**Đại học Quốc gia Thành phố Hồ Chí Minh**
**Trường Đại học Công nghệ Thông tin (UIT)**

# BÁO CÁO BÀI TẬP

**Môn học: Mật mã học**

**Kỳ báo cáo: Buổi 01 (Session 02)**

**Tên chủ đề: Thuật toán mã hoá DES**

*Ngày báo cáo: 14/03/2023*

## 1. THÔNG TIN CHUNG:

*(Liệt kê tất cả các thành viên trong nhóm)*

Lớp: **NT219.N21.ANTN**

| STT | Họ và tên | MSSV | Email |
|---|---|---|---|
| 1 | Phạm Nguyễn Hải Anh | 21502586 | 21520586@gm.uit.edu.vn |
| 2 | Nguyễn Nhật Quân | 21522497 | 21522497@gm.uit.edu.vn |

## 2. NỘI DUNG THỰC HIỆN:[1]

| STT | Công việc | Kết quả tự đánh giá | Người đóng góp |
|---|---|---|---|
| 1 | Câu hỏi 01 | 100% | Nhật Quân |
| 2 | Câu hỏi 02 | 60% | Hải Anh |
| 3 | Câu hỏi 03 | | Nhật Quân |
| 4 | Câu hỏi 04 | 20% | Hải Anh |
| 5 | Câu hỏi 05 | 20% | Hải Anh |
| | Bài luyện tập 1 | 60% | Hải Anh, Nhật Quân |
| | | | |
| | Bài luyện tập 3 | 20% | Hải Anh |
| | | | |

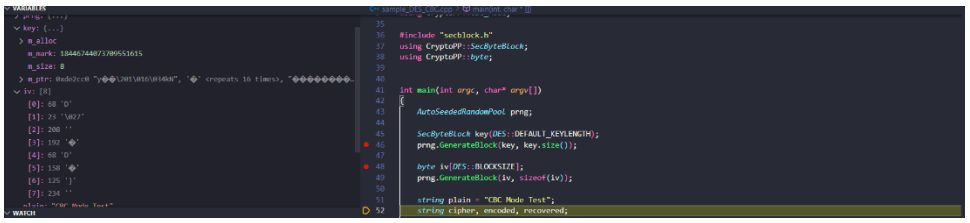
**Phần bên dưới của báo cáo này là tài liệu báo cáo chi tiết của nhóm thực hiện.**

[1] Ghi nội dung công việc, các kịch bản trong bài Thực hành

# BÁO CÁO CHI TIẾT

## 1. Exercise 1 (File sample_DES_CBC.cpp):

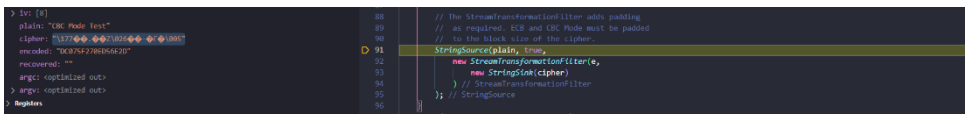
- generate 'key' and 'iv' variables by randomization using GenerateBlock() function, we get the result as string bytes, but the data type of cryptpp library is difficult to read.



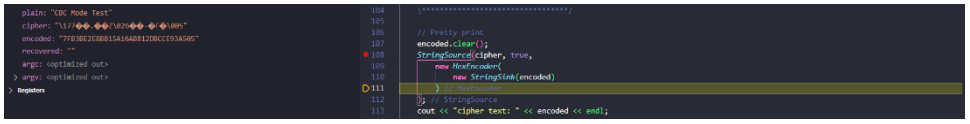

- then, to make it clear to the viewer, we convert the 'key' and 'iv' variables to a Hex type storage in 'encoded' variable and print it

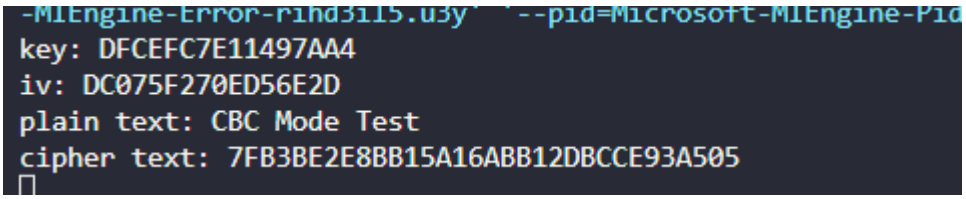

- then use CBC_Mode with 'key' and 'iv' , using SetKeyWithIV() function to set key and iv to 'e' variable


