**Write a code in cpp swapping two  value**

**// Pass by value**

#include <iostream>

void swapByValue(int a, int b) {

int temp = a;

a = b;

b = temp;

}

int main() {

int x = 5;

int y = 10;

std::cout << "Before swap by value: x = " << x << ", y = " << y << std::endl;

swapByValue(x, y);

std::cout << "After swap by value: x = " << x << ", y = " << y << std::endl;

return 0;

}

OUTPUT:

Before swap by value: x = 5, y = 10

After swap by value: x = 5, y = 10

**//Pass by reference**

#include <iostream>

void swap(int &a, int &b) {

int temp = a;

a = b;

b = temp;

}

int main() {

int x = 5;

int y = 10;

std::cout << "Before swap : x = " << x << ", y = " << y << std::endl;

swapByReference(x, y);

std::cout << "After swap: x = " << x << ", y = " << y << std::endl;

return 0;

}

OUTPUT:

Before swap by reference: x = 5, y = 10

After swap by reference: x = 10, y = 5

**Ambiguity in function overloading**

#include <iostream>

using namespace std;

void test(int a) {

cout << "x is " << a << endl;

}

void test(int a, int b = 7) {

cout << "x is " << a << endl << "y is " << b << endl;

}

int main() {

int x = 5, y = 7;

test(x); // Calls the single parameter function

test(x, y); // Calls the two-parameter function

return 0;

}

Problem Statement: Distance Calculation Using Operator Overloading You are required to implement a program that calculates distances using operator overloading in C++. The program should be able to perform the following operations on distances: Addition of Distances: Implement an addition operator (+) that adds two distances together. The distance should be represented in feet and inches. Subtraction of Distances: Implement a subtraction operator (-) that subtracts one distance from another. Ensure that the subtraction operation handles cases where the result may involve negative values or borrowing (like in subtraction of inches). Comparison of Distances: Implement comparison operators (==, !=, <, >, <=, >=) to compare distances based on their total length (combined feet and inches). Use these operators to determine which distance is greater, less than, or equal to another. Requirements: Distance Class: Implement a Distance class with appropriate member variables (feet and inches). Constructors: Implement constructors to initialize distances. Member Functions: Implement member functions for display and any other necessary operations. Operator Overloading: Overload the necessary operators (+, -, ==, !=, <, >, <=, >=) inside the Distance class to perform the specified operations. Testing: Create a main() function to test the implemented Distance class and its operator overloading functionality. Test various scenarios including addition, subtraction, and comparison of distances

#include <iostream>

using namespace std;

class Distance {

private:

int feet;

int inches;

public:

// Constructor

Distance(int ft = 0, int in = 0) : feet(ft), inches(in) {

normalize(); // Normalize the distance to ensure inches < 12

}

// Getter functions

int getFeet() const {

return feet;

}

int getInches() const {

return inches;

}

// Overloading addition operator (+)

Distance operator+(const Distance& other) const {

int totalFeet = feet + other.feet;

int totalInches = inches + other.inches;

return Distance(totalFeet, totalInches);

}

// Overloading subtraction operator (-)

Distance operator-(const Distance& other) const {

int totalFeet = feet - other.feet;

int totalInches = inches - other.inches;

if (totalInches < 0) {

totalFeet--; // Borrow one foot

totalInches += 12; // Convert negative inches to positive

}

return Distance(totalFeet, totalInches);

}

// Comparison operators

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

return (feet == other.feet && inches == other.inches);

}

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

return !(\*this == other);

}

bool operator<(const Distance& other) const {

if (feet < other.feet) {

return true;

} else if (feet == other.feet) {

return inches < other.inches;

} else {

return false;

}

}

bool operator>(const Distance& other) const {

return !(\*this < other || \*this == other);

}

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

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

}

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

return !(\*this < other);

}

// Display function

void display() const {

cout << feet << " feet " << inches << " inches";

}

private:

// Helper function to normalize distance (inches should be < 12)

void normalize() {

if (inches >= 12) {

feet += inches / 12;

inches %= 12;

} else if (inches < 0) {

int borrowFeet = (abs(inches) + 11) / 12;

feet -= borrowFeet;

inches += borrowFeet \* 12;

}

}

};

// Main function to test Distance class

int main() {

Distance d1(5, 8);

Distance d2(3, 10);

// Addition

Distance sum = d1 + d2;

cout << "Sum: ";

sum.display();

cout << endl;

// Subtraction

Distance diff = d1 - d2;

cout << "Difference: ";

diff.display();

cout << endl;

