#include <math.h>

#include<stdio.h>

#include<stdlib.h>

typedef struct {

double real;

double imag;

} complex\_t;

complex\_t \*

complex\_add(complex\_t \*a, complex\_t \*b) {

complex\_t \*c = malloc(sizeof(complex\_t));

c->real = a->real + b->real;

c->imag = a->imag + b->imag;

return c;

}

complex\_t \*

complex\_multiply(complex\_t \*a, complex\_t \*b) {

complex\_t \*c = malloc(sizeof(complex\_t));

c->real = a->real \* b->real - a->imag \* b->imag;

c->imag = a->real \* b->imag + a->imag \* b->real;

return c;

}

void

complex\_free(complex\_t \*c) {

free(c);

}

// Usage Example

int main() {

complex\_t num1 = {1.0, 2.0};

complex\_t num2 = {3.0, 4.0};

complex\_t \*result\_add = complex\_add(&num1, &num2);

complex\_t \*result\_mul = complex\_multiply(&num1, &num2);

printf("Sum: %.2f + %.2fi\n", result\_add->real, result\_add->imag);

printf("Product: %.2f + %.2fi\n", result\_mul->real, result\_mul->imag);

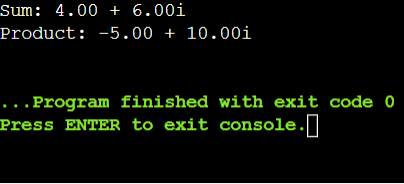
complex\_free(result\_add);

complex\_free(result\_mul);

return 0;

}

**Output:**



**Add two objects:**

#include <stdio.h>

typedef struct {

int a,b;

}Point;

Point addP(Point p, Point q) {

Point result;

result.a = p.a + q.a;

result.b = p.b + q.b;

return result;

}

int main()

{

Point a1= {1,8};

Point a2 = {7, 5};

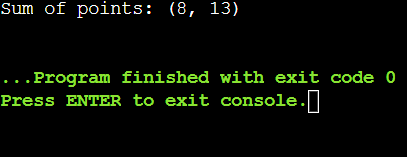
Point sum = addP(a1, a2);

printf("Sum of points: (%d, %d)\n", sum.a, sum.b);

return 0;

}

**Output:**



**Program to overload the unary operator++**

#include<iostream>

using namespace std;

class Test {

private:

int num;

public:

Test():num(8){}

void operator ++(){

num = num +2;

}

void print(){

cout<<"The count is:"<<num;

}

};

int main(){

Test tt;

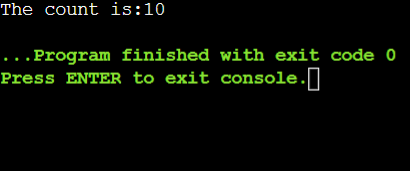
++tt;

tt.print();

return 0;

}

**Output:**



**Program to overload the binary operators**

#include <iostream>

using namespace std;

class A{

int x;

public:

A(){}

A(int i)

{

x=i;

}

void operator+(A);

void display();

};

void A :: operator+(A a)

{

int m = x+a.x;

cout<<"The result of the addition of two objects is:"<<m;

}

int main()

{

A a1(5);

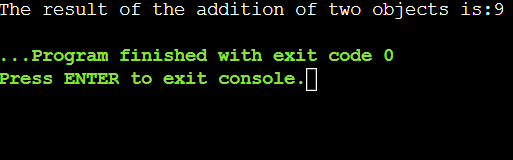
A a2(4);

a1+a2;

return 0;

}

**Output:**



#include <iostream>

using namespace std;

class A {

public:

int x;

A() {}

A(int i) {

x = i;

}

void operator+(A a) {

int m = x + a.x;

cout << "The result of the addition of two objects is: " << m << endl;

}

void operator-(A a) {

int m = x - a.x;

cout << "The result of the subtraction of two objects is: " << m << endl;