- encrypting 'plain' via StringSource(), StreamTransformationFilter() functions with 'e' parameter, then we get 'cipher' variable which is encrypted




- in order to be able to easily read the information of the cryptographic variable, we encode the 'cipher' into Hex and save it in the 'encoded' variable and then print 'encoded'




-MIEngine-Error-rihd3115.u3y --pid=Microsoft-MIEngine-Pid
key: DFCEFC7E11497AA4
iv: DC075F270ED56E2D
plain text: CBC Mode Test
cipher text: 7FB3BE2E8BB15A16ABB12DBCCE93A505

## 2. Exercise 2 (File bt2.cpp):

Main activities of the DES decryption with CBC mode:

(I do not know how to Debugging with vscode, so instead I explain the logic of the decryiption.) Because each time I run code, different permutation occurs, so in this scenario, I suppose the receiver receives the key = $12345678_{ASCII}$ (this number won't have correct parity bits) and iv = $abcdabcd_{ASCII}$ and ciphertext.

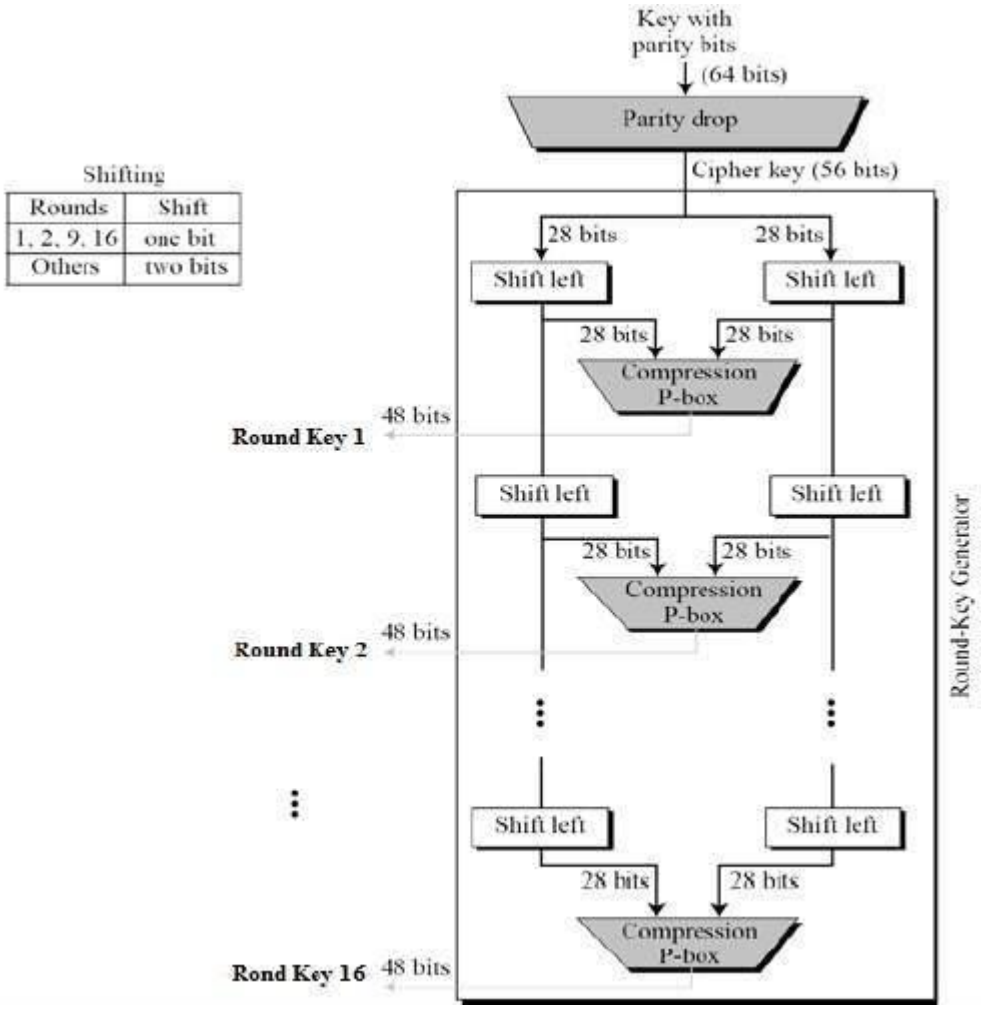
key = 00110001 00110010 00110011 00110100 00110101 00110110 00110111 00111000

iv = 01100001 01100010 01100011 01100100 01100001 01100010 01100011 01100100

ciphertext = 64 66 64 73 61 77 72 65 77 66 $61_{hex}$ = 1100100011001100110010001110011011000010111011101110010011001010 1110111011001100100001

In the decryption, the ciphertext will be separated into 64-bit blocks, and 16 subkeys (RoundKey) will be generated as the same way in the encryption. And the $16^{th}$ subkey will be used for the first round of decryption. It is reverse order to the encryption, which uses the $1^{st}$ subkey for encrypting.

