Lecture 3

Dr. Umair Rehman

Agenda

- Basic concepts (int, &, *, nullptr)
- Pointers to arrays and functions
- Pointers with objects and encapsulation
- Pointers in OOP (polymorphism, base class pointers)

What is a Pointer

- A pointer is a variable that stores the memory address of another variable.

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

```

- *p → value at address (dereference)
- &x → address of x

Declaring and Using Pointers

Syntax	Meaning
int* p;	p is a pointer to int
*p = 5;	assign 5 to the value pointed to by p
cout << *p;	print the value at the memory location

Null Pointers and Safety

- Start with “no address” (nullptr).
  - If you write a random, un-initialized address on the envelope, the mail carrier (your program) could end up anywhere—even a place that crashes the program.
  - Writing int* p = nullptr; clearly says, “I don’t have an address yet.”
- Look before you knock.
  - Before you walk up and open the door (*p), first ask: “Do I actually have an address?”if (p != nullptr) { … } is that quick check.
  - If there’s no address, you safely skip instead of crashing.
- So: set pointers to nullptr, and always check they’re not nullptr before using them.

Null Pointers and Safety

```
int* p = nullptr;        // Step 1: say "I don’t have an address yet"

if (p != nullptr) {      // Step 2: before using it, check if we actually have an address
    *p = 5;              // Step 3: only then try to put a value (5) inside that address
}
```

Pointers with Arrays

```
int arr[3] = {1, 2, 3};
int* p = arr;           // arr decays to pointer to first element
cout << *(p + 1);       // prints 2

```

```
arr[0] = 1    → stored at address 1000
arr[1] = 2    → stored at address 1004
arr[2] = 3    → stored at address 1008

```

Pointers with Functions (Pass by Address)

```cpp
void update(int* p) {
    *p = 99;
}

```

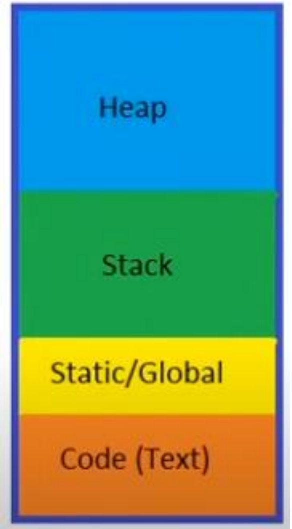
```cpp
int a = 5;
update(&a);  // modifies 'a' directly

```

Dynamic Memory (new and delete)

```cpp
int* p = new int;      // heap allocation
*p = 42;
delete p;              // cleanup

int* arr = new int[5];
delete[] arr;
```

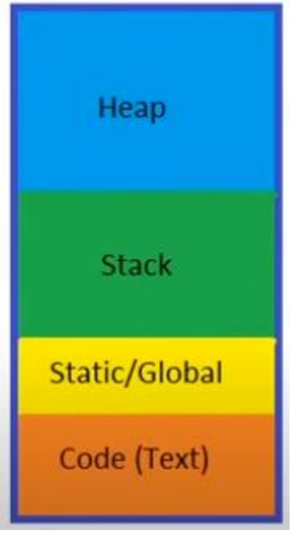
Dynamic Memory (new and delete)

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

//  ◆  Global variable (lives in Global/Static area for entire program lifetime)
int globalVar = 100;

//  ◆  Static global variable (also in Global/Static area)
static int staticGlobal = 200;
```



Dynamic Memory (new and delete)

```cpp
// 🔹 Function itself (its instructions) live in the Text (code) segment
void demo() {
    // 🔹 Local variable (goes on the Stack, destroyed when function ends)
    int localVar = 300;

    // 🔹 Static local variable (lives in Static area, keeps value between calls)
    static int staticLocal = 400;

    // 🔹 Dynamically allocated (lives on the Heap until manually freed)
    int* heapVar = new int(500);

    // Print memory segment examples
    cout << "Local (stack): " << localVar << endl;
    cout << "Static local (static area): " << staticLocal << endl;
    cout << "Heap: " << *heapVar << endl;

    delete heapVar; // cleanup heap memory
}

int main() {
    cout << "Global (static area): " << globalVar << endl;
    cout << "Static global (static area): " << staticGlobal << endl;

    demo(); // call the function to see stack + heap + static local in action
    return 0;
}

