Lab Week 2

1. Pointers

1.1 What is a Pointer?

In C++, a pointer is a variable that stores the memory address of another variable. Pointers are essential for dynamic memory management, array handling, and more.

1.2 Declaring and Initializing Pointers

```
int x = 10;
int* p; // Declaration of pointer p
p = &x; // Initialization: p stores the address of x
```

1.3 Dereferencing Pointers

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

int main() {
    int x = 10;
    int* p = &x;

    cout << "Value of x: " << *p << endl; //
Dereferencing: accessing the value at address p
    return 0;
}
```

1.4 Pointer Arithmetic

Pointer arithmetic allows you to navigate through arrays and other memory blocks.

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

int main() {
    int arr[3] = {10, 20, 30};
    int* p = arr;

    cout << "Value at p: " << *p << endl; // 10
    p++;
    cout << "Value at p: " << *p << endl; // 20
    return 0;
}
```

2. Dynamic Memory Allocation

Dynamic memory allocation in C++ is managed using `new` and `delete` operators.

2.1 `new` and `delete`

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

int main() {
    int* p = new int; // Allocates memory for one integer
    *p = 25;
    cout << "Value: " << *p << endl;
```

```
    delete p; // Deallocates memory
    return 0;
}
```

2.2 Dangling Pointers:

A dangling pointer in C++ is a pointer that continues to reference a memory location after the object it points to has been deleted. This can lead to undefined behavior if you try to access or modify the memory.

```
#include <iostream>

void danglingPointerExample() {
    int *ptr = new int; // Allocate memory
    *ptr = 10; // Assign a value

    std::cout << "Value before delete: " << *ptr <<
std::endl;

    delete ptr; // Deallocate memory

    // Accessing the pointer after delete
    // This is a dangling pointer now
    // std::cout << "Value after delete: " << *ptr <<
std::endl; // Undefined behavior

    ptr = nullptr; // Safe practice to avoid dangling
pointer
}

int main() {
    danglingPointerExample();
    return 0;
```

```
}
```

2.3 1D and 2D Dynamic Arrays

Dynamic arrays in C++ can be managed with `new` and `delete[]`. A 1D dynamic array is straightforward, while a 2D array is managed as an array of arrays.

```
#include <iostream>

void oneDDynamicArrayExample() {
    int size = 5;
    int *arr = new int[size]; // Allocate memory

    // Initialize and print the array
    for (int i = 0; i < size; ++i) {
        arr[i] = i * 10;
        std::cout << arr[i] << " ";
    }
    std::cout << std::endl;

    delete[] arr; // Deallocate memory
}

void twoDDynamicArrayExample() {
    int rows = 3, cols = 4;
    int **arr = new int*[rows]; // Allocate row pointers

    for (int i = 0; i < rows; ++i) {
        arr[i] = new int[cols]; // Allocate columns
    }

    // Initialize and print the 2D array
    for (int i = 0; i < rows; ++i) {
```

```
        for (int j = 0; j < cols; ++j) {
            arr[i][j] = i * j;
            std::cout << arr[i][j] << " ";
        }
        std::cout << std::endl;
    }

    // Free the memory
    for (int i = 0; i < rows; ++i) {
        delete[] arr[i];
    }
    delete[] arr;
}

int main() {
    oneDDynamicArrayExample();
     twoDDynamicArrayExample();
    return 0;
}
```

3. Passing Pointers to Functions

3.1 Passing a Pointer to Modify an Integer

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

// Function that takes a pointer to an integer and
modifies its value
void increment(int* ptr) {
```

```
    (*ptr)++; // Increment the value pointed to by ptr
}

int main() {
    int value = 10;
    cout << "Before increment: " << value << endl;

    increment(&value); // Pass the address of value to
the function

    cout << "After increment: " << value << endl;
    return 0;
}
```

3.2 Passing a Pointer to Modify an Array

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

// Function that takes a pointer to an array and modifies
its elements
void doubleArray(int* arr, int size) {
    for (int i = 0; i < size; ++i) {
        arr[i] *= 2; // Double each element of the array
    }
}
```

```
int main() {
    const int SIZE = 5;
    int arr[SIZE] = {1, 2, 3, 4, 5};

    cout << "Before doubling: ";
    for (int i = 0; i < SIZE; ++i) {
        cout << arr[i] << " ";
    }
    cout << endl;

    doubleArray(arr, SIZE); // Pass the array to the
function
    cout << "After doubling: ";
    for (int i = 0; i < SIZE; ++i) {
        cout << arr[i] << " ";
    }
    cout << endl;
    return 0;
}
```

4. Const Pointer vs Pointer to Const

Const Pointer: A pointer whose address cannot be changed, but the value at the address it points to can be modified.
Pointer to Const: A pointer that can change the address it points to, but cannot modify the value at that address.

```
#include <iostream>

void constPointerVsPointerToConstExample() {
    int value = 10;
    int anotherValue = 20;
```

```
    // Const Pointer
    int *const constPtr = &value; // Pointer itself is
constant
    *constPtr = 15; // Allowed
    // constPtr = &anotherValue; // Error: constPtr
cannot point to another address

    // Pointer to Const
    const int *ptrToConst = &value; // Pointer points to
const int
    // *ptrToConst = 15; // Error: Cannot modify value
through ptrToConst
    ptrToConst = &anotherValue; // Allowed: ptrToConst
can point to another address

    std::cout << "Value through const pointer: " <<
*constPtr << std::endl;
    std::cout << "Value through pointer to const: " <<
*ptrToConst << std::endl;
}

int main() {
    constPointerVsPointerToConstExample();
    return 0;
}
```

5. C-String Using Char Pointer

A C-string in C++ is an array of characters terminated by a null character (`'\0'`). You can use a `char` pointer to manage these strings dynamically.

```
#include <iostream>

void cStringUsingCharPointerExample() {
    const char *str = "Hello, world!";

    // Calculate the length of the string manually
    int length = 0;
    while (str[length] != '\0') {
        ++length;
    }

    // Allocate memory for a copy of the string
    char *copyStr = new char[length + 1]; // +1 for null
terminator
    if (copyStr == nullptr) {
        std::cerr << "Memory allocation failed" <<
std::endl;
        return;
    }

    // Copy the string manually
    for (int i = 0; i < length; ++i) {
        copyStr[i] = str[i];
    }
    copyStr[length] = '\0'; // Null terminator

    std::cout << "Original string: " << str << std::endl;
```

```
    std::cout << "Copied string: " << copyStr <<
std::endl;

    delete[] copyStr; // Deallocate memory
}

int main() {
    cStringUsingCharPointerExample();
    return 0;
}
```