Review on
Reference & Pointers

Reference

- A **reference** is an *alias* (a nickname) for another variable.

A reference is not a new variable — it is *another name* for an existing variable.

Example:

```
int x = 10;
int& ref = x;   // ref is a reference to x
```

Now:

- `ref` **is** `x`
- Changing one changes the other:

```
ref = 20;   // x becomes 20
x = 30;     // ref becomes 30
```

They refer to the **same memory location**.

Reference (key facts)

✓ 1. Must be initialized when created

You *must* bind a reference to something immediately:

```
int& r;  // ❌ not allowed
```

✓ 2. Cannot be changed to refer to something else

Once a reference is bound, it stays bound:

```
int a = 5, b = 10;
int& r = a;    // refers to a
r = b;         // assigns value 10 to a — DOES NOT rebind
```

✓ 3. Uses the same memory address as the original

```
int x = 5;
int& r = x;

cout << &x << endl;  // same address
cout << &r << endl;  // same address
```

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

int main(){
    int a = 5, b = 10;
    int& r = a;

    a = 7;
    cout << "Value or a:" << r << endl;

    r = b;
    b = 30;
    a = 8;
    cout << "Value or a:" << a << endl;
    cout << "Value or b:" << b << endl;
    cout << "Value or r:" << r << endl;

    return 0;
}
```

```
Value or a:7
Value or a:8
Value or b:30
Value or r: ?
```

Reference (key facts)

✔ 1. Must be initialized when created

You *must* bind a reference to something immediately:

```
int& r;  // ❌ not allowed
```

✔ 2. Cannot be changed to refer to something else

Once a reference is bound, it stays bound:

```
int a = 5, b = 10;
int& r = a;   // refers to a
r = b;        // assigns value 10 to a — DOES NOT rebind
```

✔ 3. Uses the same memory address as the original

```
int x = 5;
int& r = x;

cout << &x << endl;  // same address
cout << &r << endl;  // same address
```

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

int main(){
    int a = 5, b = 10;
    int& r = a;

    a = 7;
    cout << "Value or a:" << &r << endl;

    r = b;
    b = 30;
    a = 8;
    cout << "Value or a:" << &a << endl;
    cout << "Value or b:" << &b << endl;
    cout << "Value or r:" << &r << endl;

    return 0;
}
```

```
Value or a:0x16fcd29a8
Value or a:0x16fcd29a8
Value or b:0x16fcd29a4
Value or r:0x16fcd29a8
```

Pointers

- A pointer is a variable that stores a memory address.

✓ **The pointer type must match the type of the thing it points to.**

Examples:

Variable type	Pointer type
`int`	`int*`
`double`	`double*`
`char`	`char*`
`MyClass`	`MyClass*`
array of int	pointer to int → `int*`

A pointer stores an address (e.g., 0x7ffde...).

Example:

```
int x = 10;
int* p = &x;     // correct

double y = 3.14;
double* q = &y; // correct
```

❌ **Wrong:**

```
int x = 10;
double* p = &x;  // ERROR: wrong pointer type
```

```
int x = 5;
int* p = &x;

cout << typeid(p).name();     // shows the type of p (likely "Pi")
cout << typeid(*p).name();    // type of the pointed-to value ("i")
```

Note:

- The actual output varies by compiler (e.g., GCC returns "Pi" for `int*`)

Pointers (key facts)

1. A pointer stores a memory address

A pointer is a variable whose value is the **address of another variable**.