}

void operator\*(A a) {

int m = x \* a.x;

cout << "The result of the multiplication of two objects is: " << m << endl;

}

void operator/(A a) {

if (a.x != 0) {

int m = x / a.x;

cout << "The result of the division of two objects is: " << m << endl;

} else {

cout << "Division by zero is not allowed." << endl;

}

}

void display() {

cout << "Value of x: " << x << endl;

}

};

int main() {

A a1(20);

A a2(10);

a1 + a2;

a1 - a2;

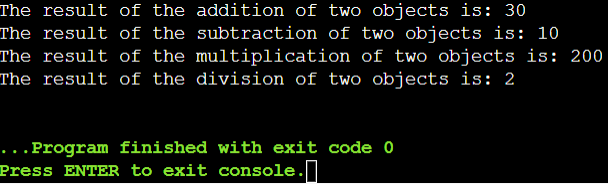
a1 \* a2;

a1 / a2;

return 0;

}

**Output:**



#include <iostream>

using namespace std;

class A {

public:

int x;

A() {}

A(int i) {

x = i;

}

A operator+(A o) {

A addition\_result;

addition\_result.x = x + o.x;

return addition\_result;

}

A operator-(A o) {

A subtraction\_result;

subtraction\_result.x = x - o.x;

return subtraction\_result;

}

A operator\*(A o) {

A multiplication\_result;

multiplication\_result.x = x \* o.x;

return multiplication\_result;

}

A operator/(A o) {

if (o.x != 0) {

A division\_result;

division\_result.x = x / o.x;

return division\_result;

} else {

cout << "Division by zero is not allowed." << endl;

return A();

}

}

void display() {

cout << "Value of x: " << x << endl;

}

};

int main() {

A a1(20);

A a2(10);

A c1 = a1 + a2;

A c2 = a1 - a2;

A c3 = a1 \* a2;

A c4 = a1 / a2;

cout << "c1: ";

c1.display();

cout << "c2: ";

c2.display();

cout << "c3: ";

c3.display();

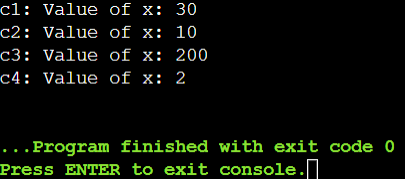
cout << "c4: ";

c4.display();

return 0;

}

**Output:**



**Program to overload the unary operator --**

#include <iostream>

using namespace std;

class Test {

private:

int num;

public:

Test() : num(8) {}

void operator --() {

num = num - 2;

}

void print() {

cout << "The count is: " << num;

}

};

int main() {

Test tt;

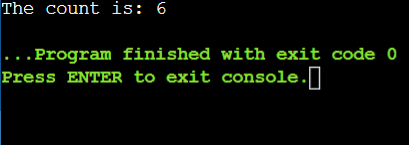
--tt;

tt.print();

return 0;

}

**Output:**



**Operator Overloading**

**1.What are the benefits and drawbacks of operator overloading?**

**Benefits:**

* Reduces the amount of code you need to write for common operations.
* Once defined, operators can be used across different instances and scenarios, saving time and effort.
* Allows you to write clearer, more understandable code.
* Once defined, operators can be used across different instances and scenarios, saving time and effort.
* Properly overloaded operators can sometimes lead to more optimized code execution, especially when dealing with complex arithmetic or data structures.

**Drawbacks:**

* Overloading can make code harder to understand if not used carefully.
* Operators might not always behave as expected, leading to confusion.
* Requires a good understanding of both C++ and object-oriented principles.
* Might not always be as fast or efficient as built-in operations.
* If not used wisely, it can lead to unexpected behaviors and bugs.