```



Example 1

```cpp
#include <iostream>
#include <string>

// =======================
// Mouse class
// =======================
class Mouse {
private:
    std::string brand;

public:
    Mouse(const std::string& b) : brand(b) {}

    void click() {
        std::cout << "[" << brand << " Mouse] Click!\n";
    }

    void move(int dx, int dy) {
        std::cout << "[" << brand << " Mouse] Moved by (" << dx << ", " << dy << ")\n";
    }

    std::string getBrand() const {
        return brand;
    }
};
```

Example 1

```cpp
// =======================
// Computer class
// =======================
class Computer {
private:
    Mouse* connectedMouse;  // raw pointer (not a smart pointer)

public:
    Computer() {
        connectedMouse = nullptr;  // no mouse plugged in yet
    }

    // Plug in a mouse
    void setMouse(Mouse* m) {
        connectedMouse = m;
        std::cout << "Mouse plugged in: " << m->getBrand() << "\n";
    }

```

Example 1

```

    // Use the mouse (if connected)
    void useMouse() {
        if (connectedMouse != nullptr) {
            connectedMouse->move(5, -3);  // simulate movement
            connectedMouse->click();     // simulate click
        } else {
            std::cout << "No mouse connected!\n";
        }
    }

    // Disconnect mouse (but DO NOT delete it here — ownership is outside)
    void unplugMouse() {
        connectedMouse = nullptr;
        std::cout << "Mouse unplugged.\n";
    }
};
```

Example 1

```
// ======================
// Main function
// ======================
int main() {
    // Create a computer (on the stack)
    Computer myPC;

    // Create a mouse using new (on the heap)
    Mouse* logitech = new Mouse("Logitech");

    // Plug the mouse into the computer
    myPC.setMouse(logitech);

    // Use the mouse via computer
    myPC.useMouse();

    // Unplug and try to use it again
    myPC.unplugMouse();
    myPC.useMouse();

    // Manually delete mouse – YOU created it with new!
    delete logitech;

    return 0;
}

```

Pointers vs References

Feature	Pointer (*)	Reference (&)
Can be nullptr	Yes	No — must always refer to something
Can be reassigned	Yes (p = &obj2)	No — bound at creation
Requires new?	Optional	No
Syntax	ptr->method()	ref.method()
Safer	(can forget to check null)	(no nulls)

Example II

```cpp
#include <iostream>
#include <string>

// ==============================
// InkCartridge class (Dependency)
// ==============================
class InkCartridge {
private:
    std::string color;

public:
    InkCartridge(const std::string& c) : color(c) {}

    void supplyInk() {
        std::cout << "[InkCartridge] Providing " << color << " ink.\n";
    }
};

```

Example II

```cpp
// ==============================
// Printer class (Depends on InkCartridge)
// ==============================
class Printer {
private:
    InkCartridge* cartridge;  // Dependency injected via pointer

public:
    // Constructor takes a pointer – this is dependency injection
    Printer(InkCartridge* c) {
        cartridge = c;
    }

    void printDocument(const std::string& text) {
        if (cartridge) {
            cartridge->supplyInk();  // Use the injected dependency
            std::cout << "[Printer] Printing: " << text << "\n";
        } else {
            std::cout << "[Printer] ERROR: No ink cartridge installed!\n";
        }
    }
};
```

Example II

```cpp

// ==============================
// Main function
// ==============================
int main() {
    // Create the dependency
    InkCartridge* blackInk = new InkCartridge("Black");

    // Inject it into the printer
    Printer myPrinter(blackInk);

    // Use the printer (which uses the injected dependency)
    myPrinter.printDocument("Hello, world!");

    // Clean up manually (since we used new)
    delete blackInk;

    return 0;
}

