OOP Doc

Infix to postfix:

```
// C++ code to convert infix expression to postfix

#include <bits/stdc++.h>
using namespace std;

// Function to return precedence of operators
int prec(char c)
{
  if (c == '^')
    return 3;
  else if (c == '/' || c == '*')
    return 2;
  else if (c == '+' || c == '-')
    return 1;
  else
    return -1;
}

// The main function to convert infix expression
// to postfix expression
void infixToPostfix(string s)
{

  stack<char> st;
  string result;

  for (int i = 0; i < s.length(); i++) {
    char c = s[i];

    // If the scanned character is
    // an operand, add it to output string.
    if ((c >= 'a' && c <= 'z') || (c >= 'A' && c <= 'Z')
      || (c >= '0' && c <= '9'))
      result += c;

    // If the scanned character is an
    // ‘(‘, push it to the stack.
    else if (c == '(')
      st.push('(');

    // If the scanned character is an ‘)’,
    // pop and add to output string from the stack
    // until an ‘(‘ is encountered.
    else if (c == ')') {
      while (st.top() != '(') {
```

```
        result += st.top();
        st.pop();
      }
      st.pop();
    }

    // If an operator is scanned
    else {
      while (!st.empty()
        && prec(s[i]) <= prec(st.top())) {
        result += st.top();
        st.pop();
      }
      st.push(c);
    }
  }

  // Pop all the remaining elements from the stack
  while (!st.empty()) {
    result += st.top();
    st.pop();
  }

  cout << result << endl;
}

// Driver code
int main()
{
  string exp = "a+b*(c^d-e)^(f+g*h)-i";

  // Function call
  infixToPostfix(exp);

  return 0;
}
```

AVL Tree:

```
// AVL tree implementation in C++

#include <iostream>
using namespace std;

class Node {
   public:
  int key;
  Node *left;
  Node *right;
```

```
  int height;
};

int max(int a, int b);

// Calculate height
int height(Node *N) {
  if (N == NULL)
    return 0;
  return N->height;
}

int max(int a, int b) {
  return (a > b) ? a : b;
}

// New node creation
Node *newNode(int key) {
  Node *node = new Node();
  node->key = key;
  node->left = NULL;
  node->right = NULL;
  node->height = 1;
  return (node);
}

// Rotate right
Node *rightRotate(Node *y) {
  Node *x = y->left;
  Node *T2 = x->right;
  x->right = y;
  y->left = T2;
  y->height = max(height(y->left),
          height(y->right)) +
        1;
  x->height = max(height(x->left),
          height(x->right)) +
        1;
  return x;
}

// Rotate left
Node *leftRotate(Node *x) {
  Node *y = x->right;
  Node *T2 = y->left;
  y->left = x;
  x->right = T2;
  x->height = max(height(x->left),
          height(x->right)) +
        1;
  y->height = max(height(y->left),
          height(y->right)) +
        1;
  return y;
```

```
}

// Get the balance factor of each node
int getBalanceFactor(Node *N) {
  if (N == NULL)
    return 0;
  return height(N->left) -
       height(N->right);
}

// Insert a node
Node *insertNode(Node *node, int key) {
  // Find the correct postion and insert the node
  if (node == NULL)
    return (newNode(key));
  if (key < node->key)
    node->left = insertNode(node->left, key);
  else if (key > node->key)
    node->right = insertNode(node->right, key);
  else
    return node;

  // Update the balance factor of each node and
  // balance the tree
  node->height = 1 + max(height(node->left),
               height(node->right));
  int balanceFactor = getBalanceFactor(node);
  if (balanceFactor > 1) {
    if (key < node->left->key) {
      return rightRotate(node);
    } else if (key > node->left->key) {
      node->left = leftRotate(node->left);
      return rightRotate(node);
    }
  }
  if (balanceFactor < -1) {
    if (key > node->right->key) {
      return leftRotate(node);
    } else if (key < node->right->key) {
      node->right = rightRotate(node->right);
      return leftRotate(node);
    }
  }
  return node;
}

// Node with minimum value
Node *nodeWithMimumValue(Node *node) {
  Node *current = node;
  while (current->left != NULL)
    current = current->left;
  return current;
}
```

```
// Delete a node
Node *deleteNode(Node *root, int key) {
  // Find the node and delete it
  if (root == NULL)
    return root;
  if (key < root->key)
    root->left = deleteNode(root->left, key);
  else if (key > root->key)
    root->right = deleteNode(root->right, key);
  else {
    if ((root->left == NULL) ||
      (root->right == NULL)) {
      Node *temp = root->left ? root->left : root->right;
      if (temp == NULL) {
        temp = root;
        root = NULL;
      } else
        *root = *temp;
      free(temp);
    } else {
      Node *temp = nodeWithMimumValue(root->right);
      root->key = temp->key;
      root->right = deleteNode(root->right,
                  temp->key);
    }
  }

  if (root == NULL)
    return root;

  // Update the balance factor of each node and
  // balance the tree
  root->height = 1 + max(height(root->left),
              height(root->right));
  int balanceFactor = getBalanceFactor(root);
  if (balanceFactor > 1) {
    if (getBalanceFactor(root->left) >= 0) {
      return rightRotate(root);
    } else {
      root->left = leftRotate(root->left);
      return rightRotate(root);
    }
  }
  if (balanceFactor < -1) {
    if (getBalanceFactor(root->right) <= 0) {
      return leftRotate(root);
    } else {
      root->right = rightRotate(root->right);
      return leftRotate(root);
    }
  }
  return root;
}
```

```
// Print the tree
void printTree(Node *root, string indent, bool last) {
  if (root != nullptr) {
    cout << indent;
    if (last) {
      cout << "R----";
      indent += "   ";
    } else {
      cout << "L----";
      indent += "|  ";
    }
    cout << root->key << endl;
    printTree(root->left, indent, false);
    printTree(root->right, indent, true);
  }
}

int main() {
  Node *root = NULL;
  root = insertNode(root, 33);
  root = insertNode(root, 13);
  root = insertNode(root, 53);
  root = insertNode(root, 9);
  root = insertNode(root, 21);
  root = insertNode(root, 61);
  root = insertNode(root, 8);
  root = insertNode(root, 11);
  printTree(root, "", true);
  root = deleteNode(root, 13);
  cout << "After deleting " << endl;
  printTree(root, "", true);
}
```

