The C++

Outline

- Aggregation **Conti...**
- Abstraction
- Friend Class
- Memory Management **Conti...**
- Recursion
- File Streaming
- Exceptional Handling
- Polymorphism
- Data Structure
- Algorithm

Aggregation coni...

```
struct Node
{
    int data;
    int color;
    Node  * left,
          * right,
          * parent;
…
```

```
class rgtree
{
private:
    Node* root;
…
```

```
class set {
    rgtree* rbTree;
…
```

Abstraction

Conceptualization: Abstracting focuses on essential features while hiding unnecessary details.

Data Hiding: It bundles data and functions, hiding implementation details.

Modularity: Breaks down complex systems into manageable units.

Generalization: Defines general concepts for reuse in different scenarios.

Interface-based Programming: Interfaces specify behavior without revealing internals.

Reduced Complexity: Simplifies systems by hiding details and providing clear boundaries.

Adaptability: Enables easier maintenance and enhancements by decoupling components.

Abstraction

Class rgtree	Class set

protected:
// Rotate the subtree rooted at the given node to the left
void rotateLeft(Node*&);
 	
// Rotate the subtree rooted at the given node to the right
void rotateRight(Node*&);
 	
…
	private:
	void intersectHelper(Node* root, rgtree* otherTree, set& result) const;
	void differenceHelper(Node* root, rgtree* otherTree, set& result) const;
	void printHelper(Node* root) const;
	
	…

Friend Class

A friend class in C++ is a class that is granted access to the private and protected members of another class.

When a class declares another class as a friend, it allows the friend class to access its private and protected members as if they were its own members.

The friend relationship is unidirectional, meaning that if class A declares class B as a friend, class B can access the private and protected members of class A, but not vice versa.

Friend classes are often used when a class needs to provide privileged access to its internals to another closely related class without exposing those internals to the public.

The friend relationship is declared using the friend keyword within the class declaration of the class that wishes to provide access to its members.

Friend Class

```
public:
    friend class set;
...
```

Memory Management Conti...

```
// Constructor
set::set() {
    rbTree = new rgtree(); // Initialize the Red Black Tree
}
// Destructor
set::~set() {
    delete rbTree; // Free the memory allocated for the Red Black Tree
}
```

Recursion

Recursion in programming refers to the technique where a function calls itself directly or indirectly in order to solve a problem.

Base Case: A terminating condition that specifies when the recursion should stop. Without a base case, the function would keep calling itself indefinitely, leading to stack overflow.

Recursive Case: A part of the function that calls itself with a modified version of the original problem. Each recursive call typically moves closer to the base case, ensuring termination.

Recursion

```
// Function to perform preorder traversal of BST
void rgtree::preorderBST(Node*& ptr) {
    if (ptr == nullptr)
        return;
    cout << ptr->data << " " << ptr->color << endl;
    preorderBST(ptr->left);
    preorderBST(ptr->right);
}
```

File Streaming

Header Inclusion: Begin by including the necessary header file <fstream> for file streaming operations.

For reading from a file, use **ifstream**. For writing to a file, use **ofstream**.

Opening a File: Use the open() method to open a file with the desired filename and mode (input, output, or both).

Reading from a File: Use >> operator to extract data from the input file stream. Use getline() function to read lines from the file.

Writing to a File: Use << operator to insert data into the output file stream.

Closing the File: Always close the file using the close() method when done with file operations to release system resources.

File Streaming

```
void loadData(const std::string& filename, std::vector<std::vector<float>>& X_out,
std::vector<float>& y_out) {

    std::ifstream file(filename);

    if (!file.is_open()) {

        throw std::runtime_error("Error opening file");

    }

...
```

Exceptional Handling

Exception handling in C++ provides a mechanism to deal with runtime errors, exceptions, and abnormal situations that may occur during program execution.

try: The try block encloses the code that may potentially throw an exception. It is followed by one or more catch blocks.

throw: The throw statement is used to explicitly raise an exception. It can be used within a function to signal an error condition.

catch: The catch block follows the try block and is used to handle exceptions. It specifies the type of exception it can handle and contains code to deal with the exception.

Exceptional Handling

```
try {
      ...
    utility::loadData("KNN.csv", X, y);
      ....
catch (const std::exception& e) {
    std::cerr << "Error: " << e.what() << std::endl;
    return 1;
}
```

Polymorphism

Compile-Time Polymorphism (Static Polymorphism): Utilizes function overloading and operator overloading, resolved by the compiler based on argument types.

Run-Time Polymorphism (Dynamic Polymorphism): Achieved via inheritance and virtual functions, resolved at runtime based on object types, typically utilizing pointers or references to base class objects.

Polymorphism :: Runtime

```
class Model {
public:
    virtual void fit(std::vector<std::vector<float>> X, std::vector<float> y) = 0;
    virtual float predict(const std::vector<float>& X) const = 0;
};
// Pure Virtual Functions
```

Data Structure

Doubly Linked List:

Efficient insertion and deletion (O(1)).

Simple traversal and dynamic size adjustment.

Red-Black Tree:

Balanced structure ensures efficient operations (O(log n)).

Automatically maintains sorted order and self-balances.

Set:

Stores unique elements efficiently.

Offers fast search operations (O(log n)) and standard library support.

Algorithm

Linear Regression:

Supervised learning for modeling the relationship between dependent and independent variables.

Assumes a linear relationship, making it suitable for numerical value prediction.

Aims to minimize error by finding the best-fitting line or hyperplane.

K-Nearest Neighbors (KNN):

Versatile supervised learning algorithm for classification and regression.

Prediction based on similarity with k nearest neighbors.

Non-parametric and lazy, storing entire dataset for computation at prediction time.

YouTube

https://youtu.be/rt4lJ7szwuA