**2.Can you overload the assignment operator (=) in C++? If so, how would you ensure proper behavior?**

Yes, you can change how objects are assigned using the = sign in C++. To make sure it works right:

1. Return \*this so you can link assignments like a = b = c.
2. Check if you're not assigning to yourself with if (this != &other).
3. Copy each part of the object from the right side (other) to the left (this) to keep all the data right.

**3.Explain the difference between member function and non-member (friend) function overloading for operators.**

|  |  |
| --- | --- |
| **Member Function** | **Non-Member Function** |
| Defined inside the class. | Defined outside the class. |
| The unary operator does not take any explicit parameter. | The unary operator takes at least one explicit parameter. |
| In these, there is no such keyword required. | The Friend keyword is generally used to declare a function as a friend function. |
| Inherited by derived classes. | Not inherited by derived classes |
| Automatically receives ‘this’ pointer | Doesn’t automatically access ‘this’ pointer |

**4.Design a class Vector2D and overload the arithmetic operators (+, -, \*, /) for vector addition, subtraction, scalar multiplication, and division (by a scalar).**

#include <iostream>

using namespace std;

class Vector2D {

public:

double x, y;

Vector2D() : x(0.0), y(0.0) {}

Vector2D(double x, double y) : x(x), y(y) {}

Vector2D add(Vector2D v) {

return Vector2D(x + v.x, y + v.y);

}

Vector2D subtract(Vector2D v) {

return Vector2D (x - v.x, y - v.y);

}

Vector2D multiply(double scalar) {

return Vector2D(x \* scalar, y \* scalar);

}

Vector2D divide(double scalar) {

if (scalar != 0.0)

return Vector2D(x / scalar, y / scalar);

else {

cout << "Error: Division by zero!" << endl;

return Vector2D();

}

}

void display() {

cout << "Vector2D: (" << x << ", " << y << ")" << endl;

}

};

int main() {

Vector2D v1(7.5, 6.5);

Vector2D v2(2.5, 3.0);

Vector2D sum = v1.add(v2);

cout << "Sum: ";

sum.display();

Vector2D difference = v1.subtract(v2);

cout << "Difference: ";

difference.display();

Vector2D scaled = v1.multiply(2.0);

cout << "Scaled: ";

scaled.display();

Vector2D divided = v1.divide(2.0);

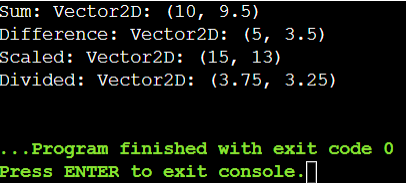
cout << "Divided: ";

divided.display();

return 0;

}

**Output:**



**5.Is it possible to overload the comparison operators (==, !=, <, >, <=, >=) for custom classes? If so, what considerations should be taken into account?**

Yes, it is possible to overload comparison operators (==, !=, <, >, <=, >=) for custom classes in C++.

**Considerations for Overloading Comparison Operators**

**Logical Consistency**: Ensure that the overloaded operators provide a meaningful and logical comparison for the objects of your class, based on the attributes that make sense for comparison.

**Efficiency**: Implement the operators efficiently, especially if they will be used frequently, to avoid performance overhead.

**Reflexivity**: The equality operator (==) should be reflexive. a == a should always be true.

**Commutativity**: For relational operators, ensure that the operation makes sense in both directions. For instance, if a < b is true, then b > a should also be true.

**Exception Safety**: Ensure that your operator functions do not throw exceptions or handle exceptions properly to maintain program stability.

**6.Can you overload the stream insertion (<<) and extraction (>>) operators for your Vector2D class to allow easy printing and reading from streams?**

Yes, you can overload the stream insertion (‘<<’) and extraction (‘>>’) operators for your Vector2D class in C++. Overloading operator<< allows you to define how Vector2D objects are formatted when streamed to an output, such as std::cout. Similarly, overloading operator>> enables you to specify how Vector2D objects are read from input streams, like std::cin, making input and output operations with Vector2D objects seamless and intuitive

**7.Describe a scenario where overloading the logical operators (&&, ||, !) for a custom class might be useful.**

Imagine you have a ‘ShoppingCart’ class with items and discounts. Overloading ‘&&’ could help check if two carts have overlapping items. Using ‘||’ could merge carts. ‘!’ could indicate if a cart has no items. This makes cart management straightforward and intuitive in your online store system.