BST:

```

// Binary Search Tree operations in C++

#include <iostream>
using namespace std;

struct node {
  int key;
  struct node *left, *right;
};

// Create a node
struct node *newNode(int item) {
  struct node *temp = (struct node *)malloc(sizeof(struct node));
  temp->key = item;
  temp->left = temp->right = NULL;
  return temp;
```

```
}

// Inorder Traversal
void inorder(struct node *root) {
  if (root != NULL) {
    // Traverse left
    inorder(root->left);

    // Traverse root
    cout << root->key << " -> ";

    // Traverse right
    inorder(root->right);
  }
}

// Insert a node
struct node *insert(struct node *node, int key) {
  // Return a new node if the tree is empty
  if (node == NULL) return newNode(key);

  // Traverse to the right place and insert the node
  if (key < node->key)
    node->left = insert(node->left, key);
  else
    node->right = insert(node->right, key);

  return node;
}

// Find the inorder successor
struct node *minValueNode(struct node *node) {
  struct node *current = node;

  // Find the leftmost leaf
  while (current && current->left != NULL)
    current = current->left;

  return current;
}

// Deleting a node
struct node *deleteNode(struct node *root, int key) {
  // Return if the tree is empty
  if (root == NULL) return root;

  // Find the node to be deleted
  if (key < root->key)
    root->left = deleteNode(root->left, key);
  else if (key > root->key)
    root->right = deleteNode(root->right, key);
  else {
    // If the node is with only one child or no child
    if (root->left == NULL) {
```

```
      struct node *temp = root->right;
      free(root);
      return temp;
    } else if (root->right == NULL) {
      struct node *temp = root->left;
      free(root);
      return temp;
    }

    // If the node has two children
    struct node *temp = minValueNode(root->right);

    // Place the inorder successor in position of the node to be deleted
    root->key = temp->key;

    // Delete the inorder successor
    root->right = deleteNode(root->right, temp->key);
  }
  return root;
}

// Driver code
int main() {
  struct node *root = NULL;
  root = insert(root, 8);
  root = insert(root, 3);
  root = insert(root, 1);
  root = insert(root, 6);
  root = insert(root, 7);
  root = insert(root, 10);
  root = insert(root, 14);
  root = insert(root, 4);

  cout << "Inorder traversal: ";
  inorder(root);

  cout << "\nAfter deleting 10\n";
  root = deleteNode(root, 10);
  cout << "Inorder traversal: ";
  inorder(root);
}
```

BFS:

```
// BFS algorithm in C++

#include <iostream>
#include <list>

using namespace std;
```

```
class Graph {
  int numVertices;
  list<int>* adjLists;
  bool* visited;

   public:
  Graph(int vertices);
  void addEdge(int src, int dest);
  void BFS(int startVertex);
};

// Create a graph with given vertices,
// and maintain an adjacency list
Graph::Graph(int vertices) {
  numVertices = vertices;
  adjLists = new list<int>[vertices];
}

// Add edges to the graph
void Graph::addEdge(int src, int dest) {
  adjLists[src].push_back(dest);
  adjLists[dest].push_back(src);
}

// BFS algorithm
void Graph::BFS(int startVertex) {
  visited = new bool[numVertices];
  for (int i = 0; i < numVertices; i++)
    visited[i] = false;

  list<int> queue;

  visited[startVertex] = true;
  queue.push_back(startVertex);

  list<int>::iterator i;

  while (!queue.empty()) {
    int currVertex = queue.front();
    cout << "Visited " << currVertex << " ";
    queue.pop_front();

    for (i = adjLists[currVertex].begin(); i != adjLists[currVertex].end(); ++i) {
      int adjVertex = *i;
      if (!visited[adjVertex]) {
        visited[adjVertex] = true;
        queue.push_back(adjVertex);
      }
    }
  }
}
```

DFS:

```
#include <vector>
#include<queue>
using namespace std;
class Graph
{
private:
  vector<int> vertices;
  vector<vector<int>> adjList;
public:
  void input();
  void output();
  void bfs();

};

void Graph::input()
{
  int n;
  cout << "Nhap so dinh: ";
  cin >> n;
  for (int i = 0; i < n; i++) {
    cout << "Nhap so canh ke voi dinh " << i << ": ";
    int numOfEdges;
    cin >> numOfEdges;
    vertices.push_back(i);
    vector<int> temp;
    for (int j = 0; j < numOfEdges; j++) {
      int n;
      cin >> n;
      temp.push_back(n);

    }
    adjList.push_back(temp);
  }
}

void Graph::output()
{
  cout << "Ma tran ke:\n";
  for (int i = 0; i < adjList.size(); i++) {
    cout << i << " | ";
    for (int j = 0; j < adjList[i].size(); j++) {
      cout << adjList[i][j] << " ";
    }
    cout << endl;
  }
}

void Graph::bfs() {
  int start;
```

```
  cout << "Nhap diem bat dau: ";
  cin >> start;
  vector<int> mark(vertices.size());
  queue<int> q;
  q.push(start);
  while (!q.empty()) {
    int y = q.front();
    q.pop();
    mark[y] = 1;
    cout << y << " ";
    for (int z : adjList[y]) {
      if (!mark[z]) {
        q.push(z);
      }
    }
  }
}
```