// Comparisons

cout << "Comparison: ";

d1.display();

if (d1 == d2) {

cout << " is equal to ";

} else if (d1 != d2) {

cout << " is not equal to ";

}

d2.display();

cout << endl;

cout << "Comparison: ";

d1.display();

if (d1 < d2) {

cout << " is less than ";

} else if (d1 > d2) {

cout << " is greater than ";

} else {

cout << " is equal to ";

}

d2.display();

cout << endl;

return 0;

}

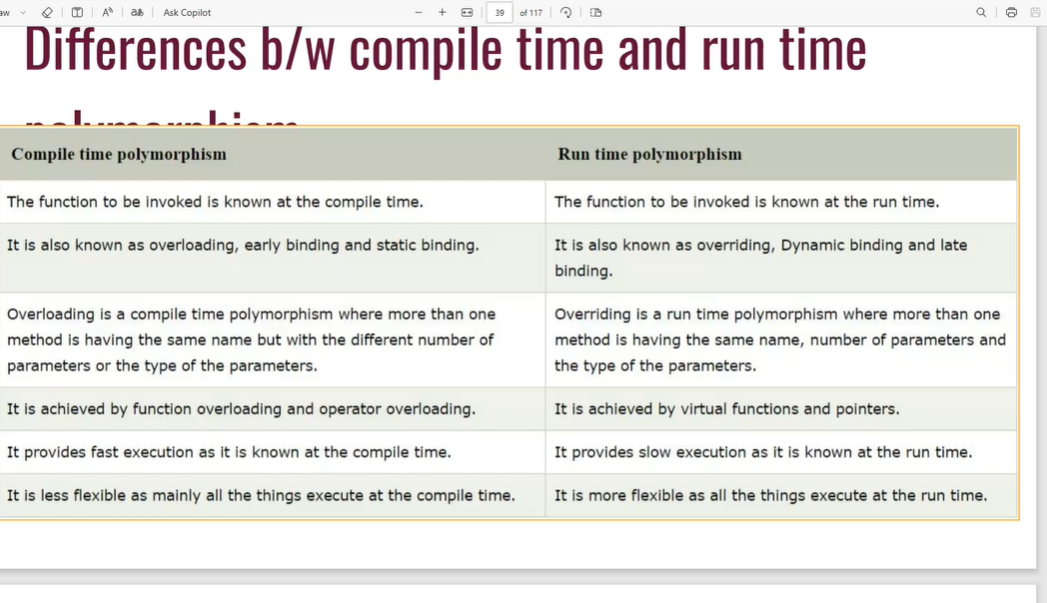
**OUTPUT:**

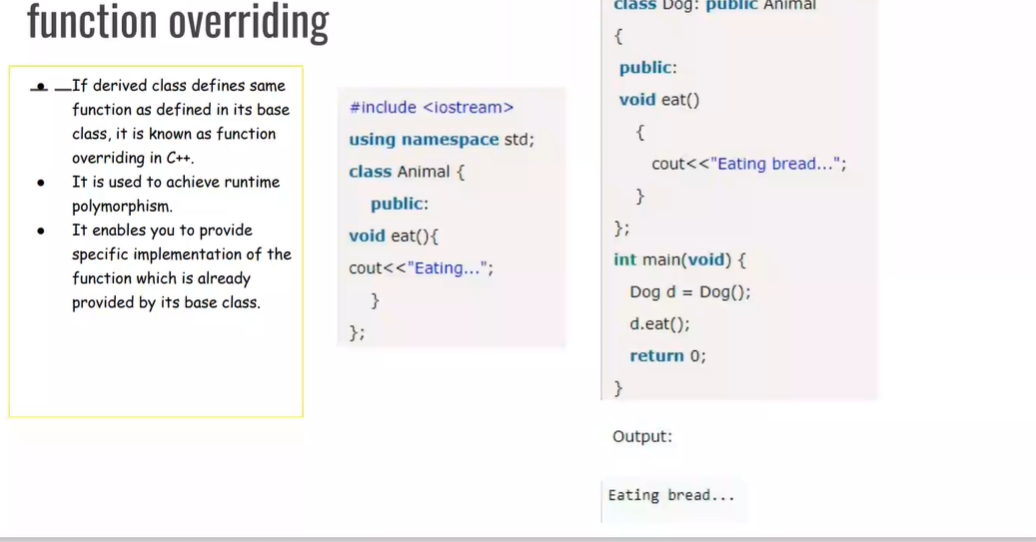
Sum: 9 feet 6 inches

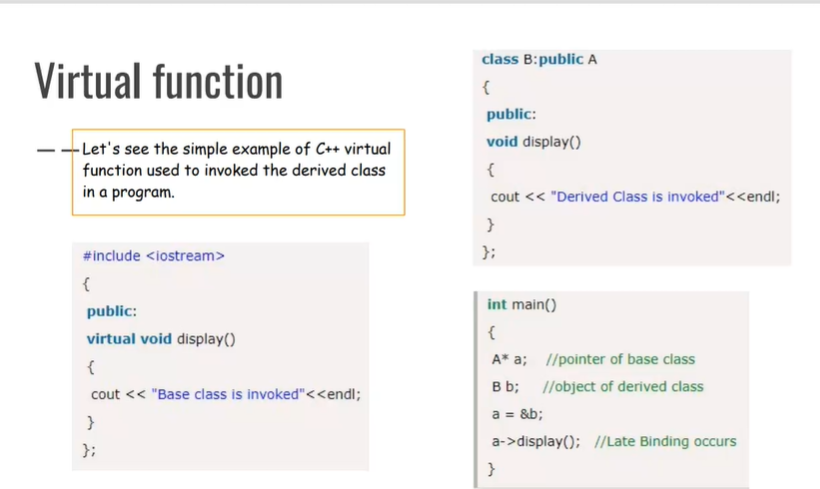
Difference: 1 feet 10 inches

Comparison: 5 feet 8 inches is not equal to 3 feet 10 inches

Comparison: 5 feet 8 inches is greater than 3 feet 10 inches







**Q.Create a base class Shape with a pure virtual function draw() that has no implementation. Derive classes Square, Circle, and Triangle from Shape. Each derived class should override draw() to provide its specific drawing behavior (e.g., printing "\*" for square, "OOO" for circle, etc.). Write a function printShape(Shape shape) that takes a base class pointer and calls draw() on it. Demonstrate polymorphism by creating objects of the derived classes, storing them in a Shape\* array, and calling printShape() on each element.**

#include <iostream>

#include <vector>

using namespace std;

class Shape {

public:

virtual void draw() const = 0;

virtual ~Shape() = default;

};

class Square : public Shape {

public:

void draw() const override {

cout << "\*\*\*\*\n\* \*\n\*\*\*\*" << endl;

}

};

class Circle : public Shape {

public:

void draw() const override {

cout << " OOO \nO O\n OOO " << endl;

}

};

class Triangle : public Shape {

public:

void draw() const override {

cout << " \* \n \* \* \n\*\*\*\*\*" << endl;

}

};

