Information Security Lab

**Practical - 1: Write a code for Caesar Encryption**

**Code:**

#include <iostream>

#include <string>

using namespace std;

void encrypt(string &cipherText, const string &plainText, int key) {

  cipherText = "";

  for (char ch : plainText) {

    if (ch >= 'a' && ch <= 'z')

     cipherText += char((ch - 'a' + key) % 26 + 'a');

    else if (ch >= 'A' && ch <= 'Z')

      cipherText += char((ch - 'A' + key) % 26 + 'a');

    else cipherText += ch;

  }

}

void decrypt(string &plainText, const string &cipherText, int key) {

  plainText = "";

  for (char ch : cipherText) {

    if (ch >= 'a' && ch <= 'z')

      plainText += char((ch - 'a' - key + 26) % 26 + 'a');

    else if (ch >= 'A' && ch <= 'Z')

      plainText += char((ch - 'A' - key + 26) % 26 + 'a');

    else plainText += ch;

  }

}

int main() {

  string plainText, cipherText, decryptedText;

  int key;

  cout << "Enter your message: ";

  getline(cin, plainText);

  cout << "Enter the shift value: ";

  cin >> key;

  encrypt(cipherText, plainText, key);

  cout << "Encrypted message: " << cipherText << endl;

  decrypt(decryptedText, cipherText, key);

  cout << "Decrypted message: " << decryptedText << endl;

  return 0;

}

**Output:**

A screenshot of a computer screen

AI-generated content may be incorrect.

**Variation in Code:**

#include <iostream>

#include <string>

using namespace std;

void encrypt(string &cipherText, const string &plainText, int key1, int key2) {

cipherText = "";

for (int i = 0; i < plainText.size(); i++) {

char ch = plainText[i];

int key = (i % 2 == 0) ? key1 : key2;

if (ch >= 'a' && ch <= 'z')

cipherText += char((ch - 'a' + key) % 26 + 'a');

else if (ch >= 'A' && ch <= 'Z')

cipherText += char((ch - 'A' + key) % 26 + 'A');

else

cipherText += ch;

}

}

void decrypt(string &plainText, const string &cipherText, int key1, int key2) {

plainText = "";

for (int i = 0; i < cipherText.size(); i++) {

char ch = cipherText[i];

int key = (i % 2 == 0) ? key1 : key2;

if (ch >= 'a' && ch <= 'z')

plainText += char((ch - 'a' - key + 26) % 26 + 'a');

else if (ch >= 'A' && ch <= 'Z')

plainText += char((ch - 'A' - key + 26) % 26 + 'A');

else

plainText += ch;

}

}

int main() {

string plainText, cipherText, decryptedText;

int key1, key2;

cout << "Enter your message: ";

getline(cin, plainText);

cout << "Enter first key: ";

cin >> key1;

cout << "Enter second key: ";

cin >> key2;

encrypt(cipherText, plainText, key1, key2);

cout << "Encrypted message: " << cipherText << endl;

decrypt(decryptedText, cipherText, key1, key2);

cout << "Decrypted message: " << decryptedText << endl;

return 0;

}

**Output:**



**Security Analysis:**

1. Standard Caesar Cipher:

* This Caesar cipher implementation is extremely insecure encryption, as it uses a simple substitution method with a fixed shift (key) over the alphabet and digits, making it highly vulnerable to brute-force and frequency analysis attacks. Since the key space is only 26 possibilities for letters and 10 for digits, an attacker can trivially try all possible shifts within milliseconds. Furthermore, the code converts uppercase letters to lowercase during encryption and decryption, which not only loses original case information but also makes patterns even easier to detect. There is no randomness, key expansion, or use of modern encryption primitives, so it offers no real confidentiality against any motivated attacker. This approach is suitable only for educational or entertainment purposes, not for securing sensitive data.

1. Variation Caesar Cipher:
   * The modified two-key Caesar cipher improves security over the traditional single-shift Caesar cipher by introducing an alternating key mechanism, which increases the key space from 26 possibilities to 26 × 26 = 676, making brute-force attacks slightly harder. It also preserves case sensitivity, which avoids information leakage from forced lowercase conversions. However, the scheme remains fundamentally weak because it is still a monoalphabetic substitution within each position group (even and odd indices). This means frequency analysis can be applied separately to even- and odd-positioned characters, reducing the cipher’s resistance to classical cryptanalysis. Additionally, the key space is still small by modern standards, making exhaustive search trivial for computers. While this variation is a good educational step toward stronger ciphers like the Vigenère cipher, it should not be considered secure for protecting sensitive data in real-world applications.

