LINEAR DATA STRUCTURES

Multi Dimensional Arrays, Sparse Arrays, Array of Pointers, Linked Lists, Stacks and Queues

Sashi Jagadam
jsashikala@gmail.com

Table of Contents

Linear Data Structures – Multi Dimensional Array

Multi-dimensional arrays are an extension of regular (one-dimensional) arrays, allowing us to create structures like 2D, 3D, or higher-dimensional arrays. These arrays are stored in contiguous memory, meaning that elements are laid out sequentially in memory. Multi-dimensional arrays are useful for representing matrices, grids, or higher-dimensional data structures.

Declaring Multi-Dimensional Arrays in C++
The syntax for declaring a multi-dimensional array is straightforward
type arrayName[size1][size2]...[sizeN];
int array[3][4];
This creates a 2D array with 3 rows and 4 columns, where each element can be accessed using two indices: array[row][column].

```
#include <iostream>

int main() {
   // Declare and initialize a 2D array
   int matrix[3][4] = {
      {1, 2, 3, 4},
      {5, 6, 7, 8},
      {9, 10, 11, 12}
   };

   // Display the matrix
   for (int i = 0; i < 3; ++i) {
      for (int j = 0; j < 4; ++j) {
         cout << matrix[i][j] << " ";
      }
      cout << endl;
   }

   return 0;
}
```

1. **Matrix Initialization**: The 2D array matrix is initialized with 3 rows and 4 columns.

2. **Accessing Elements**: Each element in the matrix can be accessed by specifying the row and column indices, matrix[i][j].
3. **Looping**: Nested for loops are used to iterate over the rows and columns.

Memory Layout of Multi-Dimensional Arrays

In memory, a 2D array like int matrix[3][4] is stored in a contiguous block of memory in **row-major order**. This means that the entire first row is stored first, followed by the second row, and so on. For example, the array matrix[3][4] in the example above would be stored in memory as:

Accessing Elements in a Multi-Dimensional Array

For a 2D array array[row][column], the memory address of an element can be calculated with the following formula:
address = base_address + (i *num_columns + j) * element_size

where:

- base_address is the starting address of the array in memory,
- i and j are the row and column indices,
- num_columns is the total number of columns in the array,
- element_size is the size of each element in bytes.

Higher-Dimensional Arrays

Higher-dimensional arrays are declared similarly. For example, a 3D array with dimensions 2x3x4 could be declared as follows:
int array[2][3][4];

Each additional dimension adds a level of indexing. Here's how you might initialize and access elements in a 3D array.

```
#include <iostream>

int main() {
   // Declare and initialize a 3D array
   int array[2][3][4] = {
```

```
        {
            {1, 2, 3, 4},
            {5, 6, 7, 8},
            {9, 10, 11, 12}
        },
        {
            {13, 14, 15, 16},
            {17, 18, 19, 20},
            {21, 22, 23, 24}
        }
    };

    // Display the 3D array
    for (int i = 0; i < 2; ++i) {
        for (int j = 0; j < 3; ++j) {
            for (int k = 0; k < 4; ++k) {
                cout << array[i][j][k] << " ";
            }
            cout << endl;
        }
        cout << endl;
    }

    return 0;
}
```

Dynamic Multi-Dimensional Arrays

In cases where the size of the array isn't known at compile-time, you can use **dynamic memory allocation** with pointers to create multi-dimensional arrays.

```
#include <iostream>

int main() {
    int rows = 3;
    int cols = 4;

    // Create a dynamic 2D array
    int** matrix = new int*[rows];
```

```
    for (int i = 0; i < rows; ++i) {
        matrix[i] = new int[cols];
    }

    // Initialize and display the 2D array
    int value = 1;
    for (int i = 0; i < rows; ++i) {
        for (int j = 0; j < cols; ++j) {
            matrix[i][j] = value++;
            cout << matrix[i][j] << " ";
        }
        cout << endl;
    }

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

    return 0;
}
```

1. **Allocate Memory**: We create an array of pointers for rows, and then each pointer is allocated an array of integers for columns.
2. **Initialize and Display**: Values are assigned, and then the matrix is displayed.
3. **Deallocation**: Since memory was dynamically allocated, it needs to be freed using delete[] to avoid memory leaks.

Summary

- **Multi-dimensional arrays** in C++ allow you to handle complex data structures like matrices and 3D grids.
- **Contiguous memory** layout provides efficient access but can consume large amounts of memory in higher dimensions.
- **Dynamic allocation** allows for flexible array sizes but requires careful memory management.

Advantages of Multi-Dimensional Arrays

1. **Structured Data Representation**: Useful for representing grids, tables, images, etc.
2. **Efficient Memory Layout**: Stored contiguously, which helps in achieving efficient cache utilization and performance.
3. **Flexible Access Patterns**: With dynamic allocation, the size of arrays can be flexible and adjusted at runtime.

Summary

- Multi-dimensional arrays in C++ are a way to represent structured data in rows, columns, and higher dimensions.
- They are stored in row-major order in contiguous memory, making them efficient for linear traversal.
- Dynamic allocation allows flexible sizing but requires careful memory management.
- Common applications include matrices, image data, and grids in numerical simulations.

Bank System example using multi-dimensional dynamic arrays
Vector or vector is used in the example to store multiple bank accounts pertaining to each of the bankbranch.

```
// Online C++ compiler to run C++ program online
#include <iostream>
#include <string>
#include <vector>

using namespace std;

class BankAccount {
protected:
   int accountNumber;        // Unique account number
   string accountHolder;      // Name of the account holder
   double balance;           // Account balance

public:
   // Constructor
```

```
    BankAccount(int accNum, string holder, double initialBalance)
        : accountNumber(accNum), accountHolder(holder), balance(initialBalance) {
        cout << "BankAccount created for " << accountHolder << " with initial balance
$" << balance << endl;
    }

    // Method to deposit money into the account
    void deposit(double amount) {
        if (amount > 0) {
            balance += amount;
            cout << "Deposited: $" << amount << ". New balance: $" << balance <<
endl;
        } else {
            cout << "Deposit amount must be positive!" << endl;
        }
    }

    // Method to withdraw money from the account
    void withdraw(double amount) {
        if (amount > balance) {
            cout << "Insufficient funds. Withdrawal of $" << amount << " failed." <<
endl;
        } else if (amount <= 0) {
            cout << "Withdrawal amount must be positive!" << endl;
        } else {
            balance -= amount;
            cout << "Withdrew: $" << amount << ". New balance: $" << balance <<
endl;
        }
    }

    // Method to display account information
    void display() const {
        cout << "Account Number: " << accountNumber << endl;
        cout << "Account Holder: " << accountHolder << endl;
        cout << "Balance: $" << balance << endl;
    }
```

```
    int getAccountNumber()const
    {
        return accountNumber;
    }

    // Virtual destructor
    /*virtual ~BankAccount() {
        cout << "BankAccount for " << accountHolder << " is being closed." << endl;
    }*/
};

class BankBranch
{
    private:
    string branchName;
    string branchAddress;
    vector<BankAccount> accounts;

    public:
    BankBranch(const string& name, const string& address) :
        branchName(name), branchAddress(address){}

    void addAccount(int accountNum, const string& accountName, double balance)
    {
        BankAccount newAcc(accountNum, accountName, balance);
        accounts.push_back(newAcc);
        cout<<"Account created for "<<accountName <<" with account number:
"<<accountNum <<endl;
    }

    //Find an account number
    BankAccount* findAccount(int accountNum)
    {
        for(auto& account: accounts)
        {
            if(account.getAccountNumber() == accountNum)
            {
                return &account;
```

```
            }
        }
        cout<<"Account number: "<<accountNum<<endl;
        return nullptr;
    }

    //Display all accounts
    void displayAllAccs() const
    {
        cout<<"All bank accounts: "<<endl;
        for(const auto& account: accounts)
        {
            account.display();
        }
    }

    string getBranch()const
    {
        return branchName;
    }
};

class BankSystem
{
    private:
    vector<vector<BankAccount>> branches;

    public:
    //Initialize specific number of branches in the constructor
    BankSystem(int noOfBranches)
    {
        branches.resize(noOfBranches);
    }

    void addAccountToBranch(int branchIndex, const BankAccount account)
    {
        if(branchIndex < branches.size())
        {
```

```
        branches[branchIndex].push_back(account);
    }
    else
    {
        cout<<" Branch index out of range" << endl;
    }
}

//Display all branches
void displayAllBranches() const
{
    for(int i=0; i< branches.size(); i++)
    {
        for(const auto& account : branches[i])
        {
            account.display();
        }
    }
}

void depositToAccount(int branchIndex, int accountNum, double amount)
{
    if(branchIndex < branches.size())
    {
        for(auto& account: branches[branchIndex])
        {
            if(account.getAccountNumber() == accountNum)
            {
                account.deposit(amount);
                cout<<"Deposited " <<amount<< " to account "<< accountNum<< " in
branch "<< branchIndex<<endl;
                return;
            }
        }
        cout<<"Account not found in branch"<<endl;
    }
    else
    {
```

```
            cout<<"Branch index out of range"<<endl;
        }
    }

    void withdrawFromAccount(int branchIndex, int accountNum, double amount)
    {
        if(branchIndex < branches.size())
        {
            for(auto& account: branches[branchIndex])
            {
                if(account.getAccountNumber() == accountNum)
                {
                    account.withdraw(amount);
                    return;
                }
            }
            cout<<"Account not found in branch "<< branchIndex << endl;
        }
        else
        {
            cout<<"Branch index out of range"<<endl;
        }
    }
};

int main() {
    BankSystem bank(2);
    bank.addAccountToBranch(0, BankAccount(10001, "User1", 10000.00));
    bank.addAccountToBranch(0, BankAccount(2001, "User2", 100.00));
    bank.addAccountToBranch(1, BankAccount(3001, "User21", 500.00));
    bank.addAccountToBranch(1, BankAccount(4001, "User22", 500000.00));
    bank.addAccountToBranch(1, BankAccount(50001, "User23", 40000.00));

    bank.displayAllBranches();
    cout<<"Deposit"<<endl;
    bank.depositToAccount(0, 2001, 300.0);
    bank.depositToAccount(1, 50001, 1000.0);
```

```
    bank.withdrawFromAccount(1, 50001, 30000.0);
    bank.withdrawFromAccount(0, 10001, 5000.0);

    return 0;
}
```

