C BASED:

Basics & Syntax

1. What are the differences between #include <filename.h> and #include "filename.h"?
2. What is the difference between printf() and scanf()?
3. What are the different data types in C?
4. What is the difference between = and == in C?
5. How do you declare and initialize a pointer in C?

Control Structures

6. What is the difference between while and do-while loop?
7. Can we use a for loop without initialization, condition, or increment?
8. What is the difference between break and continue?
9. How does switch work in C, and when would you prefer it over if-else?

Basics & Syntax

1. Differences between #include <filename.h> and #include "filename.h"

- #include <filename.h> → Compiler looks for the header file **only in the system directories** (standard library headers like stdio.h, math.h).
- #include "filename.h" → Compiler first looks in the **current directory**, and if not found, then searches system directories. Use < > for standard libraries, and " " for your custom headers.

---

2. Difference between printf() and scanf()

- printf() → Output function, used to display text/data to the console.
- scanf() → Input function, used to take input from the user.

```
#include <stdio.h>
int main() {
    int age;
    printf("Enter your age: ");
    scanf("%d", &age);   // input
    printf("You are %d years old\n", age);  // output
    return 0;
}
```

---

3. Different data types in C

- **Basic types** → int, float, double, char
- **Derived types** → Arrays, Pointers, Structures, Unions
- **Qualifiers** → short, long, signed, unsigned

Example:

```
int a = 10;
float b = 3.14;
```

```
char c = 'A';
double d = 3.14159;
```

---

4. Difference between = and ==

- = → Assignment operator (assigns a value).
- == → Comparison operator (checks equality).

```
int x = 5;       // assignment
if (x == 5) {    // comparison
    printf("x is 5\n");
}
```

---

5. Declaring and initializing a pointer

```
int a = 10;
int *p;         // declaration
p = &a;         // initialization with address of 'a'

printf("Value of a: %d\n", *p);  // dereferencing
```

*p gives value stored at address, p gives address.

Control Structures

6. Difference between while and do-while loop

- while → Condition is checked **before** execution. May not run at all.
- do-while → Condition is checked **after** execution. Runs at least once.

```
int x = 0;
while (x > 0) {  // won't execute
    printf("Hello\n");
}
```

```
do {
    printf("Hello once\n"); // executes at least once
} while (x > 0);
```

---

7. Can we use a for loop without initialization, condition, or increment?

Yes. All three parts are optional. It becomes an infinite loop if left empty.

```
for(;;) {
    printf("Infinite loop\n");
    break; // to exit
}
```

---

8. Difference between break and continue

- break → Exits the loop completely.
- continue → Skips the current iteration and moves to the next one.

```
for (int i=1; i<=5; i++) {
    if (i == 3) continue; // skip when i=3
    if (i == 5) break;    // exit loop when i=5
    printf("%d ", i);
}
// Output: 1 2 4
```

---

9. How does switch work in C, and when to prefer it over if-else?

- switch evaluates a single expression and jumps to the matching case.
- It’s preferred over multiple if-else when you are comparing the same variable with many constant values (better readability).

```
int day = 3;
switch(day) {
    case 1: printf("Monday"); break;
```

```
    case 2: printf("Tuesday"); break;
    case 3: printf("Wednesday"); break;
    default: printf("Invalid day");
}
// Output: Wednesday
```

---

switch is faster and cleaner when there are many options for the same variable. Use if-else when conditions are **ranges** or **complex expressions.**

Python Based:

1.Difference between mutable and immutable objects in Python?

2.What will be the output of the following and why?

```
a = 256
b = 256
print(a is b)

x = 257
y = 257
print(x is y)
```

3. Predict the output:

```
def add(x, y=[]):
    y.append(x)
    return y

print(add(1))
print(add(2))
print(add(3, []))
print(add(4))
```

4.Without using a temporary variable, how do you swap two numbers in Python?

5. What is the difference between these two statements?

```
list1 = [1, 2, 3]
```

```
list2 = list1
list3 = list1[:]
```

6. What will be the output?

```
print(bool("False"))
print(bool(""))
```

7. Why does this code behave differently?

```
print(0.1 + 0.2 == 0.3)
```

8. Predict the output and explain:

```
for i in range(3):
    print(i)
else:
    print("Finished")
```

9. What will be the output?

```
x = [1, 2, 3]
print(id(x))
x += [4, 5]
print(id(x))

y = (1, 2, 3)
print(id(y))
y += (4, 5)
print(id(y))
```

10. Write a one-liner in Python to reverse a string without using loops.

ANSWERS:

1. Difference between mutable and immutable objects in Python?

- **Mutable** → Can be changed after creation. Examples: `list`, `dict`, `set`.
- **Immutable** → Cannot be changed after creation. Examples: `int`, `float`, `str`, `tuple`.

```
lst = [1, 2, 3]
lst[0] = 10   # ✅ Works (mutable)

s = "hello"
# s[0] = "H" ✖ Error (immutable)
s = "Hello"   # Creates a new string
```

---

2. Output & Why?

```
a = 256
b = 256
print(a is b)   # True

x = 257
y = 257
print(x is y)   # False (in most implementations)
```

? Explanation:

- Python **caches small integers** in the range `[-5, 256]` for efficiency.
- So `a` and `b` point to the **same memory**.
- For numbers outside this range (like `257`), Python creates **new objects**, so `x is y` → `False`.