**Practical 2: Play-Fair Cipher**

Code:

#include <bits/stdc++.h>

using namespace std;

string delete\_char\_cpp(string s, char ch) {

string new\_str = "";

for (char c : s) {

if (c != ch) {

new\_str += c;

}

}

return new\_str;

}

pair<array<array<char, 5>, 5>, unordered\_map<char, pair<int,int>>> buildMatrix(const string& key) {

array<array<char, 5>, 5> matrix{};

char skipChar = 'j';

string ap = "abcdefghijklmnopqrstuvwxyz";

string alphabets = delete\_char\_cpp(ap, skipChar);

unordered\_map<char, pair<int,int>> used;

int n = key.length(), k = 0, a = 0;

for (int i = 0; i < 5; i++) {

for (int j = 0; j < 5; j++) {

if (k < n) {

while (k < n && used.find(key[k]) != used.end()) k++;

if (k < n) {

matrix[i][j] = key[k];

used[key[k]] = {i,j};

k++;

}

} else {

while (a < alphabets.size() && used.find(alphabets[a]) != used.end()) a++;

if (a < alphabets.size()) {

matrix[i][j] = alphabets[a];

used[alphabets[a]] = {i,j};

a++;

}

}

}

}

return {matrix, used};

}

vector<string> preprocessPlainText(string& plainText) {

char skipChar = 'j';

for (char &ch: plainText) {

if (ch == skipChar) {

ch = 'i'; // replace j with i

}

}

vector<string> cleanedText;

int len = plainText.length() - (plainText.length() % 2);

for (int i = 0 ; i < len; i+=2) {

string temp = "";

if (plainText[i] != plainText[i + 1]) {

temp += plainText[i];

temp += plainText[i+1];

cleanedText.push\_back(temp);

} else {

temp += plainText[i];

temp += 'z';

cleanedText.push\_back(temp);

temp = "";

temp += plainText[i+1];

temp += 'z';

cleanedText.push\_back(temp);

}

}

if (plainText.length() % 2) {

string temp = "";

temp += plainText[len];

temp += 'z';

cleanedText.push\_back(temp);

}

return cleanedText;

}

string encrypt(string& plainText, const string& key) {

auto [matrix, used] = buildMatrix(key);

vector<string> cleanedText = preprocessPlainText(plainText);

string cipherText = "";

for (const auto &x: cleanedText) {

pair<int,int> l1 = used[x[0]];

pair<int,int> l2 = used[x[1]];

if (l1.second == l2.second) {

cipherText += matrix[(l1.first + 1) % 5][l1.second];

cipherText += matrix[(l2.first + 1) % 5][l2.second];

} else if (l1.first == l2.first) {

cipherText += matrix[l1.first][(l1.second + 1) % 5];

cipherText += matrix[l1.first][(l2.second + 1) % 5];

} else {

cipherText += matrix[l1.first][l2.second];

cipherText += matrix[l2.first][l1.second];

}

}

return cipherText;

}

string decrypt(string& cipherText, const string& key) {

auto [matrix, used] = buildMatrix(key);

string plainText = "";

for (int i = 0; i < cipherText.length(); i += 2) {

pair<int,int> l1 = used[cipherText[i]];

pair<int,int> l2 = used[cipherText[i+1]];

if (l1.second == l2.second) {

plainText += matrix[(l1.first - 1 + 5) % 5][l1.second];

plainText += matrix[(l2.first - 1 + 5) % 5][l2.second];

} else if (l1.first == l2.first) {

plainText += matrix[l1.first][(l1.second - 1 + 5) % 5];

plainText += matrix[l1.first][(l2.second - 1 + 5) % 5];

} else {

plainText += matrix[l1.first][l2.second];

plainText += matrix[l2.first][l1.second];

}

}

return plainText;

}

int main() {

string plainText;

string key;

cout << "Enter Message : ";

getline(cin, plainText);

cout << "Enter Key: ";

cin >> key;

string cipherText = encrypt(plainText, key);

cout << "Encrypted: " << cipherText << endl;

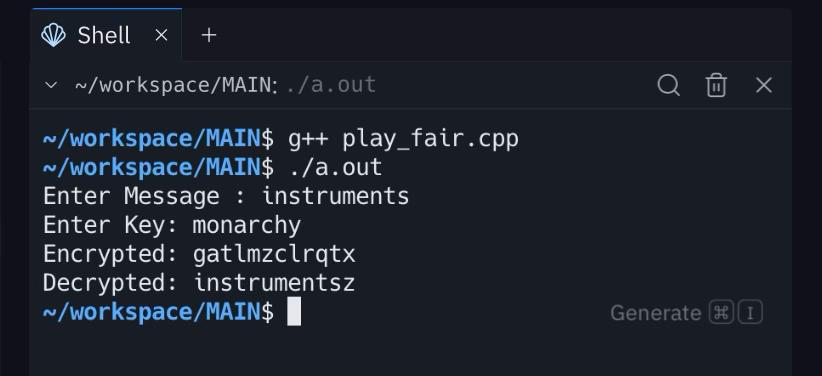
string decryptedText = decrypt(cipherText, key);

cout << "Decrypted: " << decryptedText << endl;

return 0;

}

Output:



**Variation in Code:**

#include <bits/stdc++.h>

using namespace std;

string delete\_char\_cpp(string s, char ch) {

string new\_str = "";

for (char c : s) {

if (c != ch) {

new\_str += c;

}

}

return new\_str;

}

string caesarShift(const string& text, int shift) {

string res = "";

for (char c : text) {

if (isalpha(c)) {

char base = 'a';

res += char((c - base + shift + 26) % 26 + base);

} else res += c;

}

return res;

}

pair<array<array<char, 5>, 5>, unordered\_map<char, pair<int,int>>> buildMatrix(const string& key) {

array<array<char, 5>, 5> matrix{};

string vowels = "aeiou";

char skipChar = vowels[key.length() % vowels.size()];

string ap = "abcdefghijklmnopqrstuvwxyz";

string alphabets = delete\_char\_cpp(ap, skipChar);

unordered\_map<char, pair<int,int>> used;

int n = key.length(), k = 0, a = 0;