**8.Discuss the potential ambiguity that could arise when overloading the subscript operator ([]) for a class. How can this ambiguity be resolved?**

When overloading the subscript operator (‘[]’) for a class, ambiguity can arise if the operator is overloaded without considering the context in which it will be used.

Here are some potential sources of ambiguity and ways to resolve them:

**1.Ambiguity in Return Type:** If ‘operator[]’ returns a reference (‘T&’), ambiguity can arise if it's unclear whether modifications affect the original object or a copy. **Resolution:** Use ‘const T&’ for read-only access and ‘T&’ for mutable access, clearly distinguishing between them. **2.Ambiguity in Index Type:** Overloading ‘operator[]’ with multiple parameter types (e.g., ‘int’ vs. ‘size\_t’) can confuse the compiler about which overload to use. **Resolution:** Stick to a consistent index type (preferably’ size\_t’ for safety) or use type traits to differentiate overload choices.

**9.Can operator overloading be used to implement the concept of immutability (unchanging state) for a class? Explain your answer.**

Yes, operator overloading can help implement immutability for a class by returning const references or const objects from overloaded operators like +, -, \*, and /. This ensures that operations do not modify the original object's state, promoting safer and predictable usage where objects remain unchanged after operations are performed on them.

**10.When overloading operators, what are some best practices to ensure code clarity and maintainability?**

When overloading operators in C++, consider these best practices to ensure code clarity and maintainability:

**1.Follow Conventions**: Stick to expected behaviors for each operator (e.g., + for addition, << for stream output) to maintain consistency and readability for others using your code.

**2.** **Explain What They Do**: Write comments or documentation to explain what each overloaded operator does. This helps others understand your code better.

**3**.**Use const Correctly**: If an operator doesn't change the object, mark it as const. This tells the compiler it won't modify anything.

**4.Handle Special Cases**: Think about unusual situations or bad inputs. Make sure your operators can handle them without crashing or giving wrong results.

**5. Only Overload What Makes Sense**: Don't overload operators for things that don't fit. Keep it logical and clear so others can easily understand how to use your class.

**6.** **Test Well**: Test your overloaded operators with different situations to make sure they work right. This helps catch mistakes early and ensures your code behaves as expected.

**11.What is the core concept behind function overloading?**

* Function overloading allows you to define multiple functions with the same name within the same scope.
* Each overloaded function must differ in its parameter list—either in the number of parameters, their types, or their order.
* The correct function to call is determined at compile-time based on the arguments provided during the function call.
* It provides flexibility by allowing you to create functions that perform similar tasks but can handle different types of inputs or different numbers of inputs.
* It avoids name clashes when different functions logically perform the same task but on different types or numbers of parameters.

**12.How does the compiler differentiate between overloaded functions with the same name?**

* **Number of Parameters**: If one function expects two inputs and another expects three, they're different.
* **Types of Parameters**: If one function takes integers and another takes floats, they're different.
* **Order of Parameters**: If one function takes (int, float) and another takes (float, int), they're different.
* **Const or Non-Const Parameters**: If one function has a parameter that can't change (const) and another doesn't, they're different.
* **Default Values for Parameters**: If one function has default values and another doesn't, they're different.

**13. Can functions with different return types be overloaded? Explain your reasoning**

No, functions with different return types cannot be overloaded in C++. This is because function overloading is determined by the function's name and parameter types, not by the return type.

**Reasoning**: When you overload a function, the compiler decides which version to use based on the function's name and the types of its parameters. If two functions had the same name and parameters but different return types, the compiler wouldn't know which one to choose. This could cause confusion and errors in your code.