---

3. Predict the output

```
def add(x, y=[]):
    y.append(x)
    return y

print(add(1))       # [1]
```

```
print(add(2))       # [1, 2]   (same default list reused!)
print(add(3, []))   # [3]      (new list passed explicitly)
print(add(4))       # [1, 2, 4]
```

? Explanation: Default mutable arguments (`y=[]`) are **shared** across function calls unless you explicitly pass a new list.

---

4. Swap without temp variable

```
a, b = 10, 20
a, b = b, a
print(a, b)   # 20 10
```

? Python allows **tuple unpacking** for swapping.

---

5. Difference between the statements

```
list1 = [1, 2, 3]
list2 = list1      # Reference copy → both point to same object
list3 = list1[:]   # Shallow copy → new object with same elements
```

- `list2` and `list1` are the **same object** → modifying one affects the other.
- `list3` is a **different object** (copy).

---

6. Output

```
print(bool("False"))   # True (non-empty string is True)
print(bool(""))        # False (empty string is False)
```

---

7. Why different behavior?

```
print(0.1 + 0.2 == 0.3)   # False
```

? Due to **floating-point precision error**.
Internally `0.1 + 0.2 = 0.30000000000000004`, which is not exactly equal to `0.3`.

---

8. Predict the output

```
for i in range(3):
    print(i)
else:
    print("Finished")
```

Output:

```
0
1
2
Finished
```

? Explanation: The `else` in a `for` loop executes if the loop **completes normally** (not broken by `break`).

---

9. Output

```
x = [1, 2, 3]
print(id(x))          # same id
x += [4, 5]
print(id(x))          # same id (list modified in-place)

y = (1, 2, 3)
print(id(y))          # some id
y += (4, 5)
print(id(y))          # different id (new tuple created)
```

? Explanation:

- Lists are **mutable**, so `+=` modifies in-place → `id` stays same.
- Tuples are **immutable**, so `+=` creates a **new tuple object** → `id` changes.

---

10. One-liner to reverse a string

```
s = "Python"
print(s[::-1])   # nohtyP
```

? Uses **slicing with negative step**.

JAVA BASED:

1. Difference between `==` and `.equals()` in Java?
2. Why is Java called platform-independent?
3. Explain the difference between `final`, `finally`, and `finalize()`.
4. What are access modifiers in Java?
5. Difference between `Array` and `ArrayList`?
6. What is the difference between an `abstract` class and an `interface`?
7. Explain method overloading vs. method overriding.
8. What is the difference between `==` operator and `compareTo()` in Strings?
9. Explain garbage collection in Java.
10. `What is the difference between JDK, JRE, and JVM?`

Answers

1. Difference between JDK, JRE, and JVM

- **JVM**: Runs Java bytecode.
- **JRE**: JVM + libraries to run Java programs.
- **JDK**: JRE + compiler and development tools.

2. Difference between `==` and `.equals()`

- `==`: Compares memory references.
- `.equals()`: Compares actual content (if overridden, e.g., in `String`).

3. Why is Java platform-independent?
Java compiles code into **bytecode** which runs on any OS that has a **JVM** → "Write Once, Run Anywhere."

4. Difference between `final`, `finally`, and `finalize()`

- `final`: Keyword (constants, prevent inheritance/overriding).
- `finally`: Block that always executes after `try-catch`.
- `finalize()`: Method run by Garbage Collector before object destruction.

5. Access Modifiers in Java

- `public`: Accessible everywhere.
- `protected`: Accessible in same package + subclasses.
- `default` (no keyword): Accessible only in same package.
- `private`: Accessible only within class.

6. Difference between `Array` and `ArrayList`

- Array: Fixed size, can hold primitives + objects.
- ArrayList: Dynamic size, holds only objects, has utility methods.

7. Difference between Abstract Class and Interface

- Abstract class: Can have both abstract + concrete methods, can hold fields.
- Interface: Only method definitions (before Java 8), multiple inheritance allowed.

8. Method Overloading vs. Method Overriding

- Overloading: Same name, different parameters (compile-time polymorphism).
- Overriding: Child class redefines parent method (runtime polymorphism).

9. Difference between `==` and `compareTo()` in Strings

- `==`: Checks reference equality.
- `compareTo()`: Lexicographic comparison (`0` if equal, <0 or >0 otherwise).