Sparse Arrays

Sparse arrays (or sparse matrices) are data structures that are used to efficiently store and manage arrays where most elements are zero or have no value. Representing sparse data with regular multi-dimensional arrays would waste memory and computation on all the zero or empty elements. Instead, we can use specialized structures to store only the non-zero elements, optimizing memory usage and processing time.

Sparse Array Representation

There are multiple ways to represent sparse arrays, such as:

1. **Coordinate List (COO)**: A simple list of (row, column, value) for each non-zero element.
2. **Compressed Sparse Row (CSR)**: Uses three arrays to store row pointers, column indices, and values.
3. **Dictionary of Keys (DOK)**: Uses a dictionary (like map) where keys are coordinates (row, column), and values are the non-zero entries.

Example of a Sparse Array

Here's an example of implementing a sparse array using the **Coordinate List (COO)** approach. This is often the simplest way to represent sparse matrices.

```
#include <iostream>
#include <vector>
#include <tuple>

class SparseArray {
private:
    int rows;
    int cols;
    vector<tuple<int, int, int>> elements; // Stores (row, col, value)

public:
```

```
    // Constructor
    SparseArray(int r, int c) : rows(r), cols(c) {}

    // Add a non-zero element
    void addElement(int row, int col, int value) {
        if (value != 0) {
            elements.push_back(make_tuple(row, col, value));
        }
    }

    // Display the sparse array
    void display() const {
        cout << "Sparse Array Elements (row, col, value):\n";
        for (const auto& elem : elements) {
            int row, col, value;
            tie(row, col, value) = elem;
            cout << "(" << row << ", " << col << ", " << value << ")\n";
        }
    }

    // Get the element at a specific row and column
    int getElement(int row, int col) const {
        for (const auto& elem : elements) {
            if (get<0>(elem) == row && get<1>(elem) == col) {
                return get<2>(elem);
            }
        }
        return 0; // If element is not stored, it is considered zero
    }
};

int main() {
    // Create a sparse array with 4 rows and 5 columns
    SparseArray sparseArray(4, 5);

    // Add some non-zero elements
    sparseArray.addElement(0, 1, 5);
    sparseArray.addElement(1, 3, 10);
```

```
    sparseArray.addElement(3, 2, 8);

    // Display sparse array
    sparseArray.display();

    // Access an element
    cout << "\nElement at (1, 3): " << sparseArray.getElement(1, 3) << endl;
    cout << "Element at (2, 2): " << sparseArray.getElement(2, 2) << endl;

    return 0;
}
```

Explanation

1. **Data Storage**: The sparse array is represented by a vector of tuples (row, col, value) storing only the non-zero elements.
2. **Adding Elements**: The addElement method checks if the value is non-zero before adding it to the list of elements.
3. **Accessing Elements**: The getElement method iterates through the list to find a matching (row, col) tuple. If the element is not found, it is considered zero.

Optimizations and Other Representations

1. **Compressed Sparse Row (CSR)**: Instead of storing (row, col, value) tuples, CSR uses three arrays:

- **values**: Stores all non-zero values.
- **column indices**: Stores column indices corresponding to each value.
- **row pointers**: Stores the index in values where each row starts.

CSR representation is more efficient for matrix-vector multiplication.

2. **Dictionary of Keys (DOK)**: This approach uses a map<pair<int, int>, int> in C++, where keys are (row, col) pairs. This makes element access and updates very fast since map provides logarithmic time complexity for insertions and lookups.

Example of a Sparse Array Using map

```
#include <iostream>
#include <map>

class SparseArrayMap {
```

```
private:
    int rows;
    int cols;
    map<pair<int, int>, int> elements;

public:
    SparseArrayMap(int r, int c) : rows(r), cols(c) {}

    // Add or update an element
    void setElement(int row, int col, int value) {
        if (value != 0) {
            elements[{row, col}] = value;
        } else {
            elements.erase({row, col}); // Remove if value is zero
        }
    }

    // Get the element at a specific row and column
    int getElement(int row, int col) const {
        auto it = elements.find({row, col});
        if (it != elements.end()) {
            return it->second;
        }
        return 0; // If not found, it's zero
    }

    // Display the sparse array
    void display() const {
        cout << "Sparse Array Elements (row, col, value):\n";
        for (const auto& elem : elements) {
            cout << "(" << elem.first.first << ", " << elem.first.second << ", " <<
elem.second << ")\n";
        }
    }
};

int main() {
    SparseArrayMap sparseArray(4, 5);
```

```
    sparseArray.setElement(0, 1, 5);
    sparseArray.setElement(1, 3, 10);
    sparseArray.setElement(3, 2, 8);

    sparseArray.display();

    cout << "\nElement at (1, 3): " << sparseArray.getElement(1, 3) << endl;
    cout << "Element at (2, 2): " << sparseArray.getElement(2, 2) << endl;

    return 0;
}
```

Benefits of Using Sparse Arrays

1. **Memory Efficiency**: Only non-zero elements are stored, making it memory-efficient for large sparse matrices.
2. **Performance**: Sparse representation improves performance when working with large matrices, as fewer elements are processed.
3. **Flexibility**: The map representation makes it easy to access, insert, or delete elements dynamically.

Applications of Sparse Arrays

- **Scientific Computing**: Representing large matrices in numerical simulations.
- **Graph Representations**: Adjacency matrices for sparse graphs.
- **Machine Learning**: Sparse matrices are common in data structures for text processing and recommendation systems.

Sparse arrays in C++ provide efficient ways to manage large datasets with minimal storage needs, making them an essential structure in performance-critical applications.

Sparse Array using Dictionary of Keys (DOK)

Here's a C++ implementation of a **SparseArray** class. This class uses a **Dictionary of Keys (DOK)** approach, where a map is used to store only non-zero elements by their (row, column) coordinates. This approach is memory-efficient and provides fast access and updates for sparse matrices.

```
#include <iostream>
#include <map>
```

```
class SparseArray {
private:
   int rows;
   int cols;
   map<pair<int, int>, int> elements; // Only stores non-zero elements

public:
   // Constructor to initialize array dimensions
   SparseArray(int r, int c) : rows(r), cols(c) {}

   // Method to set an element in the sparse array
   void setElement(int row, int col, int value) {
      if (value != 0) {
         elements[{row, col}] = value; // Store non-zero value
      } else {
         elements.erase({row, col}); // Remove the element if the value is zero
      }
   }

   // Method to get the value at a specific row and column
   int getElement(int row, int col) const {
      auto it = elements.find({row, col});
      if (it != elements.end()) {
         return it->second; // Return the stored non-zero value
      }
      return 0; // Return zero if element is not stored
   }

   // Display all non-zero elements in the sparse array
   void display() const {
      cout << "Sparse Array Elements (row, col, value):\n";
      for (const auto& elem : elements) {
         cout << "(" << elem.first.first << ", " << elem.first.second << ", " <<
elem.second << ")\n";
      }
   }
};
```

```
int main() {
    // Create a sparse array of size 4x5
    SparseArray sparseArray(4, 5);

    // Set some non-zero values
    sparseArray.setElement(0, 1, 5);
    sparseArray.setElement(1, 3, 10);
    sparseArray.setElement(3, 2, 8);

    // Display the non-zero elements of the sparse array
    sparseArray.display();

    // Access specific elements
    cout << "\nElement at (1, 3): " << sparseArray.getElement(1, 3) << endl;
    cout << "Element at (2, 2): " << sparseArray.getElement(2, 2) << endl;

    return 0;
}
```

Explanation

Data Storage:

- elements is a map with keys as pair<int, int> (representing (row, col) coordinates) and values as integers. This map stores only non-zero values, so memory is conserved.

Setting an Element:

1. The setElement method adds an element to the map only if it's non-zero. If a zero value is set, the method removes the element from the map, reflecting the sparse nature of the array.

Getting an Element:

1. The getElement method checks if a given (row, col) coordinate exists in elements. If found, it returns the stored non-zero value; otherwise, it returns zero by default.

Displaying Elements:

- The display method outputs all non-zero elements, each with its (row, col) coordinate.