// XOR with key length

string transformedKey = key;

for (char &c : transformedKey) {

c = ((c - 'a' + key.length()) % 26) + 'a';

}

for (int i = 0; i < 5; i++) {

for (int j = 0; j < 5; j++) {

if (k < n) {

while (k < n && used.find(transformedKey[k]) != used.end()) k++;

if (k < n) {

matrix[i][j] = transformedKey[k];

used[transformedKey[k]] = {i,j};

k++;

}

} else {

while (a < alphabets.size() && used.find(alphabets[a]) != used.end()) a++;

if (a < alphabets.size()) {

matrix[i][j] = alphabets[a];

used[alphabets[a]] = {i,j};

a++;

}

}

}

}

return {matrix, used};

}

vector<string> preprocessPlainText(string& plainText) {

for (char &ch: plainText) {

if (!isalpha(ch)) continue;

ch = tolower(ch);

}

vector<string> cleanedText;

string fillers = "xyz";

int len = plainText.length() - (plainText.length() % 2);

for (int i = 0 ; i < len; i+=2) {

string temp = "";

if (plainText[i] != plainText[i + 1]) {

temp += plainText[i];

temp += plainText[i+1];

cleanedText.push\_back(temp);

} else {

temp += plainText[i];

temp += fillers[(i/2) % fillers.size()];

cleanedText.push\_back(temp);

temp = "";

temp += plainText[i+1];

temp += fillers[(i/2+1) % fillers.size()];

cleanedText.push\_back(temp);

}

}

if (plainText.length() % 2) {

string temp = "";

temp += plainText[len];

temp += fillers[(len/2) % fillers.size()];

cleanedText.push\_back(temp);

}

return cleanedText;

}

string encrypt(string& plainText, const string& key) {

auto [matrix, used] = buildMatrix(key);

vector<string> cleanedText = preprocessPlainText(plainText);

string cipherText = "";

for (const auto &x: cleanedText) {

pair<int,int> l1 = used[x[0]];

pair<int,int> l2 = used[x[1]];

if (l1.second == l2.second) {

cipherText += matrix[(l1.first + 1) % 5][l1.second];

cipherText += matrix[(l2.first + 1) % 5][l2.second];

} else if (l1.first == l2.first) {

cipherText += matrix[l1.first][(l1.second + 1) % 5];

cipherText += matrix[l1.first][(l2.second + 1) % 5];

} else {

cipherText += matrix[l1.first][l2.second];

cipherText += matrix[l2.first][l1.second];

}

}

cipherText = caesarShift(cipherText, key.length() % 26);

return cipherText;

}

string decrypt(string& cipherText, const string& key) {

cipherText = caesarShift(cipherText, -(key.length() % 26));

auto [matrix, used] = buildMatrix(key);

string plainText = "";

for (int i = 0; i < cipherText.length(); i += 2) {

pair<int,int> l1 = used[cipherText[i]];

pair<int,int> l2 = used[cipherText[i+1]];

if (l1.second == l2.second) {

plainText += matrix[(l1.first - 1 + 5) % 5][l1.second];

plainText += matrix[(l2.first - 1 + 5) % 5][l2.second];

} else if (l1.first == l2.first) {

plainText += matrix[l1.first][(l1.second - 1 + 5) % 5];

plainText += matrix[l1.first][(l2.second - 1 + 5) % 5];

} else {

plainText += matrix[l1.first][l2.second];

plainText += matrix[l2.first][l1.second];

}

}

return plainText;

}

int main() {

string plainText;

string key;

cout << "Enter Message : ";

getline(cin, plainText);

cout << "Enter Key: ";

cin >> key;

string cipherText = encrypt(plainText, key);

cout << "Encrypted: " << cipherText << endl;

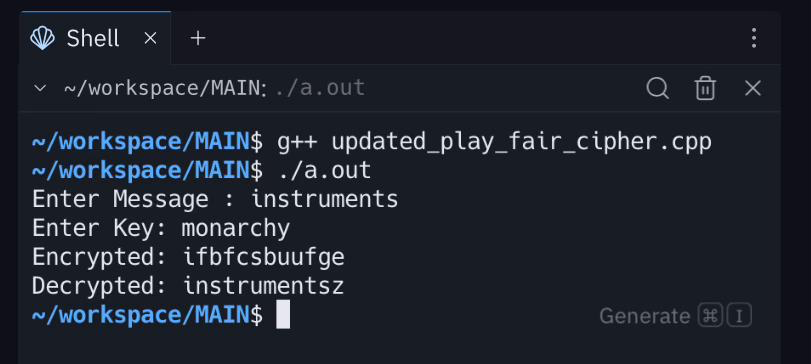
string decryptedText = decrypt(cipherText, key);

cout << "Decrypted: " << decryptedText << endl;

return 0;

}

**Output:**



**Security Analysis:**

1. Standard Play-Fair Cipher:

* This Caesar cipher implementation is extremely insecure encryption, as it uses a simple substitution method with a fixed shift (key) over the alphabet and digits, making it highly vulnerable to brute-force and frequency analysis attacks. Since the key space is only 26 possibilities for letters and 10 for digits, an attacker can trivially try all possible shifts within milliseconds. Furthermore, the code converts uppercase letters to lowercase during encryption and decryption, which not only loses original case information but also makes patterns even easier to detect. There is no randomness, key expansion, or use of modern encryption primitives, so it offers no real confidentiality against any motivated attacker. This approach is suitable only for educational or entertainment purposes, not for securing sensitive data.