Kruskal Algorithm:

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

#define edge pair<int, int>

class Graph {
   private:
  vector<pair<int, edge> > G;  // graph
  vector<pair<int, edge> > T;  // mst
  int *parent;
  int V;  // number of vertices/nodes in graph
   public:
  Graph(int V);
  void AddWeightedEdge(int u, int v, int w);
  int find_set(int i);
  void union_set(int u, int v);
  void kruskal();
  void print();
};
Graph::Graph(int V) {
  parent = new int[V];

  //i 0 1 2 3 4 5
  //parent[i] 0 1 2 3 4 5
  for (int i = 0; i < V; i++)
    parent[i] = i;

  G.clear();
  T.clear();
```

```
}
void Graph::AddWeightedEdge(int u, int v, int w) {
  G.push_back(make_pair(w, edge(u, v)));
}
int Graph::find_set(int i) {
  // If i is the parent of itself
  if (i == parent[i])
    return i;
  else
    // Else if i is not the parent of itself
    // Then i is not the representative of his set,
    // so we recursively call Find on its parent
    return find_set(parent[i]);
}

void Graph::union_set(int u, int v) {
  parent[u] = parent[v];
}
void Graph::kruskal() {
  int i, uRep, vRep;
  sort(G.begin(), G.end());  // increasing weight
  for (i = 0; i < G.size(); i++) {
    uRep = find_set(G[i].second.first);
    vRep = find_set(G[i].second.second);
    if (uRep != vRep) {
      T.push_back(G[i]);  // add to tree
      union_set(uRep, vRep);
    }
  }
}
void Graph::print() {
  cout << "Edge :"
     << " Weight" << endl;
  for (int i = 0; i < T.size(); i++) {
    cout << T[i].second.first << " - " << T[i].second.second << " : "
       << T[i].first;
    cout << endl;
  }
}
```

Prim Algorithm:

```
// Prim's Algorithm in C++

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

#define INF 9999999

// number of vertices in grapj
```

```
#define V 5

// create a 2d array of size 5x5
//for adjacency matrix to represent graph

int G[V][V] = {
  {0, 9, 75, 0, 0},
  {9, 0, 95, 19, 42},
  {75, 95, 0, 51, 66},
  {0, 19, 51, 0, 31},
  {0, 42, 66, 31, 0}};

int main() {
  int no_edge;  // number of edge

  // create a array to track selected vertex
  // selected will become true otherwise false
  int selected[V];

  // set selected false initially
  memset(selected, false, sizeof(selected));

  // set number of edge to 0
  no_edge = 0;

  // the number of egde in minimum spanning tree will be
  // always less than (V -1), where V is number of vertices in
  //graph

  // choose 0th vertex and make it true
  selected[0] = true;

  int x;  //  row number
  int y;  //  col number

  // print for edge and weight
  cout << "Edge"
     << " : "
     << "Weight";
  cout << endl;
  while (no_edge < V - 1) {
    //For every vertex in the set S, find the all adjacent vertices
    // , calculate the distance from the vertex selected at step 1.
    // if the vertex is already in the set S, discard it otherwise
    //choose another vertex nearest to selected vertex  at step 1.

    int min = INF;
    x = 0;
    y = 0;

    for (int i = 0; i < V; i++) {
      if (selected[i]) {
        for (int j = 0; j < V; j++) {
          if (!selected[j] && G[i][j]) {  // not in selected and there is an edge
```

```
            if (min > G[i][j]) {
              min = G[i][j];
              x = i;
              y = j;
            }
          }
        }
      }
    }
    cout << x << " - " << y << " :  " << G[x][y];
    cout << endl;
    selected[y] = true;
    no_edge++;
  }