Key Benefits of Using This Sparse Array Class

- **Memory Efficiency**: By storing only non-zero elements, this class is efficient for large arrays with sparse data.
- **Fast Access and Update**: Using map enables efficient access and updates, which are typically O(log n) in complexity.
- **Flexibility**: It's easy to add, retrieve, or delete non-zero elements, making this class versatile for various sparse data applications.

Potential Enhancements

- **Boundary Checks**: You could add checks in setElement and getElement to ensure that row and column indices are within valid ranges.
- **Alternative Representations**: For specific applications, a different representation (like CSR) might offer even faster access in certain patterns, especially for mathematical operations on large matrices.

This class provides a simple yet effective way to handle sparse data in C++, commonly used in applications like scientific computing, image processing, and machine learning for efficient memory and computational performance.

Example of Sparse Array using CSR representation

```
// Online C++ compiler to run C++ program online
#include <iostream>
#include <vector>
using namespace std;

class SparseArrayCSR
{
   private:
   vector<int> values;
   vector<int> colVec;
   vector<int> rowVec;

   int nRow;
   int cCol;

   public:
   SparseArrayCSR(int rows, int cols): nRow(rows), cCol(cols)
   {
```

```
    rowVec.resize(rows + 1, 0);
  }

  //Add non-zero values
  void addValues(int row, int col, int value)
  {
    if(value == 0)
      return;
    values.push_back(value);
    colVec.push_back(col);
    cout<<"rowVec size "<<rowVec.size()<<endl;;
    rowVec.push_back(row);
    //update row vector
    /*for(int i = row; i< nRow; i++)
    {
      rowVec[i]++;
    }*/
  }

  //get elements of the sparse array
  int getValue(int row, int col) const
  {
    int rowStart = rowVec[row];
    int rowEnd = rowVec[row + 1];

    //iterate through the rowvec
    for(int i = rowStart; i< rowEnd; i++)
    {
      if(colVec[i] == col)
      {
        return values[i];
      }
    }
    return 0; //Return 0 if value not found
  }

  void displayCSR() const
  {
```

```
        cout<<"Values: ";
        for(int val: values)
            cout<< val<<" ";
        cout<<endl<<"Column index: ";
        for(int col: colVec)
            cout<<col<<" ";
        cout<<endl<<"Row index: ";
        for(int row: rowVec)
            cout<<row<<" ";
            cout<<"Row size: "<<rowVec.size();
    }
};

int main() {
    SparseArrayCSR csr(4, 4);
    csr.addValues(0, 1, 0);
    csr.addValues(1, 0, 10);
    csr.addValues(1, 1, 0);
    csr.addValues(2, 0, 30);
    csr.addValues(2, 1, 40);
    csr.addValues(3, 1, 50);
    csr.addValues(5, 6, 0);
    cout<<"Display: "<<endl;
    csr.displayCSR();
    return 0;
}
```

Array of Pointers

An **array of pointers** is a collection where each element is a pointer pointing to another data element, such as an individual variable or a dynamically allocated memory block. This approach allows greater flexibility in managing data, especially in dynamic and heterogeneous scenarios.

Why Use an Array of Pointers?

Efficient Memory Usage:

- Only memory for pointers is allocated initially. The actual data can be dynamically allocated as needed.

Dynamic Storage:

- Pointers can point to dynamically allocated arrays or objects of varying sizes.

Flexibility:

- Enables implementing complex data structures like jagged arrays, polymorphism in arrays of objects, etc.

Syntax:

type* pointerArray[size];

- type*: The type of data the pointers will point to.
- pointerArray: The name of the array.
- size: The number of pointers in the array.

Examples:

1. Basic Array of Pointers

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

int main() {
   int a = 10, b = 20, c = 30;

   // Array of pointers
   int* ptrArray[3];

   // Assign addresses of variables
   ptrArray[0] = &a;
   ptrArray[1] = &b;
   ptrArray[2] = &c;

   // Access and modify variables using the array of pointers
   for (int i = 0; i < 3; i++) {
      cout << "Value at ptrArray[" << i << "] = " << *ptrArray[i] <<endl;
      *ptrArray[i] += 10; // Modify the value
   }

   cout << "\nAfter Modification:\n";
   cout << "a = " << a << ", b = " << b << ", c = " << c << endl;
```

```
    return 0;
}
```

2. Array of pointers with dynamic memory allocation

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

int main() {
    // Array of pointers
    int* ptrArray[3];

    // Allocate dynamic memory for each pointer
    for (int i = 0; i < 3; i++) {
        ptrArray[i] = new int(i * 10); // Allocate memory and initialize
    }

    // Access and print dynamically allocated values
    for (int i = 0; i < 3; i++) {
        cout << "Value at ptrArray[" << i << "] = " << *ptrArray[i] << endl;
    }

    // Deallocate dynamic memory
    for (int i = 0; i < 3; i++) {
        delete ptrArray[i];
    }

    return 0;
}
```

3. Jagged Arrays (Array of pointers to arrays)

A **jagged array** is an array of pointers where each pointer points to an array of potentially different lengths.

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

int main() {
```

```
    int rows = 3;

    // Array of pointers
    int* jaggedArray[3];

    // Allocate memory for each row
    jaggedArray[0] = new int[2]{1, 2};       // Row 1: 2 elements
    jaggedArray[1] = new int[3]{3, 4, 5};    // Row 2: 3 elements
    jaggedArray[2] = new int[4]{6, 7, 8, 9}; // Row 3: 4 elements

    // Print jagged array
    for (int i = 0; i < rows; i++) {
        int size = (i + 2); // Example: Row i has i+2 elements
        for (int j = 0; j < size; j++) {
            cout << jaggedArray[i][j] << " ";
        }
        cout << endl;
    }

    // Deallocate memory
    for (int i = 0; i < rows; i++) {
        delete[] jaggedArray[i];
    }

    return 0;
}
```

4. Array of Pointers to object

This approach is useful in Object-Oriented Programming for polymorphism.

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

class BankAccount {
private:
    string accountHolder;
    double balance;
```

```
public:
   BankAccount(string name, double bal) : accountHolder(name), balance(bal) {}

   void display() const {
      cout << "Account Holder: " << accountHolder << ", Balance: $" << balance <<
endl;
   }
};

int main() {
   // Array of pointers to objects
   BankAccount* accounts[2];

   // Dynamically allocate memory for objects
   accounts[0] = new BankAccount("Alice", 1500.75);
   accounts[1] = new BankAccount("Bob", 2000.50);

   // Access objects through the array
   for (int i = 0; i < 2; i++) {
      accounts[i]->display();
   }

   // Deallocate memory
   for (int i = 0; i < 2; i++) {
      delete accounts[i];
   }

   return 0;
}
```

Key points

Initialization:

- Always initialize pointers to avoid undefined behaviour.

Memory Management:

- Dynamically allocated memory must be freed using delete or delete[] to avoid memory leaks.

Flexibility:

- Arrays of pointers can adapt to various use cases, like jagged arrays, dynamic allocation, and heterogeneous collections.

Complexity:

- Care must be taken to ensure proper allocation and deallocation, as dangling pointers or memory leaks can lead to bugs.

Use Cases

Dynamic Structures:

- Implementing dynamic data structures such as linked lists, graphs, and trees.

Jagged Arrays:

- Used in scenarios where rows or dimensions have different lengths (e.g., matrices with varying column sizes).

Object Arrays:

- Managing collections of polymorphic objects (e.g., an array of base class pointers pointing to derived class objects).

By combining pointers with arrays, C++ provides powerful tools for handling complex memory and data management tasks.

Example: Build Mario game using jagged array as vector of vector

```
// Online C++ compiler to run C++ program online
#include <iostream>
#include <vector>
using namespace std;

class MarioGame
{
   vector<vector<char>> game; //game map represented as vector of vectors
   int mRow; //Mario's current row
   int mCol; //Mario's current column

   public:
   MarioGame(int rows, const vector<int>& cols)
   {
      game.resize(rows); //Set the number of rows
      for(int i =0; i< rows; i++)
```

```
    {
        game[i].resize(cols[i]); //Set number of columns for each row
    }
    //Initialize position of mario
    mRow = 0;
    mCol = 0;
}

//Initialize game with obstacles
void initializeGame()
{
    cout<<"Enter the elements for the game (use '.', '#', 'C', 'E'): "<<endl;
    cout<<" '.' for empty, '#' for obstable. 'C' for coin, 'E' for enemy"<<endl;
    for(int i = 0; i< game.size(); i++)
    {
        for (int j =0; j< game[i].size(); j++)
        {
            cin>>game[i][j];
        }
    }
    //Place Mario at starting position
    game[mRow][mCol] = 'M';
}

//Display current game
void displayGame() const
{
    cout<<"Game Layout"<<endl;
    for(const auto& row: game)
    {
        for(char cell: row)
        {
            cout<<cell <<" ";
        }
        cout<<endl;
    }
}
```

```
    //Move mario based on user input
    void moveMario(char direction)
    {
        int newRow = mRow;
        int newCol = mCol;
        if(direction == 'w' && mRow > 0) newRow--; //Move Up
        else if(direction == 's' && mRow< game.size() - 1) newRow++; //move down
        else if(direction == 'a' && mCol > 0) newCol--; //move left
        else if(direction == 'd' && mCol < game[mRow].size() -1) newCol++; //move
right
        else
        {
            cout<<"Invalid move" <<endl;
            return;
        }
        char nextCell = game[newRow][newCol];
        if(nextCell == '#')
        {
            cout<<"Hit an obstacle"<<endl;
            //updatePosition(newRow, newCol);
        }
        else if(nextCell == 'C')
        {
            cout<<"Collected coin:"<<endl;
            updatePosition(newRow, newCol);
        }
        else if(nextCell == 'E')
        {
            cout<<"Encountered enemy. Game Over!"<<endl;
            exit(0);
        }
        else
        {
            updatePosition(newRow, newCol);
        }
    }

    //Update Mario's position
```

```
    void updatePosition(int newRow, int newCol)
    {
        game[mRow][mCol] = '.'; //Clear old position
        mRow = newRow;
        mCol = newCol;
        game[mRow][mCol] = 'M'; //Update Mario's position
    }
};

int main() {
    int rows;
    //Input number of rows for Mario game
    cout<<"Enter number of rows for the game. ";
    cin>>rows;

    vector<int> cols(rows);
    for(int i=0; i< rows; i++)
    {
        cout<<"Enter number of columns for each row. ";
        cin>>cols[i];
    }
    MarioGame mario(rows, cols);
    mario.initializeGame();
    char cmd;
    while(true)
    {
        mario.displayGame();
        cout<<"Enter direction. [w: up, s: down, a: left, d: right, q: quit]: ";
        cin>>cmd;
        if(cmd == 'q')
        {
            cout<<"Exiting the game"<<endl;
            return 0;
        }
        mario.moveMario(cmd);
    }

    return 0;
```

```
}
```