**14.Design a function printValue that can handle different data types (e.g., int, double, std::string) by overloading it with appropriate parameter lists.**

#include <iostream>

#include <string>

using namespace std;

void printValue(int value) {

cout << "Integer value: " << value <<endl;

}

void printValue(double value) {

cout << "Double value: " << value <<endl;

}

void printValue(const std::string& value) {

cout << "String value: " << value <<endl;

}

int main() {

int intValue = 20;

double doubleValue = 7.34;

string stringValue = "Kavya";

printValue(intValue);

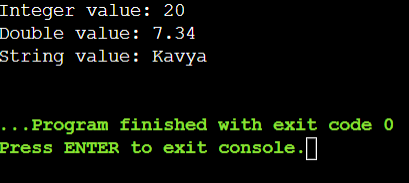
printValue(doubleValue);

printValue(stringValue);

return 0;

}

**Output:**



**15.Discuss the advantages and disadvantages of using default arguments in overloaded functions.**

**Advantages:**

* **Simplicity**: Default arguments simplify function calls by reducing the need for multiple overloaded functions.
* **Convenience**: Users can choose to omit arguments that have default values, making function calls more concise.
* **Readability**: Default arguments improve code readability by consolidating related functionality into a single function definition

**Disadvantages:**

* **Reduced Flexibility:**Default arguments limit flexibility, as users cannot easily override them without defining a new function overload.
* **Ambiguity in Function Resolution:**Multiple function definitions with different default values can create ambiguity in determining which function will be called.
* **Debugging Complexity:**Debugging can become more challenging because default arguments may obscure which function overload is being executed.

**16.In the context of function overloading, explain the concept of argument promotion and implicit type conversion.**

**Argument Promotion:**

1. **Definition**: Argument promotion happens when smaller types of data, like char or short, are automatically turned into larger types, such as int, when passed to a function.
2. **Example**: If a function can handle both int and double arguments, and you give it a char, the char gets promoted to an int before the function figures out which version to use.
3. **Reason**: It helps keep things consistent and avoids mistakes by making sure all arguments fit the types the function expects.

**Implicit Type Conversion:**

1. **Definition**: Implicit type conversion means the compiler changes one type of data into another type automatically when needed.
2. **Example**: If a function can handle both int and double, and you give it a float, the compiler will change the float into a double before calling the right version of the function.
3. **Reason**: It makes programming easier by letting you use different types of data without always having to manually change them.

**17.When might it be a better idea to use separate functions with descriptive names instead of overloading a single function?**

1. **Different Jobs**: If each version of the function does a different job or works in a different way, separate names make it clear what each function does.
2. **Avoid Confusion**: When using one function with different setups (like default options), it can confuse which version will be used, so separate names avoid this.
3. **Easier Reading**: Having distinct names makes the code easier to read and understand, especially for someone new to the code or coming back to it later.
4. **Organized Code**: It helps keep the code organized and makes maintenance easier because each function does one specific thing clearly.
5. **Clear Instructions**: For APIs or when writing instructions, clear function names tell users exactly what each function does without needing extra explanations.

**18.Can function overloading be used to achieve polymorphism (the ability to treat objects of different derived classes in a similar way)? Explain.**

Yes, function overloading can contribute to achieving polymorphism in C++. Polymorphism refers to the ability to treat objects of different derived classes through a common interface or base class pointer or reference.

**19.Describe a scenario where overloading a function with a variable number of arguments (varargs) could be beneficial.**

Imagine you're designing a logging system for a game. You want a function log that can handle messages of different types and complexities, from simple informational messages to detailed error logs with multiple pieces of data. By overloading log with varargs, you can create versions that accept different numbers and types of arguments, allowing developers to log messages with varying levels of detail and information without needing separate functions for each scenario. This makes the logging system flexible, efficient, and easy to use across different parts of the game code.

