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

In C++, the == operator is used to compare primitive data types and objects. However, .equals() is not a standard method in C++ (unlike in Java, where equals() is a method for comparing objects). In C++, if you want to compare objects, you need to overload the == operator for your class.

Example for ==:

```
int a = 5;

int b = 5;

if (a == b) {

  // This is true because both a and b have the same value.

}
```

Example of overloading == for a custom class:

```
class MyClass {

public:

  int value;

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

    return this->value == other.value;

  }

};


MyClass obj1{5};

MyClass obj2{5};

if (obj1 == obj2) {

  // This will be true because we've overloaded the == operator to compare the value fields.

}
```

2. Memory Management in C++

C++ handles memory management through manual allocation and deallocation using pointers, new, and delete operators.

Pointers: Variables that hold the address of another variable. They are essential for dynamic memory management.

new operator: Allocates memory on the heap for a variable and returns a pointer to it.

delete operator: Frees memory allocated on the heap, preventing memory leaks.

Example:

```
int* ptr = new int; // Allocate memory for an int on the heap

*ptr = 10; // Assign value to the allocated memory

delete ptr; // Free the allocated memory


int* arr = new int[10]; // Allocate memory for an array of 10 ints on the heap

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

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

Function Overloading:

Allows multiple functions with the same name but different parameter lists within the same scope.

The compiler differentiates these functions by their signature (number and type of parameters).

Example:

```
void print(int i) {
   std::cout << i << std::endl;
}

void print(double d) {
   std::cout << d << std::endl;
}

void print(std::string s) {
   std::cout << s << std::endl;
}
```

Function Overriding:

Allows a subclass to provide a specific implementation of a method that is already defined in its superclass.

The method in the subclass has the same signature as the method in the superclass.

Requires the virtual keyword in the base class method.

Example:

```
class Base {
public:
   virtual void display() {
      std::cout << "Display from Base" << std::endl;
   }
};

class Derived : public Base {
public:
```

```
  void display() override {
    std::cout << "Display from Derived" << std::endl;
  }
};

Base* b = new Derived();
b->display(); // Outputs "Display from Derived"
```

5. Significance of the virtual Keyword

The virtual keyword is used to indicate that a member function can be overridden in derived classes. It enables polymorphism, allowing the correct method to be called based on the object's type at runtime.

Inheritance and Polymorphism:

Inheritance: Allows a class (derived class) to inherit properties and behaviors (methods) from another class (base class).

Polymorphism: Allows a function to behave differently based on the object that is calling it. Achieved through virtual functions.

Example:

```
class Animal {
public:
  virtual void makeSound() const {
    std::cout << "Some generic animal sound" << std::endl;
  }
};

class Dog : public Animal {
public:
  void makeSound() const override {
    std::cout << "Woof" << std::endl;
  }
};
```

```
class Cat : public Animal {
public:
   void makeSound() const override {
      std::cout << "Meow" << std::endl;
   }
};

void describeSound(const Animal& animal) {
   animal.makeSound(); // Calls the appropriate makeSound function based on the object type
}

Animal a;
Dog d;
Cat c;

describeSound(a); // Outputs "Some generic animal sound"
describeSound(d); // Outputs "Woof"
describeSound(c); // Outputs "Meow"
```