void printShape(const Shape\* shape) {

shape->draw();

}

int main() {

Square square;

Circle circle;

Triangle triangle;

vector<Shape\*> shapes = {&square, &circle, &triangle};

for (const auto& shape : shapes) {

printShape(shape);

cout << endl;

}

return 0;

}

**OUTPUT:**

\*\*\*\*

\* \*

\*\*\*\*

OOO

O O

OOO

\*

\* \*

\*\*\*\*\*

**Design a base class Animal with a pure virtual function makeSound() that returns a string representing the animal's sound. Derive classes like Dog, Cat, and Bird from Animal, each overriding makeSound() with the appropriate sound ("Woof!", "Meow!", "Chirp!"). Create a function playAnimalSound(Animal\* animal) that takes an Animal pointer and calls makeSound(). Populate an Animal\* array with various animal objects and use playAnimalSound() to hear their sounds polymorphically.**

#include <iostream>

#include <string>

using namespace std;

// Base class Animal

class Animal {

public:

virtual string makeSound() = 0; // Pure virtual function

};

// Derived class Dog

class Dog : public Animal {

public:

string makeSound() override {

return "Woof!";

}

};

// Derived class Cat

class Cat : public Animal {

public:

string makeSound() override {

return "Meow!";

}

};

// Derived class Bird

class Bird : public Animal {

public:

string makeSound() override {

return "Chirp!";

}

};

// Function to play animal sound

void playAnimalSound(Animal\* animal) {

cout << animal->makeSound() << endl;

}

int main() {

// Create objects of derived classes

Dog dog;

Cat cat;

Bird bird;

// Array of Animal pointers

Animal\* animals[] = { &dog, &cat, &bird };

// Call playAnimalSound() on each element

for (int i = 0; i < 3; ++i) {

playAnimalSound(animals[i]);

}

return 0;

}

**OUTPUT:**

Woof!

Meow!

Chirp!