1. Variation Cipher:

* The modified two-key Caesar cipher improves security over the traditional single-shift Caesar cipher by introducing an alternating key mechanism, which increases the key space from 26 possibilities to 26 × 26 = 676, making brute-force attacks slightly harder. It also preserves case sensitivity, which avoids information leakage from forced lowercase conversions. However, the scheme remains fundamentally weak because it is still a monoalphabetic substitution within each position group (even and odd indices). This means frequency analysis can be applied separately to even- and odd-positioned characters, reducing the cipher’s resistance to classical cryptanalysis. Additionally, the key space is still small by modern standards, making exhaustive search trivial for computers. While this variation is a good educational step toward stronger ciphers like the Vigenère cipher, it should not be considered secure for protecting sensitive data in real-world applications.

**Practical 3: Rail-Fence Cipher**

Code:

#include <bits/stdc++.h>

using namespace std;

string encrypt(string plainText, int key) {

    int n = key, m = plainText.length();

    vector<vector<char>> matrix(n, vector<char>(m, '.'));

    int cnt = 0, flag = 0, i = 0, j = 0;

    while (j < m) {

        matrix[i][j] = plainText[j];

        cnt++;

        if (cnt >= key) {

            flag = !flag;

            cnt = 1;

        }

        if (!flag) i++;

        else i--;

        j++;

    }

    string encryptedText = "";

    for (int i = 0; i < n; i++) {

        for (int j = 0; j < m; j++) {

            if (matrix[i][j] != '.')

                encryptedText += matrix[i][j];

        }

    }

    return encryptedText;

}

string decrypt(string cipherText, int key) {

    int m = cipherText.length(), n = key;

    vector<vector<char>> matrix(n, vector<char>(m, '.'));

    int cnt = 0, flag = 0, i = 0, j = 0;

    while (j < m) {

        matrix[i][j] = '\*';

        cnt++;

        if (cnt >= key) {

            flag = !flag;

            cnt = 1;

        }

        if (!flag) i++;

        else i--;

        j++;

    }

    int index = 0;

    for (int r = 0; r < n; r++) {

        for (int c = 0; c < m; c++) {

            if (matrix[r][c] == '\*' && index < m) {

                matrix[r][c] = cipherText[index++];

            }

        }

    }

    string decryptedText = "";

    cnt = 0; flag = 0; i = 0; j = 0;

    while (j < m) {

        decryptedText += matrix[i][j];

        cnt++;

        if (cnt >= key) {

            flag = !flag;

            cnt = 1;

        }

        if (!flag) i++;

        else i--;

        j++;

    }

    return decryptedText;

}

int main() {

    string plainText = "wearediscoveringrailfencecipher";

    int key = 3;

    string encryptedText = encrypt(plainText, key);

    cout << "Encrypted Text : " << encryptedText << endl;

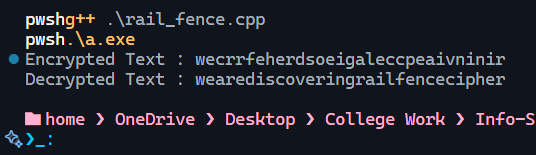
    string decryptedText = decrypt(encryptedText, key);

    cout << "Decrypted Text : " << decryptedText << endl;

    return 0;

}

Output:



**Variation in Code :**

#include <bits/stdc++.h>

using namespace std;

vector<int> generatePermutation(int size, int key) {

    vector<int> perm(size);

    iota(perm.begin(), perm.end(), 0);

    srand(key);

    random\_shuffle(perm.begin(), perm.end());

    return perm;

}

string encrypt(string plainText, int key) {

    int n = key, m = plainText.length();

    vector<vector<char>> matrix(n, vector<char>(m, '.'));

    int dir = 1, row = 0;

    for (int col = 0; col < m; col++) {

        matrix[row][col] = plainText[col];

        row += dir;

        if (row == 0 || row == n - 1) dir \*= -1;

    }

    vector<int> colPerm = generatePermutation(m, key);

    vector<char> shuffled(m);

    for (int col = 0; col < m; col++)

        shuffled[colPerm[col]] = plainText[col];

    string encrypted = "";

    for (int r = 0; r < n; r++)

        for (int c = 0; c < m; c++)

            if (matrix[r][c] != '.') encrypted += shuffled[c];

    return encrypted;

}

string decrypt(string cipherText, int key) {

    int m = cipherText.length(), n = key;

    vector<vector<char>> matrix(n, vector<char>(m, '.'));

    int dir = 1, row = 0;

    for (int col = 0; col < m; col++) {

        matrix[row][col] = '\*';

        row += dir;

        if (row == 0 || row == n - 1) dir \*= -1;

    }

    int idx = 0;

    for (int r = 0; r < n; r++)

        for (int c = 0; c < m; c++)

            if (matrix[r][c] == '\*') matrix[r][c] = cipherText[idx++];

    string railText = "";

    dir = 1; row = 0;

    for (int col = 0; col < m; col++) {

        railText += matrix[row][col];

        row += dir;

        if (row == 0 || row == n - 1) dir \*= -1;

