Advanced Programming - Supplementary Examples

Pointers

This document provides a set of examples and exercises to supplement the lecture material on pointers. It is designed to help you practice and deepen your understanding of pointers, memory management, and related concepts.

1 Examples

Pointer Arithmetic with Arrays

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

int main() {
    int arr[5] = {1, 2, 3, 4, 5};
    int* p = arr; // p points to the first element of arr

    cout << "Array elements using pointer arithmetic:" << endl;
    for (int i = 0; i < 5; ++i) {
        cout << *(p + i) << " "; // Accessing array elements using pointer arithmetic
    }
    cout << endl;
    return 0;
}
```

Array of Pointers

This example shows how to create an array of pointers, where each pointer points to an integer that is dynamically allocated. In this example the integers are initialised with random values.

```cpp
#include <iostream>
#include <cstdlib>
#include <ctime>

int main() {
    const int size = 5;
    int* array[size];

    // seed for random number generation
    std::srand(std::time(0));

    // dynamically allocate memory for each integer
    for (int i = 0; i < size; ++i) {
        array[i] = new int;
        *array[i] = std::rand() % 100;
    }

    // print values and addresses
```

```
    for (int i = 0; i < size; ++i) {
        std::cout << "Value: " << *array[i] << ", Address: " << array[i] << std::endl;
    }

    // deallocate memory
    for (int i = 0; i < size; ++i) {
        delete array[i];
    }

    return 0;
}
```

Creating and Deallocating a 2D Array Dynamically

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

int main() {
    int rows = 3, cols = 3;
    int** arr = new int*[rows];
    for (int i = 0; i < rows; ++i) {
        arr[i] = new int[cols];
    }

    // initialising the 2D array
    for (int i = 0; i < rows; ++i) {
        for (int j = 0; j < cols; ++j) {
            arr[i][j] = i * cols + j;
        }
    }

    // printing the 2D array
    cout << "2D array elements:" << endl;
    for (int i = 0; i < rows; ++i) {
        for (int j = 0; j < cols; ++j) {
            cout << arr[i][j] << " ";
        }
        cout << endl;
    }

    // deallocating the 2D array
    for (int i = 0; i < rows; ++i) {
        delete[] arr[i];
    }
    delete[] arr;

    return 0;
}
```

Dynamic 2D Array Multiplication

Recalling the previous example, we can write a program that dynamically allocates two 2D arrays and multiplies them.

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

```

```
void multiplyMatrices(int** A, int** B, int** C, int rows, int cols) {
    for (int i = 0; i < rows; ++i) {
        for (int j = 0; j < cols; ++j) {
            C[i][j] = 0;
            for (int k = 0; k < cols; ++k) {
                C[i][j] += A[i][k] * B[k][j];
            }
        }
    }
}

int main() {
    int rows = 2, cols = 2;

    // allocate matrices A, B, and C
    int** A = new int*[rows];
    int** B = new int*[rows];
    int** C = new int*[rows];
    for (int i = 0; i < rows; ++i) {
        A[i] = new int[cols];
        B[i] = new int[cols];
        C[i] = new int[cols];
    }

    // initialize matrices A and B
    cout << "Enter elements of matrix A:" << endl;
    for (int i = 0; i < rows; ++i) {
        for (int j = 0; j < cols; ++j) {
            cin >> A[i][j];
        }
    }

    cout << "Enter elements of matrix B:" << endl;
    for (int i = 0; i < rows; ++i) {
        for (int j = 0; j < cols; ++j) {
            cin >> B[i][j];
        }
    }

    // multiply matrices A and B
    multiplyMatrices(A, B, C, rows, cols);

    // print result matrix C
    cout << "Resulting matrix C after multiplication:" << endl;
    for (int i = 0; i < rows; ++i) {
        for (int j = 0; j < cols; ++j) {
            cout << C[i][j] << " ";
        }
        cout << endl;
    }

    // deallocate matrices
    for (int i = 0; i < rows; ++i) {
        delete[] A[i];
        delete[] B[i];
        delete[] C[i];
    }
    delete[] A;
```

```
    delete[] B;
    delete[] C;

    return 0;
}
```

Exercise: Implement a program that uses `std::unique_ptr` to manage dynamic memory for the arrays in the previous examples.

Using Function Pointers

This example shows the use of function pointers.

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

void add(int a, int b) {
    cout << "Sum: " << a + b << endl;
}

void subtract(int a, int b) {
    cout << "Difference: " << a - b << endl;
}

int main() {
    // declare function pointer (operation)
    // include the parameters of the functions
    // we are aiming to point to
    void (*operation)(int, int);

    // define operation pointing to our add function
    // and call the function through the pointer
    operation = &add;
    operation(5, 3);

    // we can redefine operation, now pointing to subtract
    operation = &subtract;
    operation(5, 3);

    return 0;
}
```

Shared Resource Management with `std::shared_ptr`

This example demonstrates how two `std::shared_ptr` instances can manage the same dynamically allocated memory. To check this, the code in the example shows how the reference count changes as shared pointers are assigned.

```
#include <iostream>
#include <memory>

int main() {
    // Create a shared pointer
    std::shared_ptr<int> sp1(new int(10));
    std::cout << "sp1 use count: " << sp1.use_count() << std::endl;

    {
```

```
10          // Create another shared pointer to the same resource
11          std::shared_ptr<int> sp2 = sp1;
12          std::cout << "sp1 use count: " << sp1.use_count() << std::endl;
13          std::cout << "sp2 use count: " << sp2.use_count() << std::endl;
14      }
15
16      // sp2 is now out of scope, sp1 is the only owner
17      std::cout << "sp1 use count: " << sp1.use_count() << std::endl;
18
19      return 0;
20  }
```