10. Garbage Collection in Java

- Automatic memory management.
- Removes unreferenced objects.
- `System.gc()` requests GC, but JVM decides when to run it.

C++ BASED:

1.What will be the output?

```
#include <iostream>
using namespace std;
int main() {
    int a = 10;
    cout << a++ << " " << ++a;
    return 0;
}
```

2.Predict the output:

```
#include <iostream>
using namespace std;
int main() {
    const int x = 5;
    const int *p = &x;
    int y = 10;
    p = &y;
    cout << *p;
    return 0;
}
```

3.What will this print?

```
#include <iostream>
using namespace std;
void func(int x, int y=5) {
    cout << x + y << endl;
}
```

```
int main() {
    func(10);
    func(10, 20);
}
```

4.Identify error (if any) and explain:

```
#include <iostream>
using namespace std;
int main() {
    int arr[5] = {1,2,3,4,5};
    cout << *(arr+2);
    return 0;
}
```

5.Output of this program?

```
#include <iostream>
using namespace std;
int main() {
    string s1 = "Hello";
    string s2 = s1;
    s2[0] = 'Y';
    cout << s1 << " " << s2;
}
```

6.What happens here?

```
#include <iostream>
using namespace std;
class Test {
public:
```

```
    Test() { cout << "Constructor "; }
    ~Test() { cout << "Destructor "; }
};
int main() {
    Test t1;
    {
        Test t2;
    }
}
```

7.Predict the output:

```
#include <iostream>
using namespace std;
class A {
public:
    virtual void show() { cout << "A "; }
};
class B : public A {
public:
    void show() override { cout << "B "; }
};
int main() {
    A *ptr = new B();
    ptr->show();
    delete ptr;
}
```

8.What will this print?

```
#include <iostream>
```

```
using namespace std;
int main() {
    int x = 5;
    cout << (x << 1) << " " << (x >> 1);
}
```

9.Find the mistake:

```
#include <iostream>
using namespace std;
int main() {
    int *p = new int(5);
    cout << *p;
    delete p;
    cout << *p;
}
```

10. Reverse a string using STL in one line.

Answers

Q1

```
int a = 10;
cout << a++ << " " << ++a;
```

`a++` prints 10, then `a` becomes 11.

`++a` makes it 12 before printing.

Output:

```
10 12
```

Q2

```
const int x = 5;
const int *p = &x;
int y = 10;
p = &y;
cout << *p;
```

- `p` is a pointer to **const int**, meaning you can change where it points, but not modify the value via `p`.
- Initially `p` points to `x`, then changed to `y`.

Output:

```
10
```

Q3

```
void func(int x, int y=5) {
    cout << x + y << endl;
```

```
}
func(10);       // uses default y=5 → 15
func(10, 20);   // uses y=20 → 30
```

Output:

```
15
30
```

Q4

```
int arr[5] = {1,2,3,4,5};
cout << *(arr+2);
```

? `arr` is base address, so `arr+2` points to 3rd element.

? `*(arr+2)` = `arr[2]` = 3.

Output:

```
3
```

Q5

```
string s1 = "Hello";
string s2 = s1;
s2[0] = 'Y';
cout << s1 << " " << s2;
```

In C++, `string` is deep copied, so modifying `s2` won't change `s1`.

Output:

```
Hello Yello
```

Q6

```
class Test {
public:
    Test() { cout << "Constructor "; }
    ~Test() { cout << "Destructor "; }
};
Test t1;
{
    Test t2;
}
```

- First prints `Constructor` for `t1`.
- Then inside block: `t2` is constructed → prints `Constructor`.
- Block ends → `t2` destroyed → `Destructor`.
- At end of `main`, `t1` destroyed → `Destructor`.

Output:

```
Constructor Constructor Destructor Destructor
```

Q7

```
class A { public: virtual void show() { cout << "A
"; } };
class B : public A { public: void show() override {
cout << "B "; } };
A *ptr = new B();
ptr->show();
```

? `ptr` is of type `A*` but points to `B`.

? Because `show()` is `virtual`, runtime polymorphism calls `B`'s version.

Output:

```
B
```

Q8

```
int x = 5;
cout << (x << 1) << " " << (x >> 1);
```

`5` in binary = `101`.

- `x << 1` → `1010` (binary 10).
- `x >> 1` → `10` (binary 2).

Output:

```
10 2
```

Q9

```
int *p = new int(5);
cout << *p;
delete p;
cout << *p;
```

? First `*p = 5` is valid.

? After `delete`, memory is freed. Accessing `*p` is **undefined behavior** (might print garbage or crash).

Expected safe output:

```
5 <garbage or crash>
```

Q10 Reverse a string in one line using STL:

```
#include <algorithm>
#include <iostream>
using namespace std;
int main() {
    string s = "Hello";
    reverse(s.begin(), s.end());
    cout << s;
}
```

Output:

```
olleH
```