LinkedList

Example:

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

struct Node
{
   int data;   //Data part of the list
   Node* next;

   Node(int value)
   {
      data = value;
      next = nullptr;
   }
};

class LinkedList
{
   private:
   Node* head;

   public:
   LinkedList()
   {
      head = nullptr;
   }

   void insertAtEnd(int value)
   {
      Node* newNode = new Node(value);
      if(head == nullptr)
      {
         cout<<"List is empty"<<endl;
         head = newNode;
```

```
    }
    else
    {
        Node* temp = head;
        while(temp->next != nullptr)
        {
            temp = temp->next;
        }
        temp->next = newNode;
    }
}

void deleteNode(int value)
{
    if(head == nullptr)
    {
        cout<<"Empty list. No elements to delete"<<endl;
        return;
    }
    //Check if node to be deleted is head node
    if(head->data == value)
    {
        Node* temp = head;
        head = head->next;
        delete temp;
        cout<<"Node with value "<<value<<" deleted"<<endl;
        return;
    }
    //Traverse the list to find the node to be deleted
    Node* temp = head;
    Node* prev = nullptr;
    while(temp != nullptr && temp->data != value)
    {
        prev = temp;
        temp = temp->next;
    }
    //If value not found
    if(temp == nullptr)
```

```
        {
            cout<<"Node with value "<<value<<" not found in the list"<<endl;
            return;
        }

        //unlink node and delete it
        prev->next = temp->next;
        delete temp;
        cout<<"Node with value "<<value<<" deleted"<<endl;
    }

    void display()const
    {
        if(head == nullptr)
        {
            cout<<"List is empty"<<endl;
            return;
        }
        Node* temp = head;
        while(temp != nullptr)
        {
            cout<<temp->data<<" -> ";
            temp = temp->next;
        }
        cout<<"NULL"<<endl;
    }

    ~LinkedList()
    {
        Node*  temp;
        while(head != nullptr)
        {
            temp = head;
            head = head->next;
            delete temp;
        }
    }
};
```

```
int main()
{
   LinkedList llist;
   for(int i = 0; i< 10; i++)
      llist.insertAtEnd((i+1)*10);
   llist.display();
   llist.deleteNode(50);
   llist.deleteNode(110);
   llist.deleteNode(10);
   llist.deleteNode(100);
   llist.display();
   return 0;
}

//Reverse Linked list in groups
//Reverse Linked list in groups
#include <iostream>
using namespace std;

struct Node
{
   int data;   //Data part of the list
   Node* next;

   Node(int value)
   {
      data = value;
      next = nullptr;
   }
};

class LinkedList
{
   private:
   Node* head;

   public:
```

```
LinkedList()
{
   head = nullptr;
}

void insertAtEnd(int value)
{
   Node* newNode = new Node(value);
   if(head == nullptr)
   {
      cout<<"List is empty"<<endl;
      head = newNode;
   }
   else
   {
      Node* temp = head;
      while(temp->next != nullptr)
      {
         temp = temp->next;
      }
      temp->next = newNode;
   }
}

Node* reverseListInGroup(Node* head, int k)
{
   Node* current = head;
   Node* prev = nullptr;
   Node* nnext = nullptr;

   if(!head || k<=1)
      return head;

   // Reverse the first k nodes
int count = 0;
Node* temp = head;
while (temp && count < k)
{
```

```
    count++;
    temp = temp->next;
  }

  if (count < k) {
    return head;   // If fewer than k nodes remain, do not reverse
  }

   count = 0;
  while(current && count < k)
  {
        nnext = current->next;    // Save the next node
        current->next = prev;    // Reverse the current node
        prev = current;          // Move prev to the current node
        current = nnext;          // Move curr to the next node
        count++;
  }
  // Now `head` is the tail of the reversed group; connect it to the next group
  if (nnext) {
    head->next = reverseListInGroup(nnext, k);
  }

  // Return the new head of this group
  return prev;

  }

  void reverseInGroups(int k)
  {
    head = reverseListInGroup(head,  k);
  }

  void reverseList()
  {
    Node* current = head;
    Node* prev = nullptr;
    Node* nnext = nullptr;
    while(current != nullptr)
```

```
    {
        nnext = current->next;
        current->next = prev;
        prev = current;
        current = nnext;
    }
    head = prev;
}

void deleteNode(int value)
{
    if(head == nullptr)
    {
        cout<<"Empty list. No elements to delete"<<endl;
        return;
    }
    //Check if node to be deleted is head node
    if(head->data == value)
    {
        Node* temp = head;
        head = head->next;
        delete temp;
        cout<<"Node with value "<<value<<" deleted"<<endl;
        return;
    }
    //Traverse the list to find the node to be deleted
    Node* temp = head;
    Node* prev = nullptr;
    while(temp != nullptr && temp->data != value)
    {
        prev = temp;
        temp = temp->next;
    }
    //If value not found
    if(temp == nullptr)
    {
        cout<<"Node with value "<<value<<" not found in the list"<<endl;
        return;
```

```
        }

        //unlink node and delete it
        prev->next = temp->next;
        delete temp;
        cout<<"Node with value "<<value<<" deleted"<<endl;
    }

    void display()const
    {
        if(head == nullptr)
        {
            cout<<"List is empty"<<endl;
            return;
        }
        Node* temp = head;
        while(temp != nullptr)
        {
            cout<<temp->data<<" -> ";
            temp = temp->next;
        }
        cout<<"NULL"<<endl;
    }

    ~LinkedList()
    {
        Node*  temp;
        while(head != nullptr)
        {
            temp = head;
            head = head->next;
            delete temp;
        }
    }
};

int main()
{
```

```
   LinkedList llist;
   for(int i = 0; i< 10; i++)
      llist.insertAtEnd((i+1)*10);
   llist.display();
   llist.reverseList();
   llist.display();
   llist.deleteNode(50);
   llist.deleteNode(110);
   llist.display();
   llist.reverseInGroups(4);
   llist.display();
   llist.deleteNode(10);
   llist.deleteNode(100);
   llist.display();
   //llist.reverseList();
   llist.reverseInGroups(4);
   llist.display();
   return 0;
}

//Reverse linked list by values
// Online C++ compiler to run C++ program online
#include <iostream>
#include <vector>
using namespace std;

class Node
{
   public:
   int data;   //Data part of the list
   Node* next;
   Node(int value)
   {
      data = value;
      next = nullptr;
   }
};
```

```
class LinkedList
{
   private:
   Node* head;

   public:
   LinkedList() : head(nullptr){}
   //add value at the end of the list
   void append(int value)
   {
      Node* newNode =new Node(value);
      if(!head)
      {
         head = newNode;
      }
      else
      {
         Node* temp = head;
         while(temp->next)
         {
            temp = temp->next;
         }
         temp->next = newNode;
      }
   }

   //Reverse list by values
   void reverseList()
   {
      if(!head || !head->next) //Empty or single node list then return
      {
         return;
      }
      vector<int> values;
      Node* temp = head;

      //Store all values in a vector
      while(temp)
```

```
        {
            values.push_back(temp->data);
            temp = temp->next;
        }
        //write values back in reverse order
        temp = head;
        for(int i = values.size() -1; i>=0; i--)
        {
            temp->data = values[i];
            temp = temp->next;
        }
    }

    void display() const
    {
        Node* temp = head;
        while(temp)
        {
            cout<<temp->data<<" ";
            temp = temp->next;
        }
        cout<<endl;
    }
    ~LinkedList()
    {
        while(head)
        {
            Node* temp = head;
            head = head->next;
            delete temp;
        }
    }

};

int main() {
    LinkedList llist;
    for(int i = 0; i< 10; i++)
```

```
        llist.append((i+1)*10);
    llist.display();
    llist.reverseList();
    llist.display();

    return 0;
}

//Swap nodes by address in Singly Linked list
//Task Manager example where tasks are added to the list
#include <iostream>
#include <string>
using namespace std;

class Task
{
    public:
    int taskId;
    string description;

