial of floats

std::cout << "Factorial of 5.5: " << factorial(5.5) << std::endl;

// Power of integers

std::cout << "2 raised to the power of 3: " << power(2, 3) << std::endl;

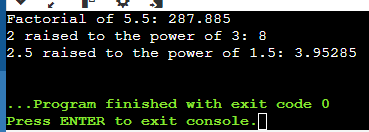
// Power of floats

std::cout << "2.5 raised to the power of 1.5: " << power(2.5, 1.5) << std::endl;

return 0;

}

Output:



11) Polynomial Addition (C++) - Define a class Polynomial to represent polynomials with terms (coefficient and exponent). Overload the + operator to add two Polynomial objects and return a new Polynomial with the combined terms.

#include <iostream>

#include <vector>

#include <algorithm>

class Term {

public:

int coefficient;

int exponent;

Term(int c, int e) : coefficient(c), exponent(e) {}

};

class Polynomial {

private:

std::vector<Term> terms;

public:

void addTerm(int coefficient, int exponent) {

terms.emplace\_back(coefficient, exponent);

}

Polynomial operator+(const Polynomial& other) const {

Polynomial result;

for (const auto& term : terms) {

result.addTerm(term.coefficient, term.exponent);

}

for (const auto& term : other.terms) {

result.addTerm(term.coefficient, term.exponent);

}

result.combineLikeTerms();

return result;

}

void combineLikeTerms() {

std::sort(terms.begin(), terms.end(), [](const Term& a, const Term& b) {

return a.exponent > b.exponent;

});

std::vector<Term> combined;

for (const auto& term : terms) {

if (!combined.empty() && combined.back().exponent == term.exponent) {

combined.back().coefficient += term.coefficient;

} else {

combined.push\_back(term);

}

}

terms = combined;

}

void print() const {

for (size\_t i = 0; i < terms.size(); ++i) {

std::cout << terms[i].coefficient << "x^" << terms[i].exponent;

if (i != terms.size() - 1) {

std::cout << " + ";

}

}

std::cout << std::endl;

}

};

int main() {

Polynomial p1;

p1.addTerm(3, 2);

p1.addTerm(5, 1);

Polynomial p2;

p2.addTerm(4, 2);

p2.addTerm(2, 0);

Polynomial p3 = p1 + p2;

std::cout << "P1: ";

p1.print();

std::cout << "P2: ";

p2.print();

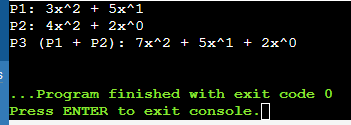
std::cout << "P3 (P1 + P2): ";

p3.print();

return 0;

}

Output:



12) Inventory Management (C++) - Implement a class Item with properties like name, price, and quantity. Overload the << operator for easy printing of item details to the console.

#include <iostream>

#include <string>

class Item {

private:

std::string name;

double price;

int quantity;

public:

Item(const std::string& itemName, double itemPrice, int itemQuantity)

: name(itemName), price(itemPrice), quantity(itemQuantity) {}

std::string getName() const { return name; }

double getPrice() const { return price; }

int getQuantity() const { return quantity; }

friend std::ostream& operator<<(std::ostream& os, const Item& item) {

os << "Item Name: " << item.name << "\n"

<< "Price: $" << item.price << "\n"

<< "Quantity: " << item.quantity;

return os;

}

};

int main() {

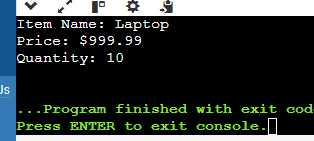
Item item1("Laptop", 999.99, 10);

std::cout << item1 << std::endl;

return 0;

}

Output:



13) Rectangle Class: Define a class Rectangle with member variables for width and height. Overload the + operator to return a new Rectangle object representing the sum of the areas of two rectangles.

#include <iostream>

class Rectangle {

public:

Rectangle(double width, double height) : width(width), height(height) {}

double getArea() const {

return width \* height;

}

Rectangle operator+(const Rectangle& other) const {

double totalArea = this->getArea() + other.getArea();

double newWidth = this->width;

double newHeight = totalArea / newWidth;

return Rectangle(newWidth, newHeight);

}

void print() const {

std::cout << "Rectangle(width=" << width << ", height=" << height << ")" << std::endl;

}

private:

double width;

double height;

};

int main() {

Rectangle rect1(4, 5);

Rectangle rect2(3, 7);

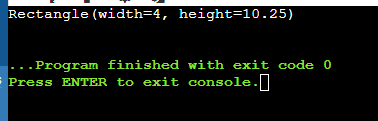
Rectangle rect3 = rect1 + rect2;

rect3.print(); // Output will be a new Rectangle with the combined area

return 0;

}

Output:



14) Money Class: Design a class Money to store currency amount and type (e.g., USD, EUR). Overload the comparison operators (==, !=, <, >, <=, >=) for Money objects, considering currency types and exchange rates.

#include <iostream>

#include <string>

#include <unordered\_map>

class Money {

public:

Money(double amount, const std::string& currency) : amount(amount), currency(currency) {}

// Overload the == operator

bool operator==(const Money& other) const {

return convertToBase(amount, currency) == convertToBase(other.amount, other.currency);

}

// Overload the != operator

bool operator!=(const Money& other) const {

return !(\*this == other);

}

// Overload the < operator

bool operator<(const Money& other) const {

return convertToBase(amount, currency) < convertToBase(other.amount, other.currency);

}

// Overload the <= operator

bool operator<=(const Money& other) const {

return \*this < other || \*this == other;

}

// Overload the > operator

bool operator>(const Money& other) const {

return !(\*this <= other);

}

// Overload the >= operator

bool operator>=(const Money& other) const {

return !(\*this < other);

}

void print() const {

std::cout << amount << " " << currency << std::endl;

}

private:

double amount;

std::string currency;

static double convertToBase(double amount, const std::string& currency) {

static std::unordered\_map<std::string, double> exchangeRates = {

{"USD", 1.0},

{"EUR", 1.1}, // Example: 1 EUR = 1.1 USD

{"GBP", 1.3}, // Example: 1 GBP = 1.3 USD

// Add more currencies as needed

};

return amount \* exchangeRates[currency];

}

};

int main() {

Money money1(100, "USD");

Money money2(90, "EUR");

money1.print(); // Output: 100 USD

money2.print(); // Output: 90 EUR

if (money1 == money2) {

std::cout << "money1 is equal to money2" << std::endl;

} else {

std::cout << "money1 is not equal to money2" << std::endl;

}

if (money1 > money2) {

std::cout << "money1 is greater than money2" << std::endl;

} else {

std::cout << "money1 is not greater than money2" << std::endl;

}

return 0;

}

Output:

