Inhaltsverzeichnis

1. Beispiel: Kryptografie

1.1. Ziele

☑ Verschiedene Algorithmen zu Kryptografie kennen lernen

☑ Objekt-Orientierung, Vererbung anwenden können

1.2. Projekt: oop1-Cipher – Die Klasse Cipher

Erstellen Sie die abstrakte Klasse Cipher:

☑ cipher.h /cipher.cpp

```
class Cipher{
     public:
          string encrypt(string s, int key);
          string decrypt(string s, int key);

          // Die Unterklassen müssen diese Methoden implementieren
          virtual char encrypt(char ch, int key)=0;
          virtual char decrypt(char ch, int key)=0;
};
```

1.3. Projekt: oop1-Cipher – Die Klasse XORSubstitution

Die einfachste Substitution ist die Xor- Verknüpfung c^key jedes Text-Bytes c mit dem Schlüssel key. Die Entschlüsselung ist identisch mit der Verschlüsselung.

Schreiben Sie eine CPP-Klasse XORSubstitution, die von Cipher abgeleitet ist.
Hier lässt sich der CPP-Operator ^ verwenden, der die 32 Bits von zwei int-Werten mit Xor verknüpft und als Ergebnis einen neuen int-Wert liefert.

Das folgende Testprogramm erwartet einen Klartext plainText als erstes und einen Schlüssel key als zweites Kommandozeilenargument.
Es erzeugt einen Verschlüssler cipher und chiffriert damit den Klartext in den Geheimtext cryptText:

Hinweise:
#include <string>
string s(argv[0]); // Strings erzeugen
char ch= s[0]; // Elementweiser Zugriff

string s, t;
...
s == t // strings vergleichen

☑ Dateien:
- ☐ cipher.h,cipher.cpp
- ☐ xorsubstitution.h, xorsubstitution.cpp

☑ test-xorsubstitution.cpp

```
#include <string>

#include "xorsubstitution.h"


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

int main (int argc, char* argv[]) {
     string plainText;
     int key;

     plainText = string(argv[1]);
     sscanf(argv[2], "%i", &key);

     Cipher* cipher = new XORSubstitution();
     string cryptText = cipher->encrypt(plainText, key);

/*
Die Bytes des verschlüsselten Textes werden auf der Konsole gezeigt. In
der Regel kann dieser String nicht lesbar ausgegeben werden, weil bei
der Verschlüsselung nichtdruckbare Zeichen entstehen.
*/
     cout << "[ ";
     for (int i=0; i< cryptText.size() -1; i++){
          cout << (int) cryptText[i] << "," ;
     }
     cout << (int) cryptText[cryptText.size() -1];
     cout << " ]" << endl;


/*
Schließlich wird der Geheimtext wieder dechiffriert und zur Kontrolle
mit dem ursprünglichen Klartext verglichen. Die Ausgabe müsste true
lauten:
*/
     string decoded = cipher->decrypt(cryptText, key);
```

```
     if (plainText == decoded)
          cout << "true" <<endl;
     else
          cout << "false" <<endl;


     delete cipher;

     return 0;
}

/*
Übersetzen:
g++ test-xorsubstitution.cpp xorsubstitution.cpp cipher.cpp -o test-
xorsubstitution.exe

Das Programm kann zum Beispiel folgendermaßen aufgerufen werden.

$ ./test-xorsubstitution.exe "Hello, world!" 23
[95, 114, 123, 123, 120, 59, 55, 96, 120, 101, 123, 115, 54]
true
*/
```

1.3.1. Projekt: oop1-Cipher – Die Klasse ADDSubstitution

Bei der additiven Substitution wird der Code jedes Zeichens benutzt und der Wert key dazu addiert. Überlauf kann ignoriert werden, denn er führt beim Typecast auf char zu einem Vorzeichenwechsel, der bei der Subtraktion während der Entschlüsselung automatisch kompensiert wird.

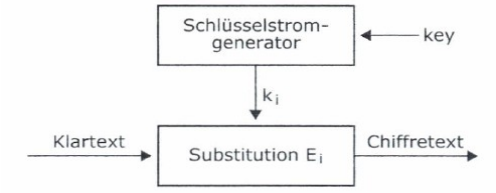
☑ Dateien:
- ☐ cipher.h,cipher.cpp
- ☐ addsubstitution.h, addsubstitution.cpp
- ☐ test-addsubstitution.cpp

a) Schreiben Sie eine Cpp-Klasse ADDSubstitution, die von Cipher abgeleitet ist.
b) Schreiben Sie analog zu oben ein Testprogramm: test-addsubstitution.cpp

1.4. Stromchiffren

Stromchiffren stützen sich ebenfalls auf eine Substitution. Sie verwenden aber einen **neuen Substitutionsschlüssel für jedes Zeichen** des Textes.

Gleiche Klartextzeichen werden somit auf verschiedene Chiffretextzeichen abgebildet. Dadurch werden die Chiffretextzeichen ausgewogener verteilt und eine statistische Kryptoanalyse erschwert. Die einfachste Stromchiffre verwendet einen Schlüsselstromgenerator.



Der Schlüsselstromgenerator kann zB. ein Zufallsgenerator sein, der unterschiedliche Zahlen/Keys liefert, die zB. Mittels XOR-Verknüpfung zu verschiedenen Chiffretextzeichen führen.

Zur Entschlüsselung muss allerdings der gleiche Zufallszahlengenerator verwendet werden. Dies erreicht man durch das Initialisieren des Zufallszahlengenerators.

C:
```
#include <stdlib.h>
srand(key);
```

Java:
```
java.util.Random keygenerator= new  java.util.Random();
keygenerator.setSeed(key);
```

1.4.1. Projekt: oop1-Cipher – Die Klasse StreamCipher

Implementieren Sie eine von Cipher::XORSubstitution abgeleitete Klasse StreamCipher, die die Zeichen Xor-verschlüsselt.

Java:
Verwenden Sie als Schlüsselstromgenerator den Zufallszahlengenerator,
der in der Bibliotheksklasse java.util.Random implementiert ist.

C/CPP:
```
#include <stdlib.h>
srand(key);
```

☑ Dateien:
- ☐ cipher.h,cipher.cpp, xorsubstitution.h, xorsubstitution.cpp
- ☐ streamcipher.h, streamcipher.cpp
- ☐ test-streamcipher.cpp

Das Testprogramm von oben arbeitet auch mit einem StreamCipher-Objekt, das mit

```
StreamCipher* streamCipher = new StreamCipher();
```

der Variablen cipher zugewiesen wird.

Das Programm gibt dann Folgendes aus:

```
$ ./test-streamcipher.exe "Hello, world!" 23
[47, 124, 106, -109, -54, -37, 68, -101, -28, -60, -31, 74, 82]
true
```

Der dritte und vierte Buchstabe des Klartextes sind gleich (l). Im Geheimtext entstehen daraus zwei verschiedene Zeichen (Codes 106 und -109).

Hinweise:
Die zeichenweise XOR-Verschlüsselung wird von der Oberklasse XORSubstitution geerbt.

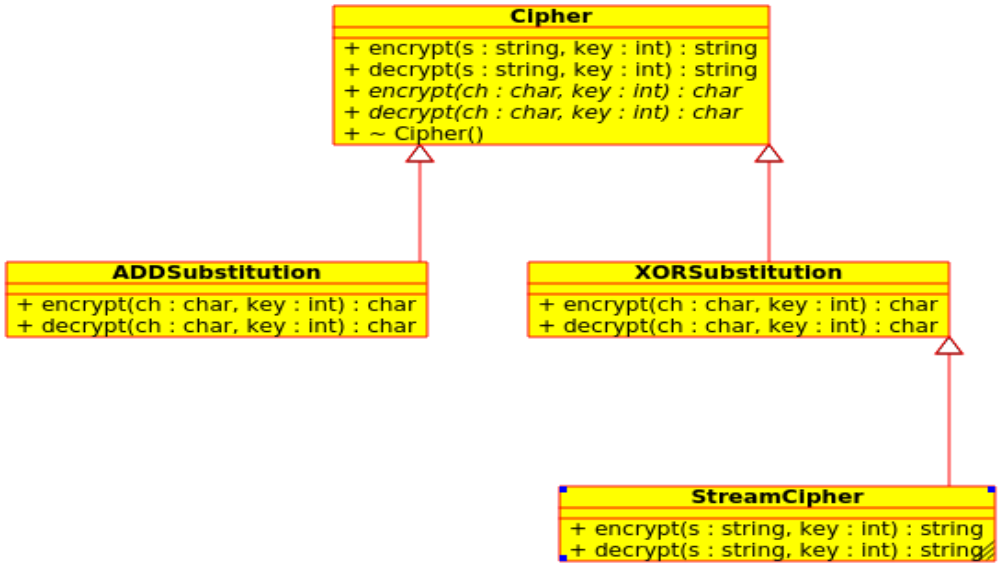
Die von Cipher geerbte stringweise Streamverschlüsselung/-entschlüsselung muss überschrieben werden, weil die darin aufgerufene zeichenweise Verschlüsselung
char encrypt(char ch, int key)

immer mit einem anderen key aufgerufen werden muss.

```
string  StreamCipher::encrypt(string s, int key){
        // Schlüsselstromgenerator
        ...
        // XOR-Verschlüsselung verwenden
        for (int i=0; i < s.size(); i++){
                // neuen Key erzeugen
                ....

                s[i]= zeichenweise Verschlüsselung der Oberklasse mit neuem key ;
        }
        ....
}
```