**Operator Overloading**

**1.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 imaginary;

public:

Complex(double real = 0.0, double imaginary = 0.0) : real(real), imaginary(imaginary) {}

double getReal() const {

return real;

}

double getImaginary() const {

return imaginary;

}

Complex operator+(const Complex& other) {

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

}

Complex operator-(const Complex& other) {

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

}

Complex operator\*(const Complex& other) {

double resultReal = (real \* other.real) - (imaginary \* other.imaginary);

double resultImaginary = (real \* other.imaginary) + (imaginary \* other.real);

return Complex(resultReal, resultImaginary);

}

void print() const {

cout << real << " + " << imaginary << "i" << endl;

}

};

int main() {

Complex c1(5.0, 6.0);

Complex c2(2.0, 2.0);

Complex sum = c1 + c2;

cout << "Sum: ";

sum.print();

Complex diff = c1 - c2;

cout << "Difference: ";

diff.print();

Complex product = c1 \* c2;

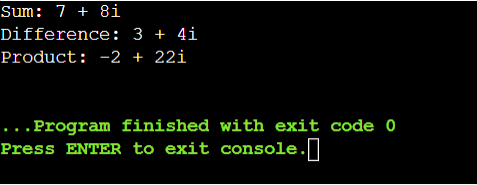
cout << "Product: ";

product.print();

return 0;

}

**Output:**



**2.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>

using namespace std;

class Point2D {

private:

double x;

double y;

public:

Point2D(double x = 0.0, double y = 0.0) {

this->x = x;

this->y = y;

}

double getX() {

return x;

}

double getY() {

return y;

}

Point2D operator+(Point2D& q) {

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

}

void print() {

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

}

};

int main() {

Point2D p1(7.0, 8.0);

Point2D p2(3.0, 2.0);

Point2D sum = p1 + p2;

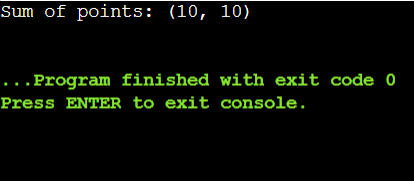
cout << "Sum of points: ";

sum.print();

return 0;

}

**Output:**



**3.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>

using namespace std;

class Time {

private:

int hours;

int minutes;

int seconds;

public:

Time(int hours = 0, int minutes = 0, int seconds = 0) {

this->hours = hours;

this->minutes = minutes;

this->seconds = seconds;

}

Time operator+(Time& p) {

int totalSeconds = seconds + p.seconds;

int carryMinutes = totalSeconds / 60;

int remainingSeconds = totalSeconds % 60;

int totalMinutes = minutes + p.minutes + carryMinutes;

int carryHours = totalMinutes / 60;

int remainingMinutes = totalMinutes % 60;

int totalHours = hours + p.hours + carryHours;

return Time(totalHours, remainingMinutes, remainingSeconds);

}

void print() {

cout << hours << " hours, " << minutes << " minutes, " << seconds << " seconds" <<endl;

}

};

int main() {

Time t1(2, 30, 45);

Time t2(1, 15, 20);

Time sum = t1 + t2;

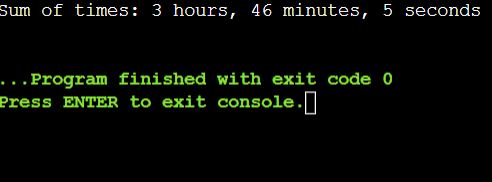
cout << "Sum of times: ";

sum.print();

return 0;

}

**Output:**



**4.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 year = 0, int month = 0, int day = 0) : year(year), month(month), day(day) {}

int getYear() {

return year;

}

int getMonth() {

return month;

}

int getDay() {

return day;

}

bool operator==(Date& t) {

return year == t.year && month == t.month && day == t.day;