    }

    vector<int> colPerm = generatePermutation(m, key);

    vector<char> unshuffled(m);

    for (int col = 0; col < m; col++)

        unshuffled[col] = railText[colPerm[col]];

    string decrypted = "";

    for (char c : unshuffled) decrypted += c;

    return decrypted;

}

int main() {

    string plainText = "wearediscoveringrailfencecipher";

    int key = 3;

    string encryptedText = encrypt(plainText, key);

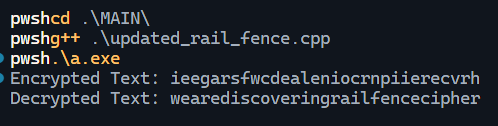
    cout << "Encrypted Text: " << encryptedText << endl;

    string decryptedText = decrypt(encryptedText, key);

    cout << "Decrypted Text: " << decryptedText << endl;

}

Ouput:



**Security Analysis:**

1. **Standard Rail Fence Cipher :** The standard rail fence cipher works by writing the text in a zig-zag pattern across a fixed number of rails and then reading it row by row.

* **Strengths:**
  + Very simple and easy to implement.
  + Provides a basic level of scrambling that hides the exact order of characters.
* **Weaknesses:**
  + The key space is extremely small (limited to the number of rails). For a message of length *m*, the maximum key values are between 2 and *m–1*, which can be brute-forced easily.
  + The cipher only rearranges letters without changing them, so letter frequencies remain intact. This means frequency analysis can still reveal patterns.
  + With only ciphertext available, an attacker can test all possible rail values and recover the message quickly.

**Conclusion:** The standard rail fence cipher is not secure. It is mainly useful for educational purposes to understand classical transposition methods.

1. **Updated Rail Fence Cipher with Column Permutation :** The updated version improves on the basic method by first applying the zig-zag rail pattern and then shuffling the columns using a permutation derived from a key.

* **Strengths:**
  + The additional column shuffling introduces more complexity and makes the ciphertext appear more jumbled compared to the standard version.
  + This approach increases the apparent key space since the ciphertext depends not only on the number of rails but also on the column order.
  + Brute-force attacks become harder compared to the basic method.
* **Weaknesses:**
  + The permutation is generated using the standard C++ random function seeded with the key (srand). This random number generator is predictable and can be recreated by an attacker who guesses the key.
  + The actual key (number of rails) is still small, so an attacker only needs to try a limited range of values.
  + As with the standard method, the letter frequencies remain unchanged, making it vulnerable to statistical analysis.
  + If part of the plaintext is known (for example, common words or phrases), the key and permutation can be deduced quickly.

**Conclusion:** The updated version is more secure than the standard rail fence cipher, but it is still not suitable for modern cryptographic use. It acts more like an obfuscation technique rather than true encryption.

**Practical 4: Columnar Cipher**

**Code:**

#include <bits/stdc++.h>

using namespace std;

// Function to get order of key

vector<int> getKeyOrder(string key) {

   int n = key.size();

   vector<pair<char,int>> keyWithIndex;

   for(int i=0; i<n; i++)

      keyWithIndex.push\_back({key[i], i});

   // Sort by character (alphabetical order)

   sort(keyWithIndex.begin(), keyWithIndex.end());

   // Assign order numbers

   vector<int> order(n);

   for(int i=0; i<n; i++)

      order[keyWithIndex[i].second] = i;

   return order;

}

// Encryption

string columnarEncrypt(string plaintext, string key) {

   int n = key.size();

   vector<int> order = getKeyOrder(key);

   // Remove spaces (optional)

   plaintext.erase(remove(plaintext.begin(), plaintext.end(), ' '), plaintext.end());

   // Fill into matrix row-wise

   int rows = ceil((double)plaintext.size() / n);

   vector<vector<char>> matrix(rows, vector<char>(n, 'X')); // pad with 'X'

   int k = 0;

   for(int i=0; i<rows; i++) {

      for(int j=0; j<n; j++) {

         if(k < plaintext.size())

               matrix[i][j] = plaintext[k++];

      }

   }

   // Read column by column in key order

   string cipher = "";

   for(int col=0; col<n; col++) {

      int currCol = find(order.begin(), order.end(), col) - order.begin();

      for(int row=0; row<rows; row++)

         cipher.push\_back(matrix[row][currCol]);

   }

   return cipher;

}

// Decryption

string columnarDecrypt(string ciphertext, string key) {

   int n = key.size();

   vector<int> order = getKeyOrder(key);

   int rows = ceil((double)ciphertext.size() / n);

   vector<vector<char>> matrix(rows, vector<char>(n, 'X'));

   // Fill column by column according to order

   int k = 0;

   for(int col=0; col<n; col++) {

      int currCol = find(order.begin(), order.end(), col) - order.begin();

      for(int row=0; row<rows; row++) {

         if(k < ciphertext.size())

               matrix[row][currCol] = ciphertext[k++];

      }

   }

   // Read row by row to get plaintext

   string plaintext = "";

   for(int i=0; i<rows; i++) {

      for(int j=0; j<n; j++) {

         if(matrix[i][j] != 'X')

               plaintext.push\_back(matrix[i][j]);

      }

   }

   return plaintext;

}

// Main

int main() {

   string plaintext = "WEAREDISCOVERINGCOLUMNAR";

   string key = "ZEBRAS";

   string cipher = columnarEncrypt(plaintext, key);

   cout << "Encrypted: " << cipher << endl;