  return 0;
}
```

Dijkstra Algorithm:

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

#define INT_MAX 10000000

using namespace std;

void DijkstrasTest();

int main() {
  DijkstrasTest();
  return 0;
}

class Node;
class Edge;

void Dijkstras();
vector<Node*>* AdjacentRemainingNodes(Node* node);
Node* ExtractSmallest(vector<Node*>& nodes);
int Distance(Node* node1, Node* node2);
bool Contains(vector<Node*>& nodes, Node* node);
void PrintShortestRouteTo(Node* destination);

vector<Node*> nodes;
vector<Edge*> edges;

class Node {
   public:
  Node(char id)
    : id(id), previous(NULL), distanceFromStart(INT_MAX) {
```

```
    nodes.push_back(this);
  }

   public:
  char id;
  Node* previous;
  int distanceFromStart;
};

class Edge {
   public:
  Edge(Node* node1, Node* node2, int distance)
    : node1(node1), node2(node2), distance(distance) {
    edges.push_back(this);
  }
  bool Connects(Node* node1, Node* node2) {
    return (
      (node1 == this->node1 &&
       node2 == this->node2) ||
      (node1 == this->node2 &&
       node2 == this->node1));
  }

   public:
  Node* node1;
  Node* node2;
  int distance;
};

//////////////////
void DijkstrasTest() {
  Node* a = new Node('a');
  Node* b = new Node('b');
  Node* c = new Node('c');
  Node* d = new Node('d');
  Node* e = new Node('e');
  Node* f = new Node('f');
  Node* g = new Node('g');

  Edge* e1 = new Edge(a, c, 1);
  Edge* e2 = new Edge(a, d, 2);
  Edge* e3 = new Edge(b, c, 2);
  Edge* e4 = new Edge(c, d, 1);
  Edge* e5 = new Edge(b, f, 3);
  Edge* e6 = new Edge(c, e, 3);
  Edge* e7 = new Edge(e, f, 2);
  Edge* e8 = new Edge(d, g, 1);
  Edge* e9 = new Edge(g, f, 1);

  a->distanceFromStart = 0;  // set start node
  Dijkstras();
  PrintShortestRouteTo(f);
}
```

```
//////////////////

void Dijkstras() {
  while (nodes.size() > 0) {
    Node* smallest = ExtractSmallest(nodes);
    vector<Node*>* adjacentNodes =
      AdjacentRemainingNodes(smallest);

    const int size = adjacentNodes->size();
    for (int i = 0; i < size; ++i) {
      Node* adjacent = adjacentNodes->at(i);
      int distance = Distance(smallest, adjacent) +
               smallest->distanceFromStart;

      if (distance < adjacent->distanceFromStart) {
        adjacent->distanceFromStart = distance;
        adjacent->previous = smallest;
      }
    }
    delete adjacentNodes;
  }
}

// Find the node with the smallest distance,
// remove it, and return it.
Node* ExtractSmallest(vector<Node*>& nodes) {
  int size = nodes.size();
  if (size == 0) return NULL;
  int smallestPosition = 0;
  Node* smallest = nodes.at(0);
  for (int i = 1; i < size; ++i) {
    Node* current = nodes.at(i);
    if (current->distanceFromStart <
      smallest->distanceFromStart) {
      smallest = current;
      smallestPosition = i;
    }
  }
  nodes.erase(nodes.begin() + smallestPosition);
  return smallest;
}

// Return all nodes adjacent to 'node' which are still
// in the 'nodes' collection.
vector<Node*>* AdjacentRemainingNodes(Node* node) {
  vector<Node*>* adjacentNodes = new vector<Node*>();
  const int size = edges.size();
  for (int i = 0; i < size; ++i) {
    Edge* edge = edges.at(i);
    Node* adjacent = NULL;
    if (edge->node1 == node) {
      adjacent = edge->node2;
    } else if (edge->node2 == node) {
      adjacent = edge->node1;
```

```
    }
    if (adjacent && Contains(nodes, adjacent)) {
      adjacentNodes->push_back(adjacent);
    }
  }
  return adjacentNodes;
}

// Return distance between two connected nodes
int Distance(Node* node1, Node* node2) {
  const int size = edges.size();
  for (int i = 0; i < size; ++i) {
    Edge* edge = edges.at(i);
    if (edge->Connects(node1, node2)) {
      return edge->distance;
    }
  }
  return -1;  // should never happen
}

// Does the 'nodes' vector contain 'node'
bool Contains(vector<Node*>& nodes, Node* node) {
  const int size = nodes.size();
  for (int i = 0; i < size; ++i) {
    if (node == nodes.at(i)) {
      return true;
    }
  }
  return false;
}

//////////////////

void PrintShortestRouteTo(Node* destination) {
  Node* previous = destination;
  cout << "Distance from start: "
     << destination->distanceFromStart << endl;
  while (previous) {
    cout << previous->id << " ";
    previous = previous->previous;
  }
  cout << endl;
}

// these two not needed
vector<Edge*>* AdjacentEdges(vector<Edge*>& Edges, Node* node);
void RemoveEdge(vector<Edge*>& Edges, Edge* edge);

vector<Edge*>* AdjacentEdges(vector<Edge*>& edges, Node* node) {
  vector<Edge*>* adjacentEdges = new vector<Edge*>();

  const int size = edges.size();
  for (int i = 0; i < size; ++i) {
    Edge* edge = edges.at(i);
```

```
    if (edge->node1 == node) {
      cout << "adjacent: " << edge->node2->id << endl;
      adjacentEdges->push_back(edge);
    } else if (edge->node2 == node) {
      cout << "adjacent: " << edge->node1->id << endl;
      adjacentEdges->push_back(edge);
    }
  }
  return adjacentEdges;
}

void RemoveEdge(vector<Edge*>& edges, Edge* edge) {
  vector<Edge*>::iterator it;
  for (it = edges.begin(); it < edges.end(); ++it) {
    if (*it == edge) {
      edges.erase(it);
      return;
    }
  }
}
```

Read/Write class:

```
void QLHoaDon::KHCoSoDienNhoHon150(string filename)
{
  ofstream fout(filename, ios::binary);
  if (fout.is_open()) {
    int count = 0;
    for (int i = 0; i < size; i++) {
      if (listHD[i]->getSoLuong() < 150) {
        count++;
      }
    }
    fout.write(reinterpret_cast<const char*>(&count), sizeof(int));
    for (int i = 0; i < count; i++) {
      int type = -1;
      if (listHD[i]->getSoLuong() < 150) {
        if (dynamic_cast<KHVietNam*>(listHD[i]) != nullptr) {
          type = 0;
          fout.write(reinterpret_cast<const char*>(&type), sizeof(int));
          fout.write(reinterpret_cast<const char*>(listHD[i]), sizeof(KHVietNam));

        }
        else {
          type = 1;
          fout.write(reinterpret_cast<const char*>(&type), sizeof(int));
          fout.write(reinterpret_cast<const char*>(listHD[i]), sizeof(KHNuocNgoai));
        }
      }
    }
    cout << "Ghi file thanh cong!\n";
```

```
  }
  else {
  cout << "Khong the mo file!\n";

  }
}
```