}

bool operator!=(Date& t) {

return !(\*this ==t);

}

};

int main() {

Date date1(2023, 6, 27);

Date date2(2024, 6, 27);

Date date3(2024, 6, 27);

if (date1 == date2) {

cout << "Date1 and Date2 are equal." <<endl;

} else {

cout << "Date1 and Date2 are not equal." <<endl;

}

if (date1 != date3) {

cout << "Date1 and Date3 are not equal." <<endl;

} else {

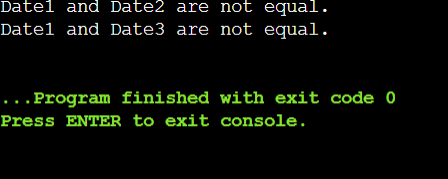
cout << "Date1 and Date3 are equal." <<endl;

}

return 0;

}

**Output:**



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

#include <iostream>

#include <cstring>

using namespace std;

class String {

private:

char\* str;

public:

String(const char\* s = "") {

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

strcpy(str, s);

}

String() {

delete[] str;

}

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

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

}

};

int main() {

String s1("Hello");

String s2("Hello");

String s3("World");

if (s1 == s2)

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

else

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

if (s1 == s3)

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

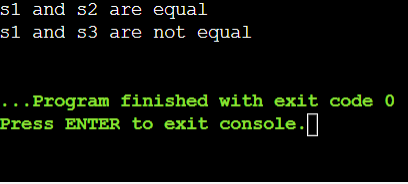
else

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

return 0;

}

**Output:**



**Function Overloading**

**6.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;

float calculateArea(float width, float height) {

return width \* height;

}

float calculateArea(float radius) {

return M\_PI \* radius \* radius;

}

int main() {

float rectangleArea = calculateArea(6.0f, 7.0f);

cout << "Area of rectangle: " << rectangleArea <<endl;

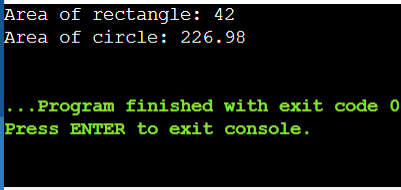
float circleArea = calculateArea(8.5f);

cout << "Area of circle: " << circleArea <<endl;

return 0;

}

**Output:**



**7.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>

using namespace std;

double convert(double value, const string& from\_unit, const string& to\_unit) {

if (from\_unit == "meters" && to\_unit == "feet") {

return value \* 3.281;

} else if (from\_unit == "feet" && to\_unit == "meters") {

return value / 3.281;

} else if (from\_unit == "Celsius" && to\_unit == "Fahrenheit") {

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

} else if (from\_unit == "Fahrenheit" && to\_unit == "Celsius") {

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

} else {

cout << "Unsupported conversion" << endl;

return 0.0;

}

}

int main() {

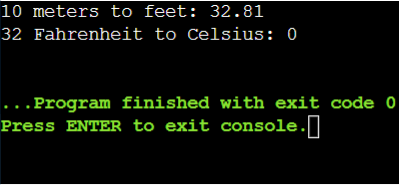
cout << "10 meters to feet: " << convert(10, "meters", "feet") << endl; // Output: 32.81 feet

cout << "32 Fahrenheit to Celsius: " << convert(32, "Fahrenheit", "Celsius") << endl; // Output: 0 Celsius

return 0;

}

**Output:**



**8.Implement functions average, minimum, and maximum that can take an array of integers or doubles as input, depending on the function call.**

#include <iostream>

using namespace std;

template <typename T>

class A{

private:

T\* arr;

int size;

public:

A(T\* arr, int size) : arr(arr), size(size) {}

double average() {

double sum = 0.0;

for (int i = 0; i < size; ++i) {

sum += arr[i];

}

return sum / size;

}

T minimum() {

T minVal = arr[0];

for (int i = 1; i < size; ++i) {

if (arr[i] < minVal) {

minVal = arr[i];