1.5. Zusammenfassung: Klassendiagram



1.6. Diffie-Hellman

Siehe auch: http://ddi.uni-wuppertal.de/material/spioncamp/dl/austausch-diffie-hellman-station2.pdf

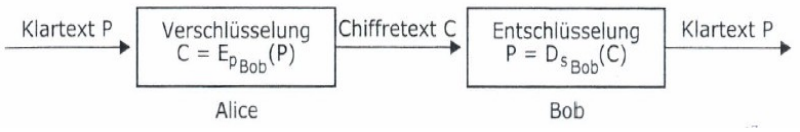
Um Daten zwischen 2 Kommunikationsteilnehmern sicher(verschlüsselt) austauschen zu können, muss zuallererst der Schlüssel ausgetauscht werden. Was selbst schon wieder ein Sicherheitsrisiko darstellt.

Asymmetrische Verfahren
Dieses Problem des Schlüsselaustausches kann aber durch ein sog. Asymmetrisches Verfahren gelöst werden.

Public key p, secret key s
Jeder Kommunikationsteilnehmer generiert sich ein Schlüsselpaar (p, s) mit einem öffentlichen Schlüssel p (public key) und einem geheimen Schlüssel s (secret key).

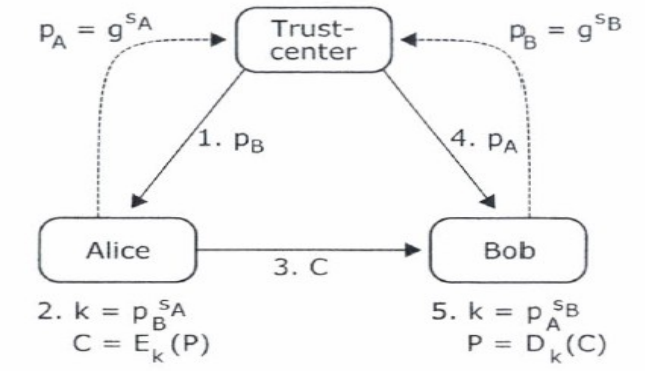
p und s sind invers
Eine Nachricht, die mit p verschlüsselt wurde, lässt sich nur mit s wieder entschlüsseln und umgekehrt. p und s sind invers zueinander.

Encrypt mit p und decrypt mit s
Will Alice einen Liebesbrief P (plaintext) an Bob schicken, den kein anderer lesen soll, so verschlüsselt sie ihn mit Bobs öffentlichem Schlüssel p_{Bob}. Nur Bob ist im Besitz des zugehörigen geheimen Schlüssels s_{Bob} und kann damit Alices verschlüsselten Brief C (ciphertext) entschlüsseln.



Geheimen Schlüssel austauschen
Asymmetrische Verfahren können auch verwendet werden, um damit über eine ungeschützte Verbindung einen geheimen symmetrischen Schlüssel zu vereinbaren, der dann zur Verschlüsselung einer vertraulichen Nachricht verwendet wird.
Symmetrische Verfahren sind schneller bei größeren Datenmengen.

Ein solches »**Schlüsselvereinbarungsprotokoll**« geht auf Whitfield **Diffie** und Martin **Hellman** zurück.



Trustcenter liefert n, g (zur Berechnung des public key)
Ein Trustcenter veröffentlicht eine große Primzahl n und dazu eine Zahl g (Generator) mit der Eigenschaft, dass die Potenzen von g modulo n alle Zahlen zwischen 1 und n-1 durchlaufen.

Public key p beim Trustcenter registrieren:
Jeder Kommunikationsteilnehmer

- wählt eine **geheime Zahl s**,
- berechnet dazu **$p = g^s \bmod n$** und schickt p zur Veröffentlichung an das Trustcenter.