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

---

2. You must dereference a pointer to access the value

Using the `*` operator retrieves or modifies the value **stored at the address** inside the pointer.

```
*p = 20;   // modifies x through the pointer
cout << *p;  // prints the value stored at that address
```

---

3. Pointer types must match what they point to

A pointer to `int` must point to an `int`.
A pointer to `double` must point to a `double`, etc.

```
double y = 3.14;
double* dp = &y;   // correct
```

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

int main(){
    int a = 5;
    int* r = &a;
    a = 10;

    cout << "r:" << r << endl; // address of 'a'
    cout << "&r:" << &r << endl; // address of the pointer variable 'r'
    cout << "*r:" << *r << endl; // *r dereferences the pointer

    return 0;
}
```

```
r:0x16f0629a8
&r:0x16f0629a0
*r:10
```

When to use Reference vs Pointers?

When to Use a Reference

✓ 1. When the relationship should be permanent

A reference cannot be reseated, so use it when you want:

- a **parameter** that always refers to the same object
- an **alias** that must never be null
- simple, safe access to something without pointer complexity

Example (best practice):

Function parameters:

```
void setValue(int& x) {
    x = 10;
}
```

References make code cleaner and safer.

When to Use a Pointer

✓ 1. When you need to point to *different* objects over time

Pointers can change where they point:

```
int a = 10, b = 20;
int* p = &a;
p = &b;     // allowed
```

References *cannot* do this.

Quick Summary

Use For	Pointer	Reference
Can be null	✓	✗
Can change what it refers to	✓	✗
Simple alias (no pointer syntax)	✗	✓
Operator overloading	✗	✓
Dynamic memory	✓	✗
Arrays & pointer arithmetic	✓	✗
Low-level / hardware / OS code	✓	✗
Safe API (no null allowed)	✗	✓

Final Rule of Thumb

➤ Use references when you want simple, safe access and you know the object exists.

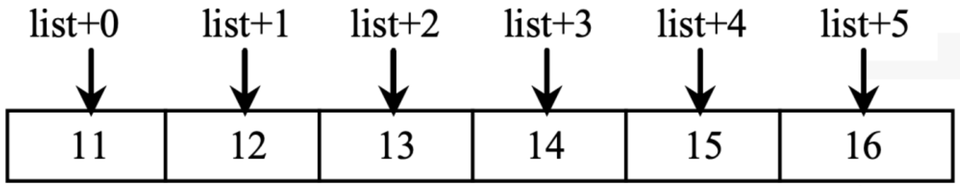
➤ Use pointers when you need flexibility, nullability, or low-level control.

Example 1 from Week 10

Example 1: Arrays And Pointers

An array variable without a bracket and a subscript actually represents the starting address of the array. In this sense, an array variable is essentially a pointer. Suppose you declare an array of int value as follows:

int list[6] = {11, 12, 13, 14, 15, 16};



int arr[5] = {1,2,3,4,5};

cout << arr;   // prints address of first element
cout << &arr[0]; // same address

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

int main(){
    int list[6] = {11, 12, 13, 14, 15, 16};

    cout << "list: " << list << endl;
    cout << "list: " << &list[0] << endl;
    cout << "list: " << &list[1] << endl;

    cout << "------------------" << endl;

    cout << "list: " << list+1 << endl;
    cout << "list: " << &list[0]+1 << endl;
    cout << "list: " << &list[1] << endl;

    return 0;
}
```

```
list: 0x16f33a990
list: 0x16f33a990
list: 0x16f33a994
```

?

Example 1 from Week 10

ARRAY POINTER

*(list + 1) is different from *list + 1. The dereference operator (*) has precedence over +. So, *list + 1 adds 1 to the value of the first element in the array, while *(list + 1) dereference the element at address (list + 1) in the array.

1. Write a complete program that uses pointers to access array elements.

2. Arrays and pointers form a close relationship. A pointer for an array can be used just like an array. You can even use pointer with index.

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

int main(){
    int list[6] = {11, 22, 33, 44, 55, 66};

    cout << "*(list + 1): " << *(list + 1)  << endl;
    cout << "*list + 1: "   << *list + 1    << endl;
    cout << "(list + 1): "  << (list + 1)   << endl;
    cout << "&list[1]: "    << &list[1]     << endl;
    cout << "list[1]: "     << list[1]      << endl;

    return 0;
}
```

```
*(list + 1): 22
*list + 1: 12
(list + 1): 0x16fa66994
&list[1]: 0x16fa66994
list[1]: 22
```

Today's Agenda

- Revise and run Example 1.
- Revise and run Example 2.