Key generating process:




Key generation process

At first, all 8x bits (8, 16, 24, 32, …, 64) will be dropped, or used for parity. Then PC-1: 64 – 8 = 56 bits left will be permutated, then divided into 2 halves (L & R), each with 28 bits:

(There is a rule for the permutation for this step, however, for convenience, I suppose the key does not change after permutated)

Key = 0011000 0011001 0011001 0011010 | 0011010 0011011 0011011 0011100

L = 0011000 0011001 0011001 0011010

R = 0011010 0011011 0011011 0011100

Then rotate left for both L and R. Round 1 will shift left 1 bit:

Shifting

| Rounds | Shift |
|---|---|
| 1, 2, 9, 16 | one bit |
| Others | two bits |

shift left bits convention

L = 011000 0011001 0011001 0011010 0

R = 011010 0011011 0011011 0011100 0

Then PC-2 (like PC-1) but only 48 bits left from both L and R. They are considered as the first 48-bit subkey, however, used in the last decryption round.

Assume first subkey is

011000 0011001 0011001 0011010 0 011010 0011011 0011011

Continue with the second subkey with the same PC-1, PC-2 and same last L and R.

…

Assume the 16th subkey is

011010 0011011 0011011 0011100 0011000 0011001 0011001

Decryption process:

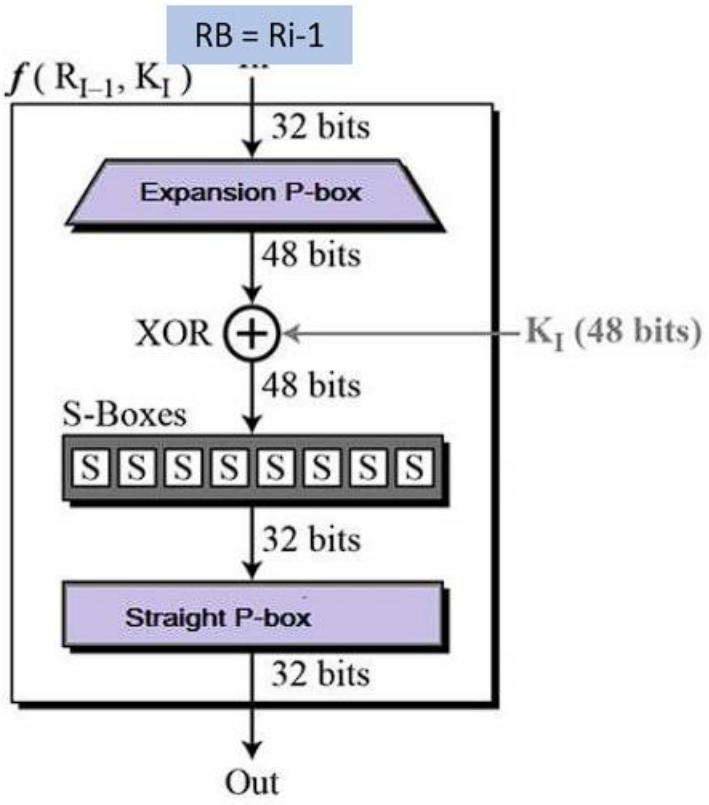
The first 64-bit block of ciphertext is:

1100100011001100110010001110011011000010111011101110010011001010

The Initial Permutation (IP) and Final Permutation (FP, which inverses the IP) is just used to support the hardware, so I assume that after IP (FP) the block does not change, for convenience. The 64-bit block will be divided into 2 32-bit blocks, LB & RB.

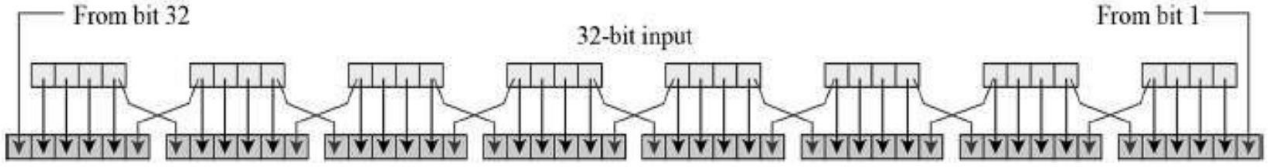
LB: 11001000110011001100100011100110

RB: 11000010111011101110010011001010




The F function

To XOR with 48-bit 16th subkey, 32-bit RB needs to be 48-bit, by using permutation method (Expansion Permutation) to add extra bits.