}

}

return minVal;

}

T maximum() {

T maxVal = arr[0];

for (int i = 1; i < size; ++i) {

if (arr[i] > maxVal) {

maxVal = arr[i];

}

}

return maxVal;

}

};

int main() {

int intArray[] = {3, 1, 4, 1, 5, 9, 2, 6};

int intSize = sizeof(intArray) / sizeof(intArray[0]);

A<int> AInt(intArray, intSize);

cout << "Average: " << AInt.average() << endl;

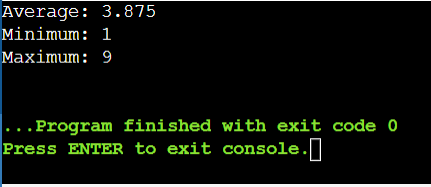
cout << "Minimum: " << AInt.minimum() << endl;

cout << "Maximum: " << AInt.maximum() << endl;

return 0;

}

**Output:**



**9.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 <iomanip>

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

std::string result = format;

size\_t pos = result.find("%");

if (pos != std::string::npos) {

result.replace(pos, 2, std::to\_string(value));

}

return result;

}

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

std::string result = format;

size\_t pos = result.find("%");

if (pos != std::string::npos) {

result.replace(pos, 2, std::to\_string(value));

}

return result;

}

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

std::string result = format;

size\_t pos = result.find("%");

if (pos != std::string::npos) {

result.replace(pos, 2, value);

}

return result;

}

int main() {

int intValue = 123;

double doubleValue = 45.6789;

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

std::cout << "Formatted int: " << formatString("%d", intValue) << std::endl;

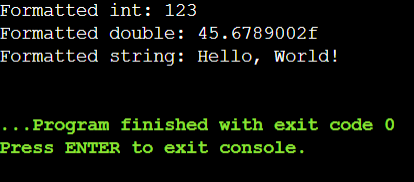
std::cout << "Formatted double: " << formatString("%.2f", doubleValue) << std::endl;

std::cout << "Formatted string: " << formatString("%s", stringValue) << std::endl;

return 0;

}

**Output:**



**10**.**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>

using namespace std;

unsigned long long factorial(int n) {

if (n < 0) {

cerr << "Error: Factorial is not defined for negative numbers." <<endl;

return 0;

}

unsigned long long result = 1;

for (int i = 2; i <= n; ++i) {

result \*= i;

}

return result;

}

double factorial(double n) {

if (n < 0) {

cerr << "Error: Factorial is not defined for negative numbers." << endl;

return 0.0;

}

double result = 1.0;

for (double i = 2.0; i <= n; ++i) {

result \*= i;

}

return result;

}

unsigned long long power(int base, int exponent) {

if (exponent < 0) {

cerr << "Error: Power operation is not defined for negative exponents." <<endl;

return 0;

}

unsigned long long result = 1;

for (int i = 0; i < exponent; ++i) {

result \*= base;

}

return result;

}

double power(double base, int exponent) {

if (exponent == 0) {

return 1.0;

}

double result = 1.0;

for (int i = 0; i < std::abs(exponent); ++i) {

result \*= base;

}

if (exponent < 0) {

result = 1.0 / result;

}

return result;

}

int main() {

int intFactorialInput = 5;

double doubleFactorialInput = 5.5;

cout << "Factorial of " << intFactorialInput << ": " << factorial(intFactorialInput) << endl;

cout << "Factorial of " << doubleFactorialInput << ": " << factorial(doubleFactorialInput) <<endl;

int intBase = 2;

double doubleBase = 2.5;

int intExponent = 5;

cout << intBase << " raised to the power of " << intExponent << ": " << power(intBase, intExponent) <<endl;

cout << doubleBase << " raised to the power of " << intExponent << ": " << power(doubleBase, intExponent) <<endl;

return 0;

}

**Output:**