Alice to Bob
Will nun Alice eine vertrauliche Nachricht an Bob schicken, so geht sie folgendermaßen vor:

1. Alice holt sich vom Trustcenter Bobs öffentlichen Schlüssel $\mathbf{P_{Bob}}$.
2. Alice berechnet einen symmetrischen Schlüssel $\mathbf{k = p_{Bob}^{sAlice} \bmod n}$ und
3. Alice verschlüsselt damit ihre Nachricht P zu **C = encrypt(P, k)**.
   Dazu kann eine beliebige symmetrische Chiffre, beispielsweise eine Stromchiffre, benutzt werden.
4. Alice schickt den chiffrierten Text C an Bob.
5. Bob holt sich vom Trustcenter Alices öffentlichen Schlüssel p_{Alice}.
6. Bob berechnet $\mathbf{k = p_{Alice}^{sBob} \bmod n}$ und
7. Bob entschlüsselt damit den Chiffretext C zu **P = decrypt(C,k)**, der ursprünglichen Nachricht.

Beachte:
Die von Bob und Alice unabhängig berechneten Schlüssel **k sind gleich**!

Alice: $\mathbf{k = p_{Bob}^{sAlice} \bmod n}$
Bob: $\mathbf{k = p_{Alice}^{sBob} \bmod n}$

Es gilt allg: $p = g^s \bmod n$

$\mathbf{p_{Alice} = g^{sAlice} \bmod n}$ bzw. $\mathbf{p_{Bob} = g^{sBob} \bmod n}$
Bob: $k = p_{Alice}^{sBob} \bmod n = (g^{sAlice} \bmod n)^{sBob} \bmod n = \mathbf{g^{sAlice * sBob} \bmod n}$
Alice: $k = p_{Bob}^{sAlice} \bmod n = (g^{sBob} \bmod n)^{sAlice} \bmod n = \mathbf{g^{sBob * sAlice} \bmod n}$

1.6.1. Projekt: oop1-Cipher – Die Klasse DHCipher

Definieren Sie eine Klasse DHCipher.

Der Konstruktor erhält als Parameter n (Primzahl) und g.
Er erzeugt einen zufälligen geheimen Schlüssel s (Random) (mit $1 < s < n-1$)und berechnet daraus den zugehörigen öffentlichen Schlüssel $p = g^s \bmod n$.

Der geheime Schlüssel bleibt selbstverständlich unter Verschluss, der öffentliche Schlüssel wird dagegen über den Getter getPublicKey() allgemein zur Verfügung gestellt.

getPublicKey() spielt die Rolle des Trustcenters.

Die Methoden encrypt(text, publicKey) und decrypt(text, publicKey) berechnen aus dem übergebenen öffentlichen Schlüssel p des Partners und dem eigenen geheimen Schlüssel s den gemeinsamen geheimen symmetrischen Schlüssel
$k = p^s \bmod n$.

Dieser dient zur Initialisierung der Stromchiffre (s.o.), mit der der übergebene Text ver- beziehungsweise entschlüsselt wird.

Das nachfolgende Testprogramm holt sich den Klartext und g und n von der Kommandozeile. Zur Vereinfachung kann auch mit festen Werten gearbeitet werden, wie zum Beispiel n = 23 und g = 5.

```
...
     cout << endl << "*** DIFFI-HELLMAN *** " << endl;
     int n=5;
     int g=23;
```

Anschließend erzeugt das Programm zwei DHCipher-Objekte alice und bob, die mit den Daten initialisiert werden, die vom Trustcenter stammen würden:

```
     DHCipher* alice = new DHCipher(n, g);
     DHCipher* bob = new DHCipher(n, g);
```

Bobs öffentlicher Schlüssel wird an Alice übergeben, die damit den Klartext zu cryptText verschlüsselt:

```
     cryptText = alice->encrypt(plainText, bob->getPublicKey());
     cout << "[ ";
     for (int i=0; i< cryptText.size() -1; i++){
          cout << (int) cryptText[i] << "," ;
     }
     cout << (int) cryptText[cryptText.size() -1];
     cout << " ]" << endl;
```

Bob entschlüsselt cryptText mithilfe von Alices öffentlichem Schlüssel. Schließlich wird verifiziert, dass die ursprüngliche Nachricht korrekt zurückgewonnen wurde:

```
     decoded = bob->decrypt(cryptText, alice->getPublicKey());

     if (plainText == decoded)
          cout << "true" <<endl;
     else
          cout << "false" <<endl;

...
```

Das Testprogramm gibt je nach zufällig gewählten Schlüsseln verschiedene Zahlenfolgen aus, beispielsweise:
$ t-DHCipher.exe "Hello, world!" 23
[50, -36, -105, -65, -60, -126, -78, -79, 76, -62, -72, -lll, 42]
true

Hier nun die Klasse DHCipher

```
class DHCipher{
     private:
          int secretKey;
          int publicKey;
          int n;
     public:
          DHCipher(int n, int g);
          int getPublicKey();
          int sharedKey(int otherPublicKey);  //k
```

```
            string encrypt(string s, int key);
            string decrypt(string s, int key);
};
```