**1. Difference between == and .equals() in C++**

In C++, == is an operator used to compare the values of two variables or objects to determine if they are equal. The behavior of == can be overloaded for user-defined types (like classes) to customize what "equality" means for those types.

.equals() is not a standard method in C++. This concept is often seen in languages like Java, where .equals() is a method that compares the contents of objects for equality. In C++, the equivalent would be a member function or a free function designed to compare two objects of a class.

**Example:**

class MyClass {

public:

int value;

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

return value == other.value;

}

};

MyClass a{10}, b{10};

if (a == b) {

// This will be true because the operator== is overloaded

}

**2. Memory Management in C++**

C++ handles memory management using manual allocation and deallocation of memory. The primary tools for this are pointers, and the new and delete operators.

Pointers: Variables that hold the memory address of another variable. They allow direct manipulation of memory and are essential for dynamic memory allocation.

new Operator: Allocates memory on the heap for a variable or an array and returns a pointer to the allocated memory.

delete Operator: Deallocates memory that was previously allocated with new, freeing the memory so it can be reused.

**Example:**

int\* ptr = new int; // Allocate an integer on the heap

\*ptr = 5; // Use the allocated memory

delete ptr; // Free the allocated memory

int\* arr = new int[10]; // Allocate an array of 10 integers

delete[] arr; // Free the allocated array memory

**3. Purpose of the const Keyword in C++**

The const keyword specifies that a variable's value cannot be modified after initialization. It can be used in various contexts, such as with variables, pointers, member functions, and function parameters.

**Example:**

const int x = 10; // x is a constant and cannot be modified

void printValue(const int& value) {

// value is a reference to a constant int and cannot be modified within this function

std::cout << value << std::endl;

}

class MyClass {

public:

void myFunction() const {

// This member function cannot modify any member variables of the class

}

};

**4. Function Overloading vs. Function Overriding in C++**

**Function Overloading:** Allows multiple functions with the same name but different parameters (number or types) within the same scope. The compiler determines which function to call based on the arguments.

**Example:**

void print(int i) {

std::cout << "Integer: " << i << std::endl;

}

void print(double d) {

std::cout << "Double: " << d << std::endl;

}

void print(const std::string& s) {

std::cout << "String: " << s << std::endl;

}

**Function Overriding:** Involves redefining a base class's virtual function in a derived class to provide a specific implementation.

**Example**:

class Base {

public:

virtual void show() {

std::cout << "Base show" << std::endl;

}

};

class Derived : public Base {

public:

void show() override {

std::cout << "Derived show" << std::endl;

}

};

Base\* b = new Derived();

b->show(); // Outputs "Derived show"

**5. Significance of the virtual Keyword in C++**

The virtual keyword is used to declare a function in a base class as virtual, allowing it to be overridden in any derived class. This supports polymorphism, enabling the correct function to be called based on the actual object type at runtime, not the pointer type.

**Inheritance**: It allows derived classes to inherit and possibly override base class methods.

**Polymorphism**: Enables dynamic binding, where the method called is determined at runtime based on the object's actual type, not its pointer/reference type.

**Example:**

cpp

Copy code

class Animal {

public:

virtual void makeSound() {

std::cout << "Some generic animal sound" << std::endl;

}

};

class Dog : public Animal {

public:

void makeSound() override {

std::cout << "Woof!" << std::endl;

}

};

Animal\* myAnimal = new Dog();

myAnimal->makeSound(); // Outputs "Woof!" because of the virtual function mechanism

Using virtual in the base class ensures that the derived class's method is invoked, providing the correct behavior in a polymorphic context.