    Task* next;
    Task(int id, string desc) : taskId(id), description(desc), next(nullptr){}
};

class TaskManager
{
    private:
    Task* head;
    int taskCnt;

    public:
    TaskManager() : head (nullptr), taskCnt(0) {}

    void addTask(const string& taskDesc)
    {
        Task* newTask = new Task(++taskCnt, taskDesc);  //Create new task
        if(head == nullptr)
        {
```

```
        head = newTask; //First task becomes the head
    }
    else
    {
        Task* temp = head;
        while(temp->next != nullptr)
        {
            temp = temp->next; //Traverse to the end of the list
        }
        temp->next = newTask; //Add new task at the end of the list
    }
    cout<<"Task added: ["<<newTask->taskId<<"] "<<taskDesc<<endl;
}

void findMiddleTask() const
{
    if(head == nullptr)
    {
        cout<<"No tasks to find middle of the list"<<endl;
        return;
    }
    //First pass: Count the total number of tasks
    int count = 0;
    Task* temp = head;
    while(temp != nullptr)
    {
        count++;
        temp = temp->next;
    }
    //Second pass: Find middle index
    int middleIdx = count/2;
    //Third pass: Traverse to the middle
    temp = head;
    for(int i=0; i< middleIdx; i++)
    {
        temp = temp->next;
    }
    //Print middle task
```

```
    cout<<"Middle task["<<temp->taskId<<"] "<<temp->description<<endl;
}
//Display all tasks
void displayTasks() const
{
    if(head == nullptr)
    {
        cout<<"No tasks to display"<<endl;
        return;
    }
    cout<<"Current tasks: "<<endl;
    Task* temp = head;
    while(temp != nullptr)
    {
        cout<<"["<<temp->taskId<<"] "<<temp->description<<endl;
        temp = temp->next;
    }
}

void swapByAddr(Task* task1, Task* task2)
{
    if(!task1 || !task2 || task1 == task2)
    {
        cout<<"Invalid operation"<<endl;
        return;
    }
    //find prev nodes of task1 and task2
    Task* prev1 = nullptr;
    Task* prev2 = nullptr;
    Task* current = head;
    while(current && current->next)
    {
        if(current->next == task1)
            prev1 = current;
        if(current->next == task2)
            prev2 = current;
        current = current->next;
    }
```

```
    //swap the tasks
    if(prev1)
        prev1->next = task2;
    if(prev2)
        prev2->next = task1;

    Task* temp = task1->next;
    task1->next = task2->next;
    task2->next = temp;
    //if head node is task1
    if(head == task1)
    {
        head = task2;
    }
    else if(head == task2)
    {
        head = task1;
    }
    cout<<"Swapped "<<task1->taskId<<" <-> "<<task2->taskId<<endl;
}

//Task by id
Task* findTaskById(int id)
{
    Task* current = head;
    while(current)
    {
        if(current->taskId == id)
        {
            return current;
        }
        current = current->next;
    }
    return nullptr;
}
//Destructor
~TaskManager()
{
```

```
        while(head != nullptr)
        {
            Task* temp = head;
            head = head->next;
            delete temp;
        }
        cout<<"All tasks cleared"<<endl;
    }
};

int main()
{
    TaskManager taskMgr;
    taskMgr.addTask("Complete reverese linked. list by groups task issue");
    taskMgr.addTask("Create taskmanager application using linked list");
    taskMgr.addTask("add tasks to task manager");
    taskMgr.addTask("Swap tasks by address");
    taskMgr.addTask("find task id");

    Task* tsk1 = taskMgr.findTaskById(3);
    Task* tsk2 = taskMgr.findTaskById(1);
    if(tsk1 && tsk2)
    {
        taskMgr.swapByAddr(tsk1, tsk2);
        taskMgr.displayTasks();
    }

    //taskMgr.displayTasks();
    //taskMgr.findMiddleTask();

    return 0;
}
```

Doubly Linked List

```
//Cyclic doubly linked list
//Detect if the doubly linked list is cyclic and remove the cycle
// Online C++ compiler to run C++ program online
#include <iostream>
```

```
using namespace std;
class Node
{
   public:
   int data;
   Node* prev;
   Node* next;
   Node(int value):data(value), prev(nullptr), next(nullptr){}
};

class DoubleLinkedList
{
   private:
   Node* head;
   public:
   DoubleLinkedList():head(nullptr){}
   void append(int value)
   {
      //create new node and set it as head
      Node* newNode =new Node(value);
      if(!head)
      {
         head = newNode;
      }
      else
      {
         Node* temp = head;
         while(temp->next)
         {
            temp = temp->next;
         }
         temp->next = newNode;
         newNode->prev = temp;
      }
   }

   void createCycle(int pos)
   {
```

```
    if(!head) return;
    Node* temp = head;
    Node* cycleNode = nullptr;
    int count = 1;

    while(temp->next)
    {
       if(count == pos)
       {
          cycleNode = temp; //mark node where cycle starts
          cout<<"Cycle data "<<temp->data;
       }
       temp = temp->next;
       count++;
    }
    //Link last node to cycle node
    if(cycleNode)
    {
       temp->next = cycleNode;
       cycleNode->prev = temp;
    }
 }
 //determine if the list is cyclic
 bool hasCycle()
 {
    if(!head) return false;
    Node* slow = head;
    Node* fast = head;
    while(fast && fast->next)
    {
       slow = slow->next; //move by one step
       fast = fast->next->next;//move by two steps
       if(slow == fast) //cycle detected
       {
          cout<<"Cycle detected";
          return true;
       }
    }
```

```
    cout<<endl;
    return false;
}

void removeCycle()
{
    if(!hasCycle())
    {
        cout<<"No cycle detected"<<endl;
        return;
    }
    Node* slow = head;
    Node* fast = head;

    //1 - detect cycle
    while(fast && fast->next)
    {
        slow = slow->next;
        fast = fast->next->next;
        if(slow == fast)
        {
            break;
        }
    }
    //2 - find the start of the cycle
    slow = head;
    while(slow != fast)
    {
        slow = slow->next;
        fast = fast->next;
    }
    //3-find node before start of the cycle
    Node* cycleStart = slow;
    Node* cycleEnd = cycleStart;
    while(cycleEnd->next != cycleStart)
    {
        cycleEnd = cycleEnd->next;
    }
```

```
        cycleEnd->next = nullptr;
        cout<<"Cycle removed"<<endl;
    }
    void display()
    {
        if(!head)
        {
            cout<<"Empty list"<<endl;
            return;
        }
        Node* temp = head;
        while(temp)
        {
            cout<<temp->data <<" ";
            temp = temp->next;
        }
        cout<<endl;
    }
    ~DoubleLinkedList()
    {
        while(head)
        {
            Node* temp = head;
            head = head->next;
            delete temp;
        }
    }
};

int main() {
    DoubleLinkedList dlist;
    for(int i = 0; i< 10; i++)
        dlist.append((i+1)*10);
    cout<<"Original list"<<endl;
    dlist.display();
    dlist.createCycle(4);
    cout<<"Checking for cycle"<<endl;
    if(dlist.hasCycle())
```

```
    {
        cout<<"List contains cycle"<<endl;
        dlist.removeCycle();
    }
    else
    {
        cout<<"No cycle detected"<<endl;
    }

    dlist.display();
    return 0;
}
```

Circular Linked List

Example: Insert at begin, inset at end, insert at specific position, remove from begin, remove at end, remove by value

```
// Online C++ compiler to run C++ program online
#include <iostream>
using namespace std;

class Node
{
    public:
    int data;
    Node* next;
    Node(int value):data(value), next(nullptr){}
};

class CircularLinkedList
{
    private:
    Node* head;
    public:
    CircularLinkedList():head(nullptr){}

    //create newnode with the given value
    //if list is empty, set head pointer to newnode and link it to itself
```

```
   //else link newnode to head->next (current head), update head->next to new
node
   void insertAtBegin(int value)
   {
      Node* newNode = new Node(value);
      if(!head)
      {
         head = newNode;
         head->next = head;
      }
      else
      {
         newNode->next = head->next;
         head->next = newNode;
      }
      cout<<"Inserted "<<value<<" at the beginning"<<endl;
   }

   //Create new node with given value
   //if list is empty, set head pointer to newnode and link it to itself
   //else link newnode to head->next (current head)
   //update head->next to newnode
   //set head pointer to newnode
   void insertAtEnd(int value)
   {
      Node* newNode = new Node(value);
      if(!head)
      {
         head = newNode;
         head->next = head;
      }
      else
      {
         newNode->next = head->next;
         head->next = newNode;
         head = newNode;
      }
      cout<<"Inserted "<<value<<" at the end"<<endl;
```