Add new elements without changing its descending order

```
size = size + 1;
  int greaterIndex = -1;
  for (int i = 0; i < size - 1; i++) {
    if (stol(p->getMaKH()) >= stol(listHD[i]->getMaKH())) {

      greaterIndex = i - 1;
    }
  }

  for (int i = greaterIndex + 1; i < size - 1; i++) {
    listHD[i + 1] = listHD[i];
  }
  listHD[greaterIndex + 1] = p;
```

cin and cout operator overloading for subclass:

```
istream& operator>> (istream& in, Employee& e) {
  Person* p = &e;
  in >> *p;
  in.ignore(1);
  cout << "Enter corp: ";
  getline(in, e.Corp, '\n');
  cout << "Enter day: ";
  in >> e.Day;
  return in;
}
ostream& operator<< (ostream& out, Employee e) {
  Person* p = &e;
  out << *p;
  out << ", " << e.Corp << ", " << e.Day << endl;
  return out;
}
```

```cpp
};
void IndividualsManage::input() {
    cout << "Enter number: ";
    int n; cin >> n;
    list.resize(n);
    for (int i = 0; i < n; ++i) {
        cout << "1. Student " << endl << "2. Employee" << endl;
        cout << "Enter option: ";
        int opt; cin >> opt;
        cin.ignore(1);
        if (opt == 1) {
            list[i] = new Student;
            cin >> *(dynamic_cast<Student*>(list[i]));
        }
        else {
            list[i] = new Employee;
            cin >> *(dynamic_cast<Employee*>(list[i]));
        }
    }
}
void IndividualsManage::print() {
    for (int i = 0; i < list.size(); ++i) {
        if (Student* s = dynamic_cast<Student*>(list[i])){
            cout << *s;
        }
        else if (Employee* e = dynamic_cast<Employee*>(list[i])) {
            cout << *e;
        }
    }
}
```

Một số câu hỏi:

In C++, does the friend keyword violate the rule of data hiding in OOP?

Từ khóa friend trong C++ có thể coi là vi phạm quy tắc về ẩn dữ liệu (data hiding) trong lập trình hướng đối tượng (OOP).

Quy tắc ẩn dữ liệu trong OOP đề xuất rằng các thành viên dữ liệu của một lớp nên được che giấu và chỉ được truy cập thông qua các phương thức công khai (public methods) của lớp đó. Điều này nhằm bảo vệ tính riêng tư và đảm bảo tính nhất quán của dữ liệu bên trong một đối tượng.

Tuy nhiên, từ khóa friend trong C++ cho phép một lớp hoặc một hàm được khai báo là bạn (friend) của một lớp khác, cho phép truy cập vào các thành viên (bao gồm thành viên dữ liệu) của lớp đó mà không cần thông qua các phương thức công khai. Điều này có nghĩa là lớp hoặc hàm bạn có quyền truy cập trực tiếp vào dữ liệu riêng tư của lớp, bỏ qua quy tắc ẩn dữ liệu trong OOP.

Mặc dù từ khóa friend có thể vi phạm quy tắc về ẩn dữ liệu trong OOP, nhưng nó cũng cung cấp một cách linh hoạt để tạo ra các quan hệ đặc biệt giữa các lớp hoặc hàm.

Việc sử dụng friend có thể hữu ích trong một số trường hợp đặc biệt, như tăng hiệu suất hoặc cung cấp khả năng truy cập nhanh chóng cho các thành viên riêng tư.

Tuy nhiên, khi sử dụng từ khóa friend, cần cân nhắc kỹ lưỡng để đảm bảo tính bảo mật và nhất quán của dữ liệu. Việc sử dụng friend nên được giới hạn và đảm bảo rằng nó thực sự cần thiết và mang lại lợi ích rõ ràng trong thiết kế và triển khai của chương trình.

How can we avoid the diamond problem?

Chúng ta có thể tránh vấn đề diamond (diamond problem) trong đa kế thừa bằng cách sử dụng kế thừa ảo (virtual inheritance) trong C++. Kế thừa ảo cho phép chỉ định rõ ràng lớp cha chung và tránh xung đột giữa các lớp cha trong cấu trúc đa kế thừa.

Để sử dụng kế thừa ảo, chúng ta chỉ cần thêm từ khóa `virtual` *vào khai báo kế thừa của các lớp cha có lớp cha chung. Điều này cho biết chỉ có một phiên bản duy nhất của lớp cha chung được chia sẻ giữa các lớp con.*

Ví dụ, để tránh vấn đề diamond trong ví dụ trước, chúng ta có thể sử dụng kế thừa ảo như sau:

```
cppCopy code
class A {
public:
    void foo() {
        cout << "A::foo()" << endl;
    }
};

class B : virtual public A {
public:
    // ...
};

class C : virtual public A {
public:
    // ...
};

class D : public B, public C {
public:
    // ...
```

```
};
```

Bằng cách sử dụng kế thừa ảo, lớp `D` chỉ có một phiên bản duy nhất của lớp `A`, giúp tránh xung đột và rõ ràng hơn trong việc sử dụng thành viên của lớp chung.

Tóm lại, để tránh vấn đề diamond trong đa kế thừa, chúng ta có thể sử dụng kế thừa ảo (virtual inheritance) trong C++. Điều này giúp chỉ định rõ ràng lớp cha chung và tránh xung đột giữa các lớp cha, đảm bảo tính rõ ràng và dễ hiểu trong cấu trúc đa kế thừa.