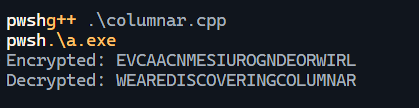
   string decrypted = columnarDecrypt(cipher, key);

   cout << "Decrypted: " << decrypted << endl;

   return 0;

}

**Output:**

****

**Variation in Code:**

#include <bits/stdc++.h>

using namespace std;

vector<int> getKeyOrder(string key) {

    int n = key.size();

    vector<pair<char,int>> keyWithIndex;

    for (int i = 0; i < n; i++) keyWithIndex.push\_back({key[i], i});

    sort(keyWithIndex.begin(), keyWithIndex.end());

    vector<int> order(n);

    for (int i = 0; i < n; i++) order[keyWithIndex[i].second] = i;

    return order;

}

string padPlaintext(string plaintext, int blockSize, int seed) {

    srand(seed);

    while (plaintext.size() % blockSize != 0) {

        plaintext.push\_back('A' + rand() % 26); // random padding

    }

    return plaintext;

}

string columnarEncrypt(string plaintext, string key) {

    int n = key.size();

    vector<int> colOrder = getKeyOrder(key);

    int seed = accumulate(key.begin(), key.end(), 0);

    plaintext = padPlaintext(plaintext, n, seed);

    int rows = plaintext.size() / n;

    vector<vector<char>> matrix(rows, vector<char>(n));

    int k = 0;

    for (int i = 0; i < rows; i++)

        for (int j = 0; j < n; j++)

            matrix[i][j] = plaintext[k++];

    // Shuffle rows based on key-derived seed

    vector<int> rowOrder(rows);

    iota(rowOrder.begin(), rowOrder.end(), 0);

    srand(seed);

    random\_shuffle(rowOrder.begin(), rowOrder.end());

    // Read columns in scrambled order

    string cipher = "";

    for (int col = 0; col < n; col++) {

        int realCol = find(colOrder.begin(), colOrder.end(), col) - colOrder.begin();

        for (int r = 0; r < rows; r++) cipher.push\_back(matrix[rowOrder[r]][realCol]);

    }

    // Apply offset rotation based on key

    int offset = seed % cipher.size();

    rotate(cipher.begin(), cipher.begin() + offset, cipher.end());

    return cipher;

}

string columnarDecrypt(string ciphertext, string key) {

    int n = key.size();

    vector<int> colOrder = getKeyOrder(key);

    int seed = accumulate(key.begin(), key.end(), 0);

    // Undo rotation

    int offset = seed % ciphertext.size();

    rotate(ciphertext.begin(), ciphertext.end() - offset, ciphertext.end());

    int rows = ciphertext.size() / n;

    vector<vector<char>> matrix(rows, vector<char>(n));

    vector<int> rowOrder(rows);

    iota(rowOrder.begin(), rowOrder.end(), 0);

    srand(seed);

    random\_shuffle(rowOrder.begin(), rowOrder.end());

    // Place ciphertext back into columns

    int k = 0;

    for (int col = 0; col < n; col++) {

        int realCol = find(colOrder.begin(), colOrder.end(), col) - colOrder.begin();

        for (int r = 0; r < rows; r++)

            matrix[rowOrder[r]][realCol] = ciphertext[k++];

    }

    // Read row-wise

    string plaintext = "";

    for (int i = 0; i < rows; i++)

        for (int j = 0; j < n; j++)

            plaintext.push\_back(matrix[i][j]);

    return plaintext;

}

int main() {

    string plaintext = "WEAREDISCOVERINGCOLUMNAR";

    string key = "ZEBRAS";

    string cipher = columnarEncrypt(plaintext, key);

    cout << "Encrypted: " << cipher << endl;

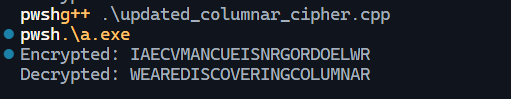
    string decrypted = columnarDecrypt(cipher, key);

    cout << "Decrypted: " << decrypted << endl;

    return 0;

}

**Output:**



**1. Standard Columnar Cipher (First Code)**

**🔹 Algorithm Summary**

* A keyword is used to determine the order of columns.
* Plaintext is written row by row into a grid, padding with X if needed.
* Ciphertext is read column by column, in the order defined by the sorted keyword.
* Decryption reverses the process.

**🔹 Strengths**

* More secure than simple Rail Fence since column order depends on a keyword.
* Different keywords produce very different ciphertexts, giving flexibility.
* Padding with X ensures grid completion, preventing ciphertext length from leaking message structure too much.

**🔹 Weaknesses**

1. **Keyspace is limited**
   * Security depends only on the keyword length and letters.
   * With common dictionary words (like *ZEBRAS*), brute-force or dictionary attacks are possible.
2. **Letter frequencies unchanged**
   * Only the positions of letters are permuted.
   * Frequency analysis still applies, making ciphertext vulnerable.
3. **Predictable padding**
   * Padding with X is easily recognizable and can leak plaintext structure.
4. **Vulnerable to brute-force + anagramming**
   * An attacker can guess the number of columns and test permutations to see which makes sense.

**Conclusion:** This is a classic transposition cipher. Stronger than Rail Fence, but still vulnerable to frequency analysis and brute force with modern computation.