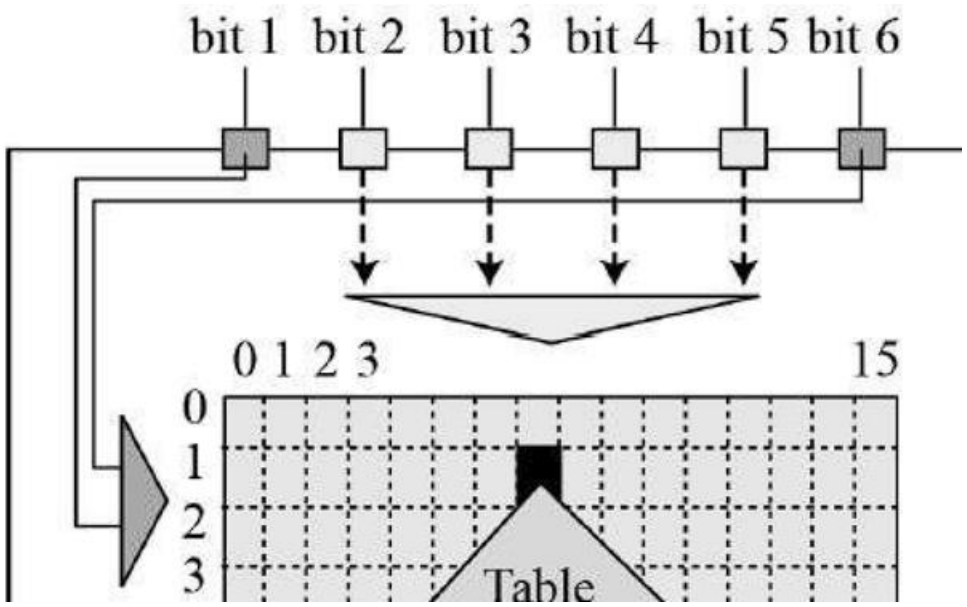



| | | | | | |
|---|---|---|---|---|---|
| 32 | 01 | 02 | 03 | 04 | 05 |
| 04 | 05 | 06 | 07 | 08 | 09 |
| 08 | 09 | 10 | 11 | 12 | 13 |
| 12 | 13 | 14 | 15 | 16 | 17 |
| 16 | 17 | 18 | 19 | 20 | 21 |
| 20 | 21 | 22 | 23 | 24 | 25 |
| 24 | 25 | 26 | 27 | 28 | 29 |
| 28 | 29 | 31 | 31 | 32 | 01 |

Expansion Permutation

The value after XOR will be divided into 6-bit groups. Each group is 6 inputs of a S-box, which has 4 outputs.

| $S_5$ | | Middle 4 bits of input | | | | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | | **0000** | **0001** | **0010** | **0011** | **0100** | **0101** | **0110** | **0111** | **1000** | **1001** | **1010** | **1011** | **1100** | **1101** | **1110** | **1111** |
| **Outer bits** | **00** | 0010 | 1100 | 0100 | 0001 | 0111 | 1010 | 1011 | 0110 | 1000 | 0101 | 0011 | 1111 | 1101 | 0000 | 1110 | 1001 |
| | **01** | 1110 | 1011 | 0010 | 1100 | 0100 | 0111 | 1101 | 0001 | 0101 | 0000 | 1111 | 1010 | 0011 | 1001 | 1000 | 0110 |
| | **10** | 0100 | 0010 | 0001 | 1011 | 1010 | 1101 | 0111 | 1000 | 1111 | 1001 | 1100 | 0101 | 0110 | 0011 | 0000 | 1110 |
| | **11** | 1011 | 1000 | 1100 | 0111 | 0001 | 1110 | 0010 | 1101 | 0110 | 1111 | 0000 | 1001 | 1010 | 0100 | 0101 | 0011 |