Describe the diamond problem in multiple inheritance?

Vấn đề diamond (diamond problem) trong đa kế thừa là một tình huống xảy ra khi một lớp kế thừa từ hai lớp cha có một lớp cha chung. Nếu không được xử lý đúng, vấn đề này có thể dẫn đến xung đột và không rõ ràng về việc sử dụng các thành viên của lớp chung trong lớp con.

Hãy xem xét một ví dụ đơn giản để minh họa vấn đề này:

```
cppCopy code
class A {
public:
    void foo() {
        cout << "A::foo()" << endl;
    }
};

class B : public A {
public:
    void foo() {
        cout << "B::foo()" << endl;
    }
};

class C : public A {
public:
    void foo() {
        cout << "C::foo()" << endl;
    }
};

class D : public B, public C {
public:
    // Nếu không có sự ghi đè, đây là nơi xảy ra xung đột
};
```

Trong ví dụ trên, lớp `A` là lớp cha chung của cả lớp `B` và `C`. Lớp `D` kế thừa từ cả `B` và `C`. Vấn đề xảy ra khi chúng ta muốn gọi hàm `foo()` từ lớp `D`. Do `D` có thể truy cập đến `foo()` từ cả `B` và `C`, việc gọi `D::foo()` sẽ không rõ ràng và xung đột.

Để giải quyết vấn đề diamond, chúng ta có thể sử dụng các kỹ thuật như virtual inheritance (kế thừa ảo) để chỉ định rõ ràng lớp cha chung và tránh xung đột. Điều này có thể được thực hiện bằng cách sử dụng từ khóa `virtual` trong khai báo kế thừa của `B` và `C`:

```
cppCopy code
class B : virtual public A {
    // ...
};

class C : virtual public A {
    // ...
};
```

Với việc sử dụng kế thừa ảo, lớp `D` sẽ có quyền truy cập duy nhất đến một phiên bản của lớp `A`, tránh xung đột và đảm bảo rõ ràng hơn trong việc sử dụng thành viên của lớp chung.

Tóm lại, vấn đề diamond trong đa kế thừa xảy ra khi một lớp kế thừa từ hai lớp cha có một lớp cha chung. Để giải quyết vấn đề, kế thừa ảo (virtual inheritance) có thể được sử dụng để chỉ định rõ ràng lớp cha chung và tránh xung đột.

What is a pure virtual function? Give an example to illustrate.

Một hàm ảo tinh khiết là một hàm được khai báo trong một lớp cơ sở mà không có triển khai trong lớp cơ sở và dùng để được ghi đè (triển khai) trong các lớp dẫn xuất. Nó cũng được biết đến như là một phương thức trừu tượng hoặc phương thức ảo tinh khiết.

Hàm ảo tinh khiết được khai báo bằng cú pháp "= 0" ở cuối khai báo hàm trong lớp cơ sở. Ví dụ:

```
cppCopy code
class Shape {
public:
    virtual void draw() = 0; // Hàm ảo tinh khiết
```

```
};

class Circle : public Shape {
public:
    void draw() {
        // Triển khai của hàm draw() đặc biệt cho Circle
        // ...
    }
};

class Square : public Shape {
public:
    void draw() {
        // Triển khai của hàm draw() đặc biệt cho Square
        // ...
    }
};
```

Trong ví dụ trên, lớp `Shape` có một hàm ảo tinh khiết `draw()`. Các lớp `Circle` và `Square` là các lớp dẫn xuất từ `Shape` và phải triển khai hàm `draw()` theo cách riêng biệt cho mỗi lớp.

Nếu một lớp có một hàm ảo tinh khiết, nó trở thành một lớp trừu tượng. Một lớp trừu tượng không thể được khởi tạo trực tiếp, điều này có nghĩa là bạn không thể tạo đối tượng của lớp đó. Nó được sử dụng như một lớp cơ sở cho các lớp dẫn xuất khác và cung cấp một giao diện chung thông qua hàm ảo tinh khiết.

Sự hiện diện của một hàm ảo tinh khiết trong một lớp làm cho lớp đó trở thành một lớp trừu tượng vì nó cho thấy lớp cơ sở chưa hoàn chỉnh và yêu cầu các lớp dẫn xuất cung cấp một triển khai cho hàm đó. Mỗi lớp dẫn xuất phải ghi đè hàm ảo tinh khiết để cung cấp triển khai riêng cho nó. Việc không ghi đè hàm ảo tinh khiết trong một lớp dẫn xuất sẽ làm cho lớp dẫn xuất đó cũng trở thành lớp trừu tượng.

Lớp trừu tượng được sử dụng chủ yếu để định nghĩa một giao diện hoặc một tập hợp các hành vi chung mà các lớp dẫn xuất nên triển khai. Chúng hoạt động như các bản thiết kế cho các lớp dẫn xuất và cung cấp một cách để bắt buộc một số chức năng cụ thể trên nhiều lớp. Lớp trừu tượng thường được thiết kế để được phụ thuộc, và các đối tượng được tạo từ các lớp dẫn xuất cụ thể.

Tóm lại, việc có một hàm ảo tinh khiết trong một lớp làm cho lớp đó trở thành trừu tượng, ngăn cản việc khởi tạo trực tiếp và yêu cầu các lớp dẫn xuất cung cấp một triển khai cho hàm ảo tinh khiết đó.

Trình bày và cho ví dụ mã nguồn về hàm hủy ảo (virtual destructor) trong ngôn ngữ lập trình C++. Khi lập trình với ngôn ngữ lập trình C++, nếu không dùng hàm hủy ảo thì có nguy cơ gì? Bạn hãy cho ví dụ cụ thể có minh họa bằng mã nguồn.