```
}

//Edge case: position is 1, call insertatbegin
//if list is empty, ensure position is valid
//Traverse list to find the node at pos-1
//insert newnode between the found node and its next node
//if position is at the end, update head pointer
void insertAtPosition(int value, int pos)
{
   Node* newNode = new Node(value);
   if(!head)
   {
      //if list is empty; initilize the first node to position 1
      if(pos == 1)
      {
         head = newNode;
         head->next = head;
         cout<<"Inserted "<<value<<" at position "<<pos<<endl;
      }
      else
      {
         cout<<"Invalid position"<<endl;
      }
      return;
   }
   if(pos == 1)
   {
      insertAtBegin(value);
      return;
   }
   Node* current = head->next;
   int count = 1;
   while(count < pos-1 && current != head)
   {
      current = current->next;
      count++;
   }
   if(count == pos-1)
```

```
    {
        newNode->next = current->next;
        current->next = newNode;
        if(current == head)
        {
            head = newNode;
        }
        cout<<"Inserted "<<value<<" at position "<<pos<<endl;
    }
    else
    {
        cout<<"Invalid: position exceeds list size"<<endl;
    }
}

void insert(int value)
{
    Node* newNode = new Node(value);
    if(!head)
    {
        head = newNode;
        newNode->next = head; //point to itself to make it circular
    }
    else
    {
        Node* temp = head;
        while(temp->next != head) //traverse to last node
        {
            temp = temp->next;
        }
        temp->next = newNode;
        newNode->next = head; //complete the circle
    }
}
//remove by value
void remove(int value)
{
    if(!head)
```

```
{
   cout<<"Cannot delete. Empty list"<<endl;
   return;
}
Node* current = head;
Node* prev = nullptr;

//find the node to delete
do
{
   //Single node in the list
   if(current->data == value)
   {
      if(current == head && current->next == head)
      {
         delete current;
         head = nullptr;
      }
      else if(current == head) //delete head
      {
         Node* temp = head;
         while(!temp->next) //find the last node
         {
            temp = temp->next;
         }
         temp->next = head->next;
         head = head->next;
         delete current;
      }
      else
      {
         prev->next = current->next;
         delete current;
      }
      cout<<"Node with value "<<value<<" deleted"<<endl;
      return;
   }
   prev = current;
```

```
            current = current->next;
        }while(current != head);
        cout<<"Node with value "<<value<<" not found"<<endl;
    }
    //remove at begin
    //check if list is empty. if list has only one node then delete node and set head
to nullptr
    //else update head->next pointer to skip to current head
    //delete current head
    void removeAtBegin()
    {
        if(!head)
        {
            cout<<"Empty list"<<endl;
            return;
        }
        Node* current = head->next;
        if(head == current)
        {
            //One node in the list
            delete current;
            head = nullptr;
        }
        else
        {
            head->next = current->next; //update head->next to skip to current
            delete current; //delete old head
        }
        cout<<"Removed node at beginning"<<endl;
    }
    //remove node at end
    //check if list is empty
    //If only one node, then delete node and set head pointer to null
    //else traverse to last but one node. update the next pointer of the last but one
node to point to the current head. delete old head. update head pointer
    void removeAtEnd()
    {
        if(!head)
```

```
    {
        cout<<"Empty list"<<endl;
        return;
    }
    Node* current = head->next;
    if(head == current) //one node in the list
    {
        delete head;
        head = nullptr;
    }
    else
    {
        Node* temp = head->next;
        while(temp->next != head)
        {
            temp = temp->next;
        }
        //delete temp->next; //delete old head
        //temp points to last but one node
        temp->next = head->next;
        delete head;
        head = temp; //update head to new last node
    }
    cout<<"Removed node at end"<<endl;
}

void display()
{
    if(!head)
    {
        cout<<"Empty list"<<endl;
        return;
    }
    Node* temp = head->next;
    do
    {
        cout<<temp->data<<" ";
        temp = temp->next;
```

```
        }while(temp!= head->next);//stop when circle is completed
        cout<<endl;
    }

    ~CircularLinkedList()
    {
        if(!head)
            return;
        Node* temp = head;
        while(head->next != temp)
        {
            Node* current = head;
            head = head->next;
            delete current;
        }
        //delete last node
        delete head;
        head = nullptr;
    }
};

int main() {
    CircularLinkedList clist;
    clist.insertAtEnd(10);
    clist.insertAtEnd(30);
    clist.insertAtEnd(40);
    clist.display();
    clist.insertAtBegin(20);
    clist.insertAtBegin(50);
    clist.display();
    clist.insertAtPosition(25, 4);
    clist.display();
    clist.insertAtPosition(35, 1);
    clist.display();
    clist.removeAtBegin();
    clist.display();
    clist.removeAtEnd();
    clist.display();
```

```
    clist.remove(25);
    clist.display();
    return 0;
}
```

Doubly Circular Linked List

```
// Online C++ compiler to run C++ program online
#include <iostream>
using namespace std;

// Node structure
class Node
{
    public:
    int data;
    Node* next;
    Node* prev;

    Node(int value) : data(value), next(nullptr), prev(nullptr) {}
};

// Doubly Circular Linked List class
class DoublyCircularLinkedList
{
    private:
    Node* head;

    public:
    // Constructor
    DoublyCircularLinkedList() : head(nullptr) {}

    // Insert at the end
    void insertEnd(int value)
    {
        Node* newNode = new Node(value);
        if (!head)
        { // If the list is empty
            head = newNode;
```

```
        head->next = head;
        head->prev = head;
    }
    else
    {
        Node* tail = head->prev;
        tail->next = newNode;
        newNode->prev = tail;
        newNode->next = head;
        head->prev = newNode;
    }
}

// Insert at the beginning
void insertBegin(int value)
{
    Node* newNode = new Node(value);
    if (!head)
    { // If the list is empty
        head = newNode;
        head->next = head;
        head->prev = head;
    }
    else
    {
        Node* tail = head->prev;
        newNode->next = head;
        newNode->prev = tail;
        tail->next = newNode;
        head->prev = newNode;
        head = newNode;
    }
}

// Delete a node with a given value
void deleteNode(int value)
{
    if (!head)
```

```
{ // If the list is empty
   cout << "List is empty!" << endl;
   return;
}

Node* current = head;
Node* previous = nullptr;

// Search for the node to be deleted
do {
   if (current->data == value)
   {
      if (current == head && current->next == head)
      { // Only one node
         head = nullptr;
      }
      else if (current == head)
      { // Deleting the head node
         Node* tail = head->prev;
         head = head->next;
         tail->next = head;
         head->prev = tail;
      }
      else if (current->next == head)
      { // Deleting the tail node
         Node* tail = current->prev;
         tail->next = head;
         head->prev = tail;
      }
      else
      { // Deleting a middle node
         Node* previous = current->prev;
         Node* next = current->next;
         previous->next = next;
         next->prev = previous;
      }
      delete current;
      return;
```

```
        }
        previous = current;
        current = current->next;
    } while (current != head);

    cout << "Value not found in the list!" << endl;
}

// Display the list
void display()
{
    if (!head)
    {
        cout << "List is empty!" << endl;
        return;
    }

    Node* current = head;
    cout << "Doubly Circular Linked List: ";
    do
    {
        cout << current->data << " ";
        current = current->next;
    } while (current != head);
    cout << endl;
}

// Display the list in reverse
void displayReverse()
{
    if (!head)
    {
        cout << "List is empty!" << endl;
        return;
    }

    Node* tail = head->prev;
    Node* current = tail;
```

```
        cout << "Reverse Doubly Circular Linked List: ";
        do {
            cout << current->data << " ";
            current = current->prev;
        } while (current != tail);
        cout << endl;
    }

    // Destructor to free memory
    ~DoublyCircularLinkedList()
    {
        if (!head) return;

        Node* current = head;
        do {
            Node* temp = current;
            current = current->next;
            delete temp;
        } while (current != head);

        head = nullptr;
    }
};

// Main function
int main()
{
    DoublyCircularLinkedList list;

    // Insert elements
    list.insertEnd(10);
    list.insertEnd(20);
    list.insertEnd(30);
    list.insertBegin(5);

    // Display the list
    list.display();
```

```
    // Display in reverse
    list.displayReverse();

    // Delete a node
    list.deleteNode(20);
    list.display();

    // Delete a non-existent node
    list.deleteNode(50);

    return 0;
}
```

Example: Traffic Control System to process vehicles at signal per cycle for the duration of the signal that it is active.
Vehicles are added to the queue to process then at intersection

```
// Online C++ compiler to run C++ program online
#include <iostream>
#include <string>
using namespace std;

class VehicleNode
{
    public:
    string vehicle;
    VehicleNode* next;

    VehicleNode(string veh):vehicle(veh), next(nullptr){}
};

class TrafficSignal
{
    public:
    string signalColour;
    int duration;  //duration for which signal is active
    TrafficSignal* next, *prev;
    VehicleNode* vehQFront, *vehQRear;
```

```
  public:
  TrafficSignal(string colour, int dura): signalColour(colour), duration(dura),
next(nullptr), prev(nullptr), vehQFront(nullptr), vehQRear(nullptr){}

