```
int main() {
    int* p;
    p = new int;

    *p = 100;

    cout << "Value at *p = " << *p << endl;

    delete p;

    // Set pointer to nullptr (good practice)
    p = nullptr;

    return 0;
}
```

```
int main() {
    int* x = new int;
    int* y = new int;

    *x = 10;
    *y = *x;
    *x = 20;

    cout << "*x = " << *x << endl;
    cout << "*y = " << *y << endl;

    delete x;
    delete y;

    return 0;
}
```

Initial State (After `int* x = new int;` and `int* y = new int;`)

Memory Area	Variable	Address	Value	Notes
Stack	`x` (pointer)	`1000`	`2000`	Stores the **address** of the integer it points to (on the Heap).
	`y` (pointer)	`1004`	`3000`	Stores the **address** of the integer it points to (on the Heap).
Heap	Anonymous int	`2000`	Garbage	Dynamically allocated memory for the integer pointed to by `x`.
	Anonymous int	`3000`	Garbage	Dynamically allocated memory for the integer pointed to by `y`.
Code	`main()` ...	`...`	`...`	The executable program instructions.

After `*x = 10;`

Memory Area	Variable	Address	Value	Notes
Stack	`x` (pointer)	`1000`	`2000`	
	`y` (pointer)	`1004`	`3000`	
Heap	Anonymous int	`2000`	**10**	The value at the memory address **pointed to by** `x` is set to 10.
	Anonymous int	`3000`	Garbage	

After `*y = *x;`

Memory Area	Variable	Address	Value	Notes
Stack	`x` (pointer)	`1000`	`2000`	
	`y` (pointer)	`1004`	`3000`	
Heap	Anonymous int	`2000`	**10**	
	Anonymous int	`3000`	**10**	The **value** pointed to by `x` (which is 10) is copied to the **value** pointed to by `y`. **The pointers still point to separate memory locations.**

After `*x = 20;`

Memory Area	Variable	Address	Value	Notes
Stack	`x` (pointer)	`1000`	`2000`	
	`y` (pointer)	`1004`	`3000`	
Heap	Anonymous int	`2000`	**20**	The value at the memory address **pointed to by** `x` is changed to 20.
	Anonymous int	`3000`	**10**	The value pointed to by `y` remains unchanged.

```
int main() {
    int* p = new int;
    *p = 50;

    int* q = p;
    delete p;

    if (q != nullptr) {
        cout << "*q = " << *q << endl;
    } else {
        cout << "q is null" << endl;
    }

    return 0;
}
```

Explanation:

- p = new int → dynamic memory allocation
- *p = 50 → value stored at that location
- q = p → both p and q point to the same memory
- delete p → memory is freed, but q is still pointing to it (dangling pointer)
- q != nullptr → true, but it points to **freed memory**
- *q → dereferencing a **dangling pointer** → ❗ **Undefined Behavior**

It **might** print 50, it **might crash**, or give garbage — but it's not safe or predictable.

```
int main() {
    int* p = new int;
    *p = 50;

    int* q = p;
    delete p;

    if (q != nullptr) {
        cout << "*q = " << *q << endl;
    } else {
        cout << "q is null" << endl;
    }

    return 0;
}
```

What will be printed?

A) Nothing
B) Memory address
C) 25
D) Compilation error

```
int* p = new int;
*p = 25;
cout << *p << endl;
delete p;
```

What will be printed?

A) 10

B) 20

C) Garbage

D) Compilation error

```
int* a = new int;
*a = 10;

int* b = a;
*a = 20;

cout << *b << endl;
delete a;
```

What will be printed?

A) 100

B) 0

C) Compilation error

D) Undefined behavior

```
int* x = new int;
*x = 100;
delete x;

cout << *x << endl;
```

What happens if this function is called many times?

```
void func() {
    int* temp = new int(42);
    cout << *temp << endl;
}
```

A) Nothing

B) Values won't print

C) Memory leak

D) Crashes instantly

What will be printed?

```
int* p = nullptr;

if (p)
    cout << "Pointer is valid" << endl;
else
    cout << "Pointer is null" << endl;
```

A) Pointer is valid

B) Pointer is null

C) Compilation error

D) Runtime error

Classes & Object

```
class Rectangle {
public:
    int length;
    int width;

    int area() {
        return length * width;
    }
};

int main() {
    Rectangle r1;
    r1.length = 4;
    r1.width = 5;

    Rectangle r2 = r1;
    r2.width = 10;

    cout << r1.area() << endl;
    cout << r2.area() << endl;

    return 0;
}
```

```
class Counter {
public:
    static int count;
    int id;
    void increment() {
        count = count + 1;
        id = count;
    }
    void print() {
        cout << "id: " << id << " count: " << count
<< endl;
    }
};
int Counter::count = 0;
int main() {
    Counter c1;
    Counter c2;

    c1.increment();
    c2.increment();

    c1.print();
    c2.print();

    return 0;
}
```

```
class Box {
public:
    int* length;
    void setLength(int l) {
        length = new int(l);
    }
    void copy(Box& other) {
        length = new int(*other.length);
    }
    void print() {
        cout << *length << endl;
    }
    void cleanup() {
        delete length;
    }
};
```

```
int main() {
    Box b1;
    b1.setLength(5);

    Box b2;
    b2.copy(b1);
    *b2.length = 10;

    b1.print();
    b2.print();

    b1.cleanup();
    b2.cleanup();

    return 0;
}
```

What is the output of this code?

A) Hello
B) Jello
C) Compilation error
D) Runtime error

```
#include <iostream>
using namespace std;

int main() {
    char str[] = "Hello";
    char* p = str;
    p[0] = 'J';
    cout << str << endl;
    return 0;
}
```

Which of the following correctly declares an array of 5 integers and initializes all to 0?

A) int arr[5] = {0};
B) int arr[5] = {};
C) int arr[5] = {0,0,0,0,0};
D) All of the above

What happens when you do this in C++?

A) Prints 10
B) Compilation error
C) Runtime error (dangling pointer)
D) Undefined behavior but prints 0

```
class A {
public:
    int* ptr;
};

int main() {
    A a1;
    a1.ptr = new int(10);

    A a2 = a1;
    delete a1.ptr;

    cout << *a2.ptr << endl;
    return 0;
}
```

Given this class, what is the output?

A) 0
B) 1
C) 2
D) 3

```
#include <iostream>
using namespace std;

class Counter {
public:
    static int count;
    Counter() {
        count++;
    }
};

int Counter::count = 0;

int main() {
    Counter c1, c2, c3;
    cout << Counter::count << endl;
    return 0;
}
```