Trong ngôn ngữ lập trình C++, hàm hủy ảo (virtual destructor) được sử dụng để đảm bảo rằng tất cả các tài nguyên đã được cấp phát cho một đối tượng con trong hệ thống kế thừa được giải phóng đúng cách. Hàm hủy ảo được khai báo bằng từ khóa "virtual" và có thể gọi đến hàm hủy của lớp cơ sở trong quá trình hủy một đối tượng con.

Dưới đây là một ví dụ về mã nguồn minh họa sử dụng hàm hủy ảo trong C++:

```
cppCopy code
#include <iostream>class Base {
public:
    Base() {
        std::cout << "Constructor of Base" << std::endl;
    }

    virtual ~Base() {
        std::cout << "Destructor of Base" << std::endl;
    }
};

class Derived : public Base {
public:
    Derived() {
        std::cout << "Constructor of Derived" << std::endl;
    }

    ~Derived() {
        std::cout << "Destructor of Derived" << std::endl;
    }
};

int main() {
    Base* obj = new Derived();
    delete obj;

    return 0;
}
```

Trong ví dụ này, chúng ta có hai lớp: Base và Derived. Lớp Derived kế thừa từ lớp Base và ghi đè (override) hàm hủy của lớp Base. Trong hàm main(), chúng ta tạo một con trỏ kiểu Base và gán nó cho một đối tượng Derived mới được tạo. Khi chúng ta gọi `delete`

`obj`, hàm hủy của lớp Derived được gọi trước, và sau đó hàm hủy của lớp Base được gọi thông qua hàm hủy ảo. Kết quả sẽ là:

```
Copy code
Constructor of Base
Constructor of Derived
Destructor of Derived
Destructor of Base
```

Nếu không sử dụng hàm hủy ảo trong trường hợp này, chỉ hàm hủy của lớp cơ sở (Base) sẽ được gọi khi `delete obj` được gọi. Điều này có thể dẫn đến việc không giải phóng đúng các tài nguyên được cấp phát trong lớp Derived, gây ra rò rỉ bộ nhớ và các vấn đề khác.

Dưới đây là ví dụ minh họa về nguy cơ xảy ra khi không sử dụng hàm hủy ảo:

```
cppCopy code
#include <iostream>class Base {
public:
    Base() {
        std::cout << "Constructor of Base" << std::endl;
    }

    ~Base() {
        std::cout << "Destructor of Base" << std::endl;
    }
};

class Derived : public Base {
private:
    int* dynamicArray;

public:
    Derived() {
        dynamicArray = new int[10];
        std::cout << "Constructor of Derived" << std::endl;
    }

    ~Derived() {
        delete[] dynamicArray;
        std::cout << "Destructor of Derived" << std::endl;
    }
};

int main() {
    Base* obj = new Derived();
```

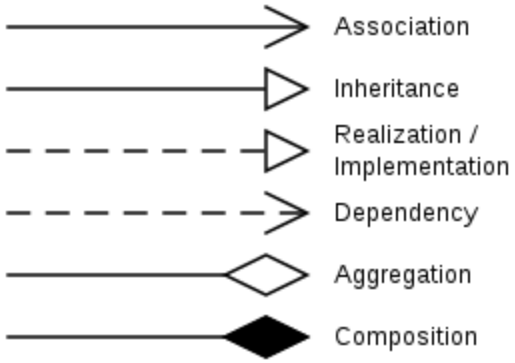
```
    delete obj;

    return 0;
}
```

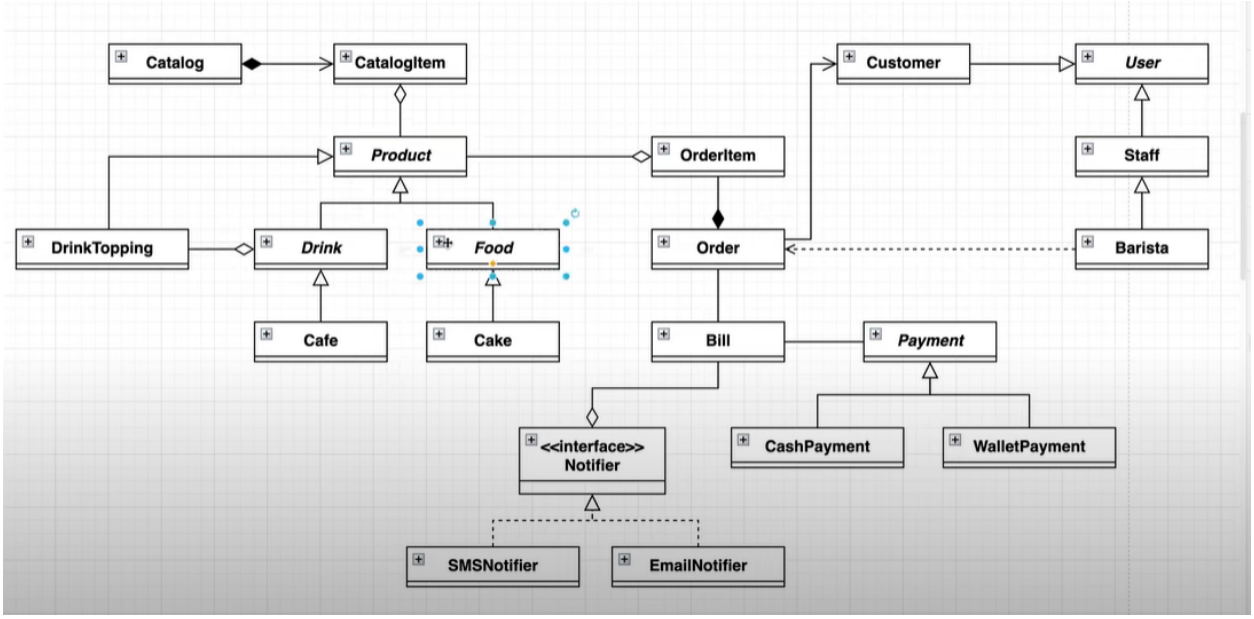
Trong ví dụ này, lớp Derived có một mảng động (dynamicArray) được cấp phát trong hàm khởi tạo. Tuy nhiên, vì không có hàm hủy ảo trong lớp Base, khi `delete obj` được gọi, chỉ có hàm hủy của lớp Base được gọi. Do đó, hàm hủy của lớp Derived không được gọi và mảng động dynamicArray không được giải phóng. Điều này dẫn đến rò rỉ bộ nhớ và có thể gây ra các vấn đề khác liên quan đến quản lý tài nguyên.