  //Add vehivle to the queue
  void addVehicle(string veh)
  {
    VehicleNode* newVeh = new VehicleNode(veh);
    if(vehQRear == nullptr)
    {
        vehQFront = vehQRear = newVeh; //first vehicle in the queue
    }
    else
    {
      vehQRear->next = newVeh;
      vehQRear = newVeh;
    }
  }
  //process vehicles from the queue
  void processVeh(int maxVeh)
  {
    for(int i=0; i< maxVeh && vehQFront != nullptr; i++)
    {
      cout<<" - "<<vehQFront->vehicle <<" passed "<<endl;
      VehicleNode* temp = vehQFront;
      vehQFront = vehQFront->next;
      delete temp;
    }
    //if no vehicles then reset
    if(vehQFront == nullptr)
    {
      vehQRear = nullptr;
    }
  }

  void displayVeh()
  {
    if(vehQFront == nullptr)
```

```
        {
            cout<<"No vehicles in queue"<<endl;
            return;
        }
        VehicleNode* current = vehQFront;
        while(current != nullptr)
        {
            cout<<current->vehicle<<" ";
            current = current->next;
        }
    }
};

class TrafficControl
{
    public:
    TrafficSignal* head;
    public:
    TrafficControl() : head(nullptr){}

    //add signal to the list
    void addSignal(string colour, int duration)
    {
        TrafficSignal* newSignal = new TrafficSignal(colour, duration);
        if(head == nullptr)
        {
            //first signal in the list
            head = newSignal;
            head->next = head;
            head->prev = head;
        }
        else
        {
            //insert at the end of the list
            TrafficSignal* tail = head->prev;
            tail->next = newSignal;
            newSignal->prev = tail;
            newSignal->next = head;
```

```
        head->prev = newSignal;
    }
    cout<<"Added signal "<<colour<<" with duration "<<duration<< endl;
}

void addVehicle(string colour, string vehicle)
{
    TrafficSignal* current = head;
    do
    {
        if(current->signalColour == colour)
        {
            current->addVehicle(colour);
            cout<<"Vehicle "<<vehicle<<" added to signal "<<colour<<endl;
            return;
        }
        current = current->next;
    }while(current != head);
    cout<<"Signal colour "<<colour<<" not found"<<endl;
}

void manageTraffic(int cycle, int vehPerCycle)
{
    if(head == nullptr)
    {
        cout<<"No signal"<<endl;
        return;
    }
    TrafficSignal* current = head;
    for(int i=0; i<cycle; i++)
    {
        cout<<"Current signal "<<current->signalColour<<" duration ("<<current-
>duration<<" seconds)"<<endl;
        if(current->vehQFront != nullptr)
        {
            cout<<"Vehicles passing through "<<current->signalColour<<"
signal"<<endl;
            //process vehicles for the duration of the signal
```

```
            for(int j=0; j< current->duration; j++)
            {
                current->processVeh(vehPerCycle);
            }
        }
        else
        {
            cout<<"No vehicles at "<<current->signalColour<<" signal"<<endl;
        }
        cout<<endl;
        current = current->next;
    }
}

void displaySignal()
{
    if(head == nullptr)
        return;
    TrafficSignal* current = head;
    cout<<"Traffic Signal and Vehicles"<<endl;
    do
    {
        cout<<current->signalColour<<" ";
        current->displayVeh();
        cout<<endl;
        current = current->next;
    }while(current != head);
    cout<<endl;
}

~TrafficControl()
{
    if(head == nullptr)
        return;
    TrafficSignal* current = head;
    do
    {
        //clean up vehicle queue
```

```
        while(current->vehQFront != nullptr)
        {
            VehicleNode* temp = current->vehQFront;
            current->vehQFront = current->vehQFront->next;
            delete temp;
        }
        TrafficSignal* temp = current;
        current = current->next;
        delete temp;
    }while(current != head);
    head = nullptr;
  }
};

int main() {

  TrafficControl trfSystem;
  trfSystem.addSignal("Red", 3);
  trfSystem.addSignal("Green", 5);
  trfSystem.addSignal("Yellow", 2);

  trfSystem.addVehicle("Red", "Car1");
  trfSystem.addVehicle("Red", "Bus1");
  trfSystem.addVehicle("Red", "Bike1");
  trfSystem.addVehicle("Yellow", "Car2");
  trfSystem.addVehicle("Green", "Bus2");
  trfSystem.addVehicle("Green", "Truck1");
  trfSystem.addVehicle("Green", "Bike2");
  trfSystem.displaySignal();

  cout<<"Simulating Traffic Flow"<<endl;
  trfSystem.manageTraffic(6, 2); //6 cycles, 2 vehicles processed per cycle
  trfSystem.displaySignal();
  return 0;
}
```

Stack

A **stack** is a linear data structure that follows the **Last In, First Out (LIFO)** principle, meaning the last element added to the stack is the first one to be removed.

Key Stack Operations

1. **Push**: Add an element to the top of the stack.
2. **Pop**: Remove the top element from the stack.
3. **Peek (or Top)**: Retrieve the top element without removing it.
4. **IsEmpty**: Check if the stack is empty.
5. **Size**: Get the number of elements in the stack.

Stack Implementation in C++

You can implement a stack in multiple ways:

1. Using the **Standard Template Library (STL)**.
2. Using an **array** or a **linked list**.

Example: Stack implementation using array

```
// Online C++ compiler to run C++ program online
#include <iostream>
using namespace std;

class Stack
{
   private:
   int* arr; // array to hold the stack elements
   int top; // index of the top element
   int capacity; // maximum size of the stack

   public:
   Stack(int size)
   {
      arr = new int(size);
      capacity = size;
      top = -1; //empty stack initially
   }

   bool isFull()
   {
```

```
    return top==capacity-1;
}

bool isEmpty()
{
    return top == -1;
}

void push(int value)
{
    if(isFull())
    {
        cout<<"Stack overflow. Cannot push "<<value<<endl;
        return;
    }
    arr[++top] = value;
    cout<<"Pushed "<<value<<" onto stack"<<endl;
}

int pop()
{
    if(isEmpty())
    {
        cout<<"Stack underflow. No elements to pop"<<endl;
        return -1; //return invalid value for error
    }
    cout<<"Popped "<<arr[top]<<" from the stack"<<endl;
    return arr[top--];
}

int peek()
{
    if(isEmpty())
    {
        cout<<"Empty stack. No elements to peek"<<endl;
        return -1;
    }
    return arr[top];
```

```
    }

    int size()
    {
        return top+1;
    }

    ~Stack()
    {
        delete[] arr;
    }
};

int main() {
    Stack stack(5);
    for(int i=0; i<5; i++)
    {
        stack.push((i+1)*10);
    }
    stack.push(100);

    cout<<"Top element of stack "<<stack.peek()<<endl;
    for(int i=0; i<=5; i++)
    {
        stack.pop();
    }

    return 0;
}
```

Example: Stack implementation using linked list

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

struct Node {
    int data;
    Node* next;
```

```
    Node(int value) : data(value), next(nullptr) {}
};

class Stack {
private:
    Node* top;

public:
    Stack() : top(nullptr) {}

    // Push an element onto the stack
    void push(int value) {
        Node* newNode = new Node(value);
        newNode->next = top;
        top = newNode;
    }

    // Pop an element from the stack
    void pop() {
        if (!top) {
            cout << "Stack underflow. Cannot pop." << endl;
            return;
        }
        Node* temp = top;
        top = top->next;
        delete temp;
    }

    // Get the top element
    int peek() {
        if (!top) {
            cout << "Stack is empty." << endl;
            return -1;
        }
        return top->data;
    }

    // Check if the stack is empty
```

```
    bool isEmpty() {
        return top == nullptr;
    }

    // Destructor to free memory
    ~Stack() {
        while (top) {
            pop();
        }
    }
};

int main() {
    Stack myStack;

    myStack.push(10);
    myStack.push(20);
    myStack.push(30);

    cout << "Top element: " << myStack.peek() << endl;

    myStack.pop();
    cout << "Top element after pop: " << myStack.peek() << endl;

    if (myStack.isEmpty()) {
        cout << "The stack is empty." << endl;
    } else {
        cout << "The stack is not empty." << endl;
    }

    return 0;
}
```

Example: Infix expression evaluation and output the result

```
// Online C++ compiler to run C++ program online
#include <iostream>
#include <stack>
#include <string>
```

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

class InfixEval
{
   public:

//Function to perform arithmetic operations
   int performOpr(int val1, int val2, char opr)
   {
      switch(opr)
      {
         case '+': return val1 + val2;
         case '-': return val1 - val2;
         case '*': return val1 * val2;
         case '/':
         if(val2 != 0)
         {
            return val1 / val2;
         }
         case '^': return pow(val1, val2);
         default: return 0;
      }
   }

//Function to determine operator precedence
   int oprPrecedence(char opr)
   {
      if(opr == '*' || opr == '-')
      {
         return 1;
      }
      if(opr == '*' || opr == '/')
      {
         return 2;
      }
      if(opr == '^')
         return 3;
```

```
    return 0;
  }

//Function to evaluate the top of the stack
  void evaluateTop(stack<int>& operand , stack<char>& operators)
  {
    int val2 = operand.top();
    operand.pop();
    int val1 = operand.top();
    operand.pop();
    char op = operators.top();
    operators.pop();
    operand.push(performOpr(val1, val2, op));
  }

//Function to evaluate infix expression
  int evaluateInfix(const string& expr)
  {
    stack<int> operand; //stack for numbers
    stack<char> operators; //stack for operators

    for(size_t i=0; i< expr.length(); i++)
    {
      char ch = expr[i];

      //if character is a digit, extract the number
      if(isdigit(ch))
      {
        int num = 0;
        while (i < expr.length() && isdigit(expr[i]))
        {
          num = num *10 +(expr[i] - '0');
          i++;
        }
        operand.push(num);
        i--; //adjust index
      }
      else if(ch == '('
```