```
void f(int* p1, int* &p2)
```

which is equivalently to

```
typedef int* intPointer;
void f(intPointer p1, intPointer& p2)
```

- Revise and run Example 3-4.
- Briefing on Example 5 (30 min before the lab ends).

Download this PPT from:

Example 5 (Explicit vs Implicit Destructor)

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

class Course {
public:
    Course(const string& name) : courseName(name) {
        cout << "Course created: " << courseName << endl;
    }

    ~Course() {
        cout << "Course destroyed: " << courseName << endl;
    }

private:
    string courseName;
};

int main() {
    // Dynamically allocating a single Course object
    Course* myCourse = new Course("Math 101");

    // Use myCourse...

    // When myCourse goes out of scope, manually delete it to free memory
    delete myCourse;  // Destructor is called here

    return 0;
}
```

Output:

```
Course created: Math 101
Course destroyed: Math 101
```

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

class Course {
public:
    Course(const string& name) : courseName(name) {
        cout << "Course created: " << courseName << endl;
    }

    ~Course() {
        cout << "Course destroyed: " << courseName << endl;
    }

private:
    string courseName;
};

int main() {
    // Variable 'otherVariable' stays in the main function's scope
    int otherVariable = 10;

    {
        // Create a new scope
        Course myCourse("Math 101");

        // This object (myCourse) will be destroyed when this block ends.
        cout << "In inner scope, 'otherVariable' is: " << otherVariable << endl;
    } // 'myCourse' goes out of scope here, and the destructor is called

    cout << "Back in outer scope, 'otherVariable' is still: " << otherVariable << endl;

    return 0;
}
```

Output:

```
Course created: Math 101
In inner scope, 'otherVariable' is: 10
Course destroyed: Math 101
Back in outer scope, 'otherVariable' is still: 10
```

Example 5 (Shallow vs Deep Copy)

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

class Course {
public:
    string* students;   // ONLY variable we use (a pointer)

    // Constructor: allocate memory for ONE student (simple!)
    Course() {
        students = new string("John");
    }

    // ❌ Shallow copy constructor
    // Copies ONLY the pointer, not the data
    Course(const Course& other) {
        students = other.students;  // shallow copy → both point to same memory
    }

    string* getStudentsMemoryAddress() const {
        return students;
    }

    ~Course() {
        delete students;   // both objects will try to delete the same pointer → dangerou
    }
};

int main() {
    Course course1;        // create original
    Course course2(course1); // shallow copy

    cout << "course1 students pointer: " << course1.getStudentsMemoryAddress() << endl;
    cout << "course2 students pointer: " << course2.getStudentsMemoryAddress() << endl;

    return 0;
}
```

Output:

```
course1 students pointer: 0x556cbb4dd2a0
course2 students pointer: 0x556cbb4dd2a0
```

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

class Course {
public:
    string* students;   // ONLY variable we use (a pointer)

    // Constructor: allocate memory for ONE student
    Course() {
        students = new string("John");
    }

    // ✅ Deep copy constructor
    Course(const Course& other) {
        students = new string(*other.students);
        // ^ allocates new memory and copies the content
    }

    string* getStudentsMemoryAddress() const {
        return students;
    }

    ~Course() {
        delete students;
    }
};

int main() {
    Course course1;         // original
    Course course2(course1); // deep copy

    cout << "course1 students pointer: " << course1.getStudentsMemoryAddress() << endl;
    cout << "course2 students pointer: " << course2.getStudentsMemoryAddress() << endl;

    return 0;
}
```

Output:

```
course1 students pointer: 0x556cbb4dd2a0
course2 students pointer: 0x556cbb4dd330
```