Trong ngôn ngữ lập trình hướng đối tượng (OOP) như C++, có hai khái niệm quan trọng là "lớp đối tượng" (class) và "đối tượng cụ thể" (object, instance). Dưới đây là sự khác biệt giữa hai khái niệm này:

1. Lớp đối tượng (class):
    - Lớp đối tượng định nghĩa cấu trúc, thuộc tính và hành vi của một nhóm đối tượng có tính chất tương đồng.
    - Nó chứa các thành viên dữ liệu (biến thành viên) và các phương thức (hàm thành viên) để thao tác với dữ liệu.
    - Lớp đối tượng là một bản thiết kế cho các đối tượng cụ thể, không tồn tại thực tế trong bộ nhớ.
2. Đối tượng cụ thể (object, instance):
    - Đối tượng cụ thể là một phiên bản cụ thể của một lớp đối tượng.
    - Nó được tạo ra từ lớp đối tượng bằng cách cấp phát bộ nhớ và khởi tạo.
    - Đối tượng cụ thể có thể truy cập và sử dụng các thành viên (biến thành viên và hàm thành viên) của lớp đối tượng.

UML:






```
#pragma once
#ifndef VECTOR_H
#define VECTOR_H
#include <iostream>
#include <string>
using namespace std;

/* vector class template */
template<typename T>
class vector {
private:
  T* ptr;
  int capacity;
  int sz;
public:

  // Constructors
  vector();
```

```
  vector(int cap);
  ~vector();
  // utilities methods
  /* operator[] */
  T& operator[](int n);
  /* Expand extra memory */
  void reserve(int newalloc);
  // input, output
  void push(const T& element);
  void erase(int pos);
  void print();
  int getCapacity() const;
  int getSz() const;
  void setCapacity(int cap);
  void setSz(int _sz);
};

#endif
```

```
#include "vector.h"
#include "Phim.h"
template<typename T>
vector<T>::vector() {
  capacity = 100;
  sz = 0;
  ptr = new T[capacity];
}

template<typename T>
vector<T>::vector(int cap) {
  capacity = cap;
  sz = 0;
  ptr = new T[capacity];
}

template<typename T>
vector<T>::~vector() {
  if (sz > 0) {
    delete[]ptr;
  }
}

template<typename T>
T& vector<T>::operator[](int n) {
  return ptr[n];
}

template<typename T>
void vector<T>::reserve(int newalloc) {
  if (sz >= capacity) {
    cout << "Khong the cap phat phan tu vi vuot qua suc chua (" << capacity << ")!\n";
```

```
    return;
  }
  T* temp = new T[ newalloc];
  for (int i = 0; i < sz; i++) {
    temp[i] = ptr[i];
  }
  delete[]ptr;
  ptr = temp;
  capacity += newalloc;
}

template<typename T>
void vector<T>::push(const T& element) {
  if (sz >= capacity) {
    cout << "Khong the them phan tu vi vuot qua suc chua (" << capacity << ")!\n";
    return;
  }
  ptr[sz] = element;
  sz++;
}

template<typename T>
void vector<T>::print() {
  for (int i = 0; i < sz; i++) {
    cout << ptr[i] << " ";
  }
}

template<typename T>
int vector<T>::getCapacity() const
{
  return capacity;
}

template<typename T>
int vector<T>::getSz() const
{
  return sz;
}

template<typename T>
void vector<T>::setSz(int _sz) {

  sz = _sz;
}

template<typename T>
void vector<T>::setCapacity(int cap) {
  capacity = cap;
}

template<typename T>
void vector<T>::erase(int pos) {
  T* temp = new T[sz - 1];
```

```
  for (int i = 0; i < pos; i++) {
    temp[i] = ptr[i];
  }
  for (int i = pos + 1; i < sz; i++) {
    temp[i] = ptr[i];
  }

  delete[] ptr;
  ptr = temp;
  sz--;
}


template class vector<string>;
template class vector<int>;
template class vector<double>;
template class vector<Phim*>;
```

Exception:

```
class myException {
private:
  string message;
public:
   myException() {
      message = "";
   }
   myException(string msg) {
      message = msg;
   }
   string getMessage() {
     return message;
   }
};
#endif

try {
  int age = 15;
  if (age >= 18) {
    cout << "Access granted - you are old enough.";
  } else {
    throw (age);
  }
}
catch (int myNum) {
  cout << "Access denied - You must be at least 18 years old.\n";
  cout << "Age is: " << myNum;
}
```