```
        {
            //If the character is '(', push it to operators stack
            operators.push(ch);
        }
        else if(ch == ')') //if char is ')', push it to operators stack
        {
            while(!operators.empty() && operators.top() != '(')
            {
                evaluateTop(operand, operators);
            }
            operators.pop(); // remove '('
        }
        else if(ch == '+' || ch == '-' || ch == '*' || ch == '/' || ch == '^') //if ch is an
operator
        {
            //ensure operator precedence rules are followed
            while(!operators.empty() && oprPrecedence(operators.top()) >=
oprPrecedence(ch))
            {
                evaluateTop(operand, operators);
            }
            operators.push(ch);
        }
    }

    //evaluate remaining operators in the stack
    while(!operators.empty())
    {
        evaluateTop(operand, operators);
    }
//result will be the only number left in the operand stack
    return operand.top();
  }

};

int main()
{
```

```
    InfixEval infix;
    string expr;
    cout<<"Enter an infix expression: ";
    cin>>expr;
    int res = infix.evaluateInfix(expr);
    cout<<"Result is: "<<res<<endl;

    return 0;
}
```

How the above example works

1. Operands:
   - Numbers are pushed onto the operands stack
   - Multidigit numbers are supported by iterating until the entire number is parsed
2. Operators:
   - Operators are pushed onto the operators stack
   - Precedence is enforced using oprPrecedence() function
3. Parenthesis:
   - When a closing parenthesis ‘)’ is encountered, the program evaluates operators within the parenthesis
4. Final evaluation: After parsing the input, any remaining operators are evaluated

Time Complexity:

- Parsing of expression: O(n)
- Evaluation of operators: O(n)
- Total: O(n) where ‘n’ is the length of the expression

Space Complexity:

- Stacks for operations and operands: O(n)

Queue

A **queue** is a linear data structure that follows the **First-In-First-Out (FIFO)** principle. This means the element that is added first is removed first. Think of a queue as a line of people waiting for service; the person at the front gets served first.

In C++, a **queue** is commonly implemented using:

- STL’s std::queue.

- Custom implementations using arrays or linked lists.

Key Operations on a Queue

1. **Enqueue (push)**: Add an element to the rear of the queue.
2. **Dequeue (pop)**: Remove an element from the front of the queue.
3. **Front**: Access the element at the front without removing it.
4. **Rear**: Access the element at the rear without removing it.
5. **Empty**: Check if the queue is empty.
6. **Size**: Get the number of elements in the queue.

Examples

- Queue using STL

```
#include <iostream>
#include <queue>  // For std::queue

using namespace std;

int main() {
   queue<int> q;

   // Enqueue elements
   q.push(10);
   q.push(20);
   q.push(30);

   // Display the front and rear elements
   cout << "Front element: " << q.front() << endl;  // 10
   cout << "Rear element: " << q.back() << endl;    // 30

   // Dequeue elements
   cout << "Dequeuing: " << q.front() << endl;
   q.pop();  // Removes 10

   cout << "Front element after dequeuing: " << q.front() << endl;  // 20

   // Size and empty status
   cout << "Queue size: " << q.size() << endl;  // 2
   cout << "Is queue empty? " << (q.empty() ? "Yes" : "No") << endl;  // No
```

```
    return 0;
}
```

- Custom queue implementation using arrays

```
#include <iostream>
#define MAX 100  // Maximum size of the queue

using namespace std;

class Queue {
private:
    int arr[MAX];  // Array to store queue elements
    int front;     // Index of the front element
    int rear;      // Index of the rear element
    int size;      // Current size of the queue

public:
    // Constructor
    Queue() : front(0), rear(-1), size(0) {}

    // Enqueue operation
    void enqueue(int value) {
        if (size == MAX) {
            cout << "Queue is full!" << endl;
            return;
        }
        rear = (rear + 1) % MAX;
        arr[rear] = value;
        size++;
    }

    // Dequeue operation
    void dequeue() {
        if (size == 0) {
            cout << "Queue is empty!" << endl;
            return;
        }
```

```
        front = (front + 1) % MAX;
        size--;
    }

    // Get the front element
    int getFront() const {
        if (size == 0) {
            cout << "Queue is empty!" << endl;
            return -1;
        }
        return arr[front];
    }

    // Get the rear element
    int getRear() const {
        if (size == 0) {
            cout << "Queue is empty!" << endl;
            return -1;
        }
        return arr[rear];
    }

    // Check if the queue is empty
    bool isEmpty() const {
        return size == 0;
    }

    // Get the size of the queue
    int getSize() const {
        return size;
    }
};

int main() {
    Queue q;

    // Enqueue elements
    q.enqueue(10);
```

```
    q.enqueue(20);
    q.enqueue(30);

    // Display the front and rear elements
    cout << "Front element: " << q.getFront() << endl;
    cout << "Rear element: " << q.getRear() << endl;

    // Dequeue elements
    cout << "Dequeuing..." << endl;
    q.dequeue();

    cout << "Front element after dequeuing: " << q.getFront() << endl;
    cout << "Queue size: " << q.getSize() << endl;

    return 0;
}
```

- Queue using linked list for dynamic sizing

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

// Node structure
struct Node {
    int data;
    Node* next;

    Node(int value) : data(value), next(nullptr) {}
};

// Queue class
class Queue {
private:
    Node* front;  // Pointer to the front of the queue
    Node* rear;   // Pointer to the rear of the queue

public:
    // Constructor
    Queue() : front(nullptr), rear(nullptr) {}
```

```
// Enqueue operation
void enqueue(int value) {
    Node* newNode = new Node(value);
    if (!rear) {  // If queue is empty
        front = rear = newNode;
    } else {
        rear->next = newNode;
        rear = newNode;
    }
}

// Dequeue operation
void dequeue() {
    if (!front) {
        cout << "Queue is empty!" << endl;
        return;
    }
    Node* temp = front;
    front = front->next;
    if (!front) {
        rear = nullptr;
    }
    delete temp;
}

// Get the front element
int getFront() const {
    if (!front) {
        cout << "Queue is empty!" << endl;
        return -1;
    }
    return front->data;
}

// Check if the queue is empty
bool isEmpty() const {
    return !front;
```

```
    }

    // Destructor to free memory
    ~Queue() {
        while (front) {
            dequeue();
        }
    }
};

int main() {
    Queue q;

    // Enqueue elements
    q.enqueue(10);
    q.enqueue(20);
    q.enqueue(30);

    // Display the front element
    cout << "Front element: " << q.getFront() << endl;

    // Dequeue elements
    cout << "Dequeuing..." << endl;
    q.dequeue();
    cout << "Front element after dequeuing: " << q.getFront() << endl;

    return 0;
}
```

Time and Space Complexity of Queue Operations

The **time complexity** and **space complexity** of queue operations depend on the type of implementation used. Let's analyse the complexities for various queue operations and implementations.

1. *Queue Operations*

Here are the primary operations of a queue and their complexities:

Operation	Description	Complexity (Array)	Complexity (Linked List)

Enqueue	Add an element at the rear	O(1)	O(1)
Dequeue	Remove an element from the front	O(1)	O(1)
Front	Access the front element	O(1)	O(1)
Rear	Access the rear element	O(1)	O(1)
IsEmpty	Check if the queue is empty	O(1)	O(1)

2. Space Complexity

The space complexity depends on the underlying data structure used:
Array-Based Queue

1. **Static Array**:
Space complexity is O(n), where n is the maximum size of the queue.
Requires a pre-allocated fixed size, which may lead to wasted space if the queue is not full.

2. Dynamic Array:
Space complexity is O(n), but the array can grow dynamically to accommodate more elements. Resizing involves O(n) operations during growth.

3. Linked List-Based Queue
Space complexity is O(n), where n is the number of elements in the queue.
Additional memory is required for storing the pointers (O(n)).

Detailed Comparison

1. Array-Based Queue

Advantages:
Simpler implementation.
Faster access to elements due to contiguous memory storage.
Disadvantages:
Fixed size in static arrays.

Dynamic resizing is computationally expensive.
Complexity:
All operations (enqueue, dequeue, front, rear, isEmpty, size) are O(1), except resizing during dynamic array expansion, which is O(n).

2. **Linked List-Based Queue**

Advantages:
No fixed size; grows dynamically.
No resizing overhead.
Disadvantages:
Slightly slower due to pointer dereferencing.
Extra space required for storing pointers.
Complexity:
All operations are O(1).

3. Circular Queue

A circular queue is a variation of an array-based queue where the last position connects to the first, optimizing space usage.
Time Complexity: Enqueue, Dequeue, Front, Rear, IsEmpty, and Size are all O(1).
Space Complexity: O(n), where n is the size of the array.

Real-World Queue Usage

The choice of implementation depends on the application:

1. **Array-Based Queue**: Suitable when the maximum size is known and small, e.g., buffers in embedded systems.
2. **Linked List-Based Queue**: Ideal for scenarios with an unpredictable number of elements, e.g., task scheduling.
3. **Circular Queue**: Efficient for cyclic operations, e.g., CPU scheduling using round-robin or fixed-size buffers.

By analysing the time and space complexities, you can select the most appropriate queue implementation for your specific needs.