```

Example III (Polymorphism with Pointers)

```cpp
#include <iostream>
#include <string>

// ==========================
// Base class: Animal
// ==========================
class Animal {
public:
    // Virtual function → enables polymorphism
    virtual void makeSound() {
        std::cout << "Animal makes a sound.\n";
    }

    // Always good to have a virtual destructor in base class
    virtual ~Animal() {}
};

// ==========================
// Derived class: Dog
// ==========================
class Dog : public Animal {
public:
    void makeSound() override {
        std::cout << "Dog says: Woof!\n";
    }
};

```

Example III (Polymorphism with Pointers)

```cpp
// ==========================
// Derived class: Cat
// ==========================
class Cat : public Animal {
public:
    void makeSound() override {
        std::cout << "Cat says: Meow!\n";
    }
};

// ==========================
// Main function
// ==========================
int main() {
    // Pointers to base class
    Animal* a1 = new Dog();  // actually a Dog
    Animal* a2 = new Cat();  // actually a Cat

    // Calls the correct method at runtime!
    a1->makeSound();  // Dog's version
    a2->makeSound();  // Cat's version

    // Clean up
    delete a1;
    delete a2;

    return 0;
}
```

Example IV (Incorrect Polymorphism - Slicing)

```cpp
#include <iostream>

class Animal {
public:
    virtual void makeSound() {
        std::cout << "Animal sound!\n";
    }
};

class Dog : public Animal {
public:
    void makeSound() override {
        std::cout << "Dog says Woof!\n";
    }
};

int main() {
    Dog d;
    Animal a = d;        // ❗ slicing happens here
    a.makeSound();       // ❌ calls Animal::makeSound() — NOT Dog
}
```

Example V

```
class Animal {
public:
    virtual void speak() { std::cout << "Animal sound\n"; }
    virtual ~Animal() {}
};

class Cat : public Animal {
public:
    void speak() override { std::cout << "Meow\n"; }
};

Animal* a = new Cat();  // pointer to base, object is derived
a->speak();             // Meow
delete a;

```

Example VI (Inheritance)

```
#include <iostream>

class Animal {
public:
    void eat() {
        std::cout << "Animal eats\n";
    }
};

class Dog : public Animal {
public:
    void bark() {
        std::cout << "Dog barks\n";
    }
};
```

Example VI (Inheritance)

```
int main() {
    Dog d;
    Animal* a = &d;     // base class pointer → derived class object

    a->eat();           // ✅ works (inherited method)
    // a->bark();       // ❌ error: Animal* doesn't know about bark()

    return 0;
}

```

Example VII

```cpp
#include <iostream>
#include <string>

// ============================
// Inner class: Details (private info)
// ============================
class Details {
private:
    std::string bio;
    std::string photo;

public:
    Details() {
        std::cout << "[Loading profile details...]\n";
        bio = "Loves C++ and coffee.";
        photo = "profile.jpg";
    }

    void show() {
        std::cout << "Bio: " << bio << "\n";
        std::cout << "Photo: " << photo << "\n";
    }
};
```

Example VII

```
// ============================
// Profile class
// ============================
class Profile {
private:
    std::string name;
    Details* detailsPtr;  // pointer to details (lazy initialized)

public:
    // Constructor: only sets name
    Profile(const std::string& n) {
        name = n;
        detailsPtr = nullptr;  // details not loaded yet
    }

    // Show just the name
    void showName() {
        std::cout << "Name: " << name << "\n";
    }
```

Example VII

```

    // Show full profile (loads details only if needed)
    void showDetails() {
        if (detailsPtr == nullptr) {
            detailsPtr = new Details();  // Lazy load
        }
        detailsPtr->show();  // call method through pointer
    }

    // Destructor: clean up if details were loaded
    ~Profile() {
        delete detailsPtr;
    }
};
```

Example VII

```cpp
// ============================
// Main
// ============================
int main() {
    Profile p("Umair");

    std::cout << "\nShowing name only:\n";
    p.showName();

    std::cout << "\nNow showing full details:\n";
    p.showDetails();

    std::cout << "\nShowing details again (should not reload):\n";
    p.showDetails();

    // Destructor will clean up
    return 0;
}

```

Conclusion

- Basic concepts (int, &, *, nullptr)
- Pointers to arrays and functions
- Pointers with objects and encapsulation
- Pointers in OOP (polymorphism, base class pointers)