**2. Enhanced Columnar Cipher with Row Shuffling + Rotation (Second Code)**

**🔹 Algorithm Summary**

1. Same basic columnar transposition as above.
2. **Improvements added:**
   1. **Random padding**: Instead of fixed X, random letters are used.
   2. **Row shuffling**: Rows are scrambled based on a key-derived seed.
   3. **Rotation step**: Ciphertext is rotated by an offset derived from the key.
3. Decryption reverses these steps systematically.

**🔹 Strengths**

1. **Stronger diffusion**
   * Row shuffling hides plaintext structure better than standard columnar.
   * Rotation further disguises column boundaries.
2. **Random padding**
   * Padding characters are unpredictable, unlike fixed X.
   * Harder for an attacker to identify where padding begins.
3. **Increased complexity**
   * Even if the attacker knows it’s a columnar cipher, they must also handle row permutation and ciphertext rotation.

**🔹 Weaknesses**

1. **Predictable randomness**
   * Uses srand(seed) with a seed derived from the keyword.
   * rand() is not cryptographically secure and can be replicated by attackers.
2. **Seed space is small**
   * The seed is just the ASCII sum of the keyword (e.g., "ZEBRAS" → 90+69+... = small integer).
   * This drastically limits the key strength — attacker can brute-force all possible sums.
3. **Still preserves frequencies**
   * Despite extra scrambling, characters are not substituted or masked.
   * Statistical attacks remain effective.
4. **Vulnerable to known-plaintext attack**
   * If part of plaintext is known (like “WE ARE”), the row shuffle and rotation can be reconstructed.

**Conclusion:** This variation improves over the standard method by adding randomness, padding, and transformations. However, because the randomization is based on a weak predictable generator and the key space is small, it is still not strong enough for real-world cryptography.

**Practical 5: Vigenere Cipher**

Code:

#include <bits/stdc++.h>

using namespace std;

char vigenereTable[26][26];

void initVigenereTable() {

    for (int i = 0; i < 26; i++) {

        for (int j = 0; j < 26; j++) {

            vigenereTable[i][j] = 'A' + (i + j) % 26;

        }

    }

}

string generateKey(const string &text, const string &key) {

    string newKey = key;

    while (newKey.size() < text.size())

        newKey += key;

    return newKey.substr(0, text.size());

}

string vigenereEncrypt(const string &plaintext, const string &key) {

    string newKey = generateKey(plaintext, key);

    string cipher = "";

    for (int i = 0; i < plaintext.size(); i++) {

        char p = toupper(plaintext[i]);

        char k = toupper(newKey[i]);

        if (isalpha(plaintext[i])) {

            char enc = vigenereTable[p - 'A'][k - 'A'];

            cipher += isupper(plaintext[i]) ? enc : tolower(enc);

        } else {

            cipher += plaintext[i];

        }

    }

    return cipher;

}

string vigenereDecrypt(const string &ciphertext, const string &key) {

    string newKey = generateKey(ciphertext, key);

    string plain = "";

    for (int i = 0; i < ciphertext.size(); i++) {

        char c = toupper(ciphertext[i]);

        char k = toupper(newKey[i]);

        if (isalpha(ciphertext[i])) {

            int row = k - 'A';

            int col = 0;

            while (col < 26 && vigenereTable[row][col] != c) col++;

            char dec = 'A' + col;

            plain += isupper(ciphertext[i]) ? dec : tolower(dec);

        } else {

            plain += ciphertext[i];

        }

    }

    return plain;

}

int main() {

    initVigenereTable();

    for(int i = 0; i < 26; i++){

        for(int j = 0; j < 26; j++){

            cout << vigenereTable[i][j] << " ";

        }

        cout << endl;

    }

    cout << endl;

    string plaintext = "WEAREDISCOVERINGVIGENERE";

    string key = "KEY";

    string encrypted = vigenereEncrypt(plaintext, key);

    cout << "Encrypted: " << encrypted << endl;

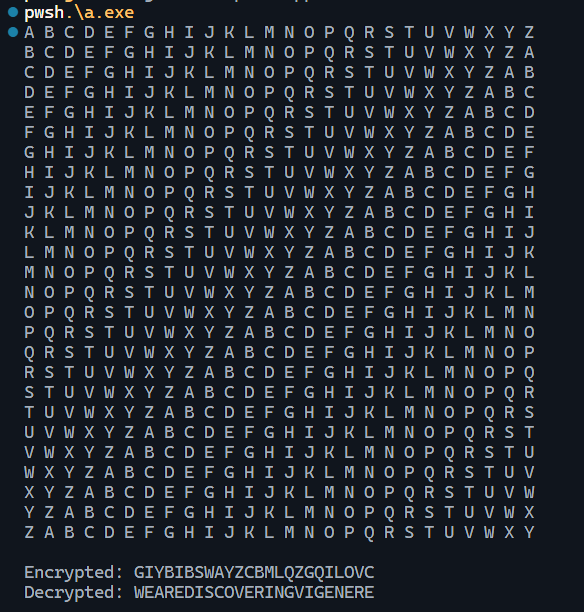
    string decrypted = vigenereDecrypt(encrypted, key);

    cout << "Decrypted: " << decrypted << endl;

    return 0;

}

**Output :**

****

**Variation in Code:**

#include <bits/stdc++.h>

using namespace std;

char vigenereTable[26][26];

void initVigenereTable() {

    for (int i = 0; i < 26; i++) {

        for (int j = 0; j < 26; j++) {

            vigenereTable[i][j] = 'A' + (i + j) % 26;

