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

In C++, there is no `.equals()` method as there is in languages like Java. Instead, the `==` operator is commonly overloaded to compare objects. Here's a detailed explanation:

- **`==` Operator**: This operator is used to compare the values of two variables. For primitive data types (like int, char, etc.), it checks for value equality. For objects, it can be overloaded to provide custom comparison logic.
- **`.equals()` Method**: While C++ does not natively provide an `.equals()` method, similar functionality can be achieved through member functions. For example, you can define an `equals` method in your class to compare objects.

Example:

```
class MyClass {
public:
    int data;
    bool operator==(const MyClass &other) const {
        return this->data == other.data;
    }

    bool equals(const MyClass &other) const {
        return this->data == other.data;
    }
};

MyClass obj1, obj2;
obj1.data = 5;
obj2.data = 5;

if (obj1 == obj2) {
    // Uses the overloaded == operator
}

if (obj1.equals(obj2)) {
    // Uses the equals method
}
```

2. Memory Management in C++: Pointers, `new`, and `delete` Operators

C++ provides explicit control over memory management through pointers and dynamic memory allocation.

- **Pointers**: Pointers are variables that store memory addresses. They are used to directly access and manipulate memory.

```
int x = 10;
int *ptr = &x; // ptr holds the address of x
```

- **`new` Operator**: This operator allocates memory on the heap and returns a pointer to the beginning of the allocated memory. It is used for dynamic memory allocation.

```
int *p = new int; // dynamically allocate memory for an int
*p = 20;
```

- **`delete` Operator**: This operator deallocates memory that was previously allocated with `new`, preventing memory leaks.

```
delete p; // free the memory allocated for the int
```

For arrays, `new[]` and `delete[]` are used:

```
int *arr = new int[10]; // dynamically allocate memory for an array of 10
ints
delete[] arr; // deallocate the array memory
```

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

The `const` keyword is used to define variables or parameters whose value cannot be changed after initialization. It can be applied to variables, pointers, function parameters, and member functions.

Examples:

- **Const Variable**:

```
const int MAX = 100;
```

- **Const Pointer**:

```
const int *ptr = &MAX; // pointer to a const int
int *const ptr2 = &x; // const pointer to an int
```

- **Const Function Parameter**:

```
void print(const int value) {
    // value cannot be modified inside this function
}
```

- **Const Member Function**:

```
class MyClass {
public:
    void display() const {
        // this function cannot modify any member variables
    }
};
```

4. Function Overloading vs Function Overriding in C++

- **Function Overloading**: This is a feature where multiple functions can have the same name but different parameters (different type or number of parameters). It is resolved at compile time (compile-time polymorphism).

Example:

```
void display(int i) {
    // implementation for int
}

void display(double d) {
    // implementation for double
}
```

- **Function Overriding**: This occurs when a derived class has a definition for one of the member functions of the base class. That base function is said to be overridden. It is resolved at runtime (runtime polymorphism).

Example:

```
class Base {
public:
    virtual void show() {
        // base class implementation
    }
};

class Derived : public Base {
public:
    void show() override {
        // derived class implementation
    }
};
```

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

The `virtual` keyword is used to declare a member function in the base class that can be overridden in a derived class. It enables runtime polymorphism, allowing the program to decide at runtime which function to call based on the type of object being referenced.

Example:

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

```
class Derived : public Base {
public:
    void display() override {
        cout << "Derived display" << endl;
    }
};

Base *b = new Derived();
b->display(); // calls Derived's display() method due to virtual keyword
```

The `virtual` keyword ensures that the correct function is called for an object, regardless of the type of reference (or pointer) used for the function call, enabling dynamic (runtime) polymorphism.