        }

    }

}

string generateKey(const string &text, const string &key) {

    string newKey = key;

    while (newKey.size() < text.size())

        newKey += key;

    return newKey.substr(0, text.size());

}

string vigenereEncrypt(const string &plaintext, const string &key) {

    string newKey = generateKey(plaintext, key);

    string cipher = "";

    int dynamicShift = 0; // Changes after each character

    for (int i = 0; i < plaintext.size(); i++) {

        char p = toupper(plaintext[i]);

        char k = toupper(newKey[i]);

        if (isalpha(p)) {

            int row = (k - 'A' + dynamicShift + i) % 26;

            int col = p - 'A';

            char enc = vigenereTable[row][col];

            cipher += isupper(plaintext[i]) ? enc : tolower(enc);

            // Dynamically update the shift according to the encrypted character.

            dynamicShift = (dynamicShift + (enc - 'A')) % 26;

        } else {

            cipher += plaintext[i];

        }

    }

    return cipher;

}

// Decrypt with variation

string vigenereDecrypt(const string &ciphertext, const string &key) {

    string newKey = generateKey(ciphertext, key);

    string plain = "";

    int dynamicShift = 0;

    for (int i = 0; i < ciphertext.size(); i++) {

        char c = toupper(ciphertext[i]);

        char k = toupper(newKey[i]);

        if (isalpha(c)) {

            int row = (k - 'A' + dynamicShift + i) % 26;

            // Find the column in the table

            int col = 0;

            while (col < 26 && vigenereTable[row][col] != c) col++;

            char dec = 'A' + col;

            plain += isupper(ciphertext[i]) ? dec : tolower(dec);

            dynamicShift = (dynamicShift + (c - 'A')) % 26;

        } else {

            plain += ciphertext[i];

        }

    }

    return plain;

}

int main() {

    initVigenereTable();

    string plaintext = "WEAREDISCOVERINGVIGENERE";

    string key = "KEY";

    string encrypted = vigenereEncrypt(plaintext, key);

    cout << "Encrypted: " << encrypted << endl;

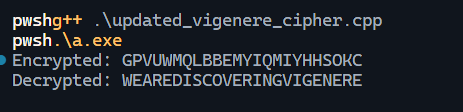
    string decrypted = vigenereDecrypt(encrypted, key);

    cout << "Decrypted: " << decrypted << endl;

    return 0;

}

**Output:**

****

**Security Analysis :**

1. **Standard Vigenère Cipher**

**Strengths**

* **Better than Caesar Cipher**: Multiple Caesar shifts depending on key letters.
* **Simple and symmetric**: Encryption/Decryption works correctly with same key.
* **Resistant to brute force (short term)**: Larger keyspace compared to Caesar cipher.

**Weaknesses**

1. **Key Repetition**
   * The key is repeated (generateKey()), so if key is small compared to message length → **patterns repeat**.
   * Example: Key "KEY" repeats every 3 chars.
2. **Kasiski Examination & Frequency Analysis**
   * Attackers can guess the key length using repeated ciphertext segments.
   * Once length is known, each "column" of ciphertext is just a Caesar shift → cracked easily.
3. **Small Keyspace**
   * If the key length is short (say 3–5), brute force is **computationally trivial**.
4. **Not Secure Against Known-Plaintext Attack**
   * If an attacker knows part of the plaintext (like "WEARE" → "CIPHERTEXT"), they can recover parts of the key directly.

**Conclusion:** **Standard Vigenère is only a polyalphabetic substitution cipher, not secure by modern standards**.

1. **Variation with Dynamic Shift**

This code modifies Vigenère by introducing **stateful shifting**:

int row = (k - 'A' + dynamicShift + i) % 26;

dynamicShift = (dynamicShift + (enc - 'A')) % 26;

**Security Impact**

1. **Dynamic State Increases Entropy**
   * Each character depends not only on the key, but also on the **previous ciphertext** and **position i**.
   * This prevents simple repetition cycles like standard Vigenère.
2. **Resists Kasiski Attack**
   * Classical "key length detection" is useless because the ciphertext shifts are not purely periodic.
3. **Ciphertext Feedback (like CFB in block ciphers)**
   * The next character's encryption depends on the previous ciphertext.
   * This makes ciphertexts more random-like and **harder to frequency-analyze**.
4. **Still Symmetric**
   * Decryption works correctly since it mirrors the same dynamic shift logic.

**Weaknesses**

1. **Still a Substitution Cipher**
   * Even though dynamic, it's still limited to **26 possible outputs per letter**.
   * With large enough ciphertext, **statistical analysis** could still reveal weaknesses.
2. **Deterministic with Same Key & Plaintext**
   * Same plaintext + same key = same ciphertext (no salt/IV).
   * Makes it vulnerable to **replay attacks** and **ciphertext comparison**.
3. **Chosen-Plaintext Attack**
   * If attacker can encrypt controlled messages, they can reverse-engineer how dynamicShift evolves and break the scheme.
4. **Not Cryptographically Secure**
   * Modern encryption requires strong diffusion, avalanche effect, large keyspace (128–256 bits).
   * This scheme is still within classical cipher territory.

**Conclusion:** **Stronger than classic Vigenère** (prevents periodic key weaknesses), but still **not secure against modern attacks**.