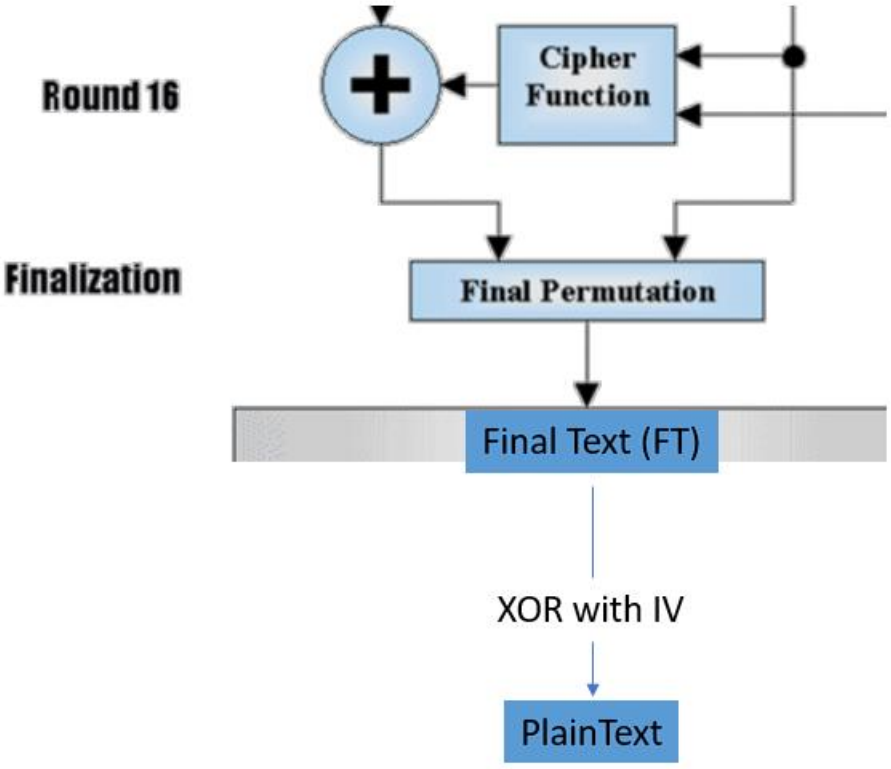

DES S-box logic table

For example a 6-bit input is 011011" has lsb 1, msb 0 and inner bits "1101" then output is "1001".

With 8 S-box then we have 8 . 4 = 32 bits, those will be RB of Round 2. Meanwhile, the RB of Round 1:

11000010111011101110010011001010

will be used as LB of Round 2. At the Round 16, LB and RB of that round will be merged to form the final text (FT). The first 64-bit plaintext block is the result of FT XOR IV.




First PlainText block is formed

The second 64-bit block of cipher is the ciphertext bits that are not used yet ( 111011101100110011000001), following by the padding bits (assume a padding bit is 0). The first 64-bit block will be XOR with the FT of the second decryption, instead of IV like the first decryption, to form the second plaintext block.

Now is the logic of code by debugging:

At line 45:

```
44    SecByteBlock key(DES::DEFAULT_KEYLENGTH);
45    prng.GenerateBlock(key, key.size());
```

The key is generate by GenerateBlock function with size equals to key.size:

```
key: {...}
  m_alloc
    CryptoPP::AllocatorBase<unsigned char> (base):
  m_mark: 18446744073709551615
  m_size: 8
  m_ptr: 0xe229d0 "\r\r", '<repeats 16 times>,
    *m_ptr: 13 '\r'
```

Key's length is 8 bytes.

Next, at line 48:

```
47    CryptoPP::byte iv[DES::BLOCKSIZE];
48    prng.GenerateBlock(iv, sizeof(iv));
```

IV is generate, with the length of 8 bytes:

```
iv: [8]
  [0]: 46 '.'
  [1]: 134 '?'
  [2]: 147 '?'
  [3]: 64 '@'
  [4]: 222 ''
  [5]: 25 '\031'
  [6]: 116 't'
  [7]: 166 '?'
```

At line 78:

```
77    CBC_Mode< DES >::Decryption d;
78    d.SetKeyWithIV( key, key.size(), iv );
```

Decryption function class is generated, whose properties are dependent on key and iv. We can see that it repeats 16 times, because DES has 16 rounds:

```
d: {...}
 CryptoPP::ObjectHolder<CryptoPP::BlockCipherFinal<(CryptoPP::CipherDir)1,
  m_object
 CryptoPP::AlgorithmImpl<CryptoPP::CBC_Decryption, CryptoPP::CipherModeFina
  CryptoPP::CBC_Decryption (base): CryptoPP::CBC_Decryption
   CryptoPP::CBC_ModeBase (base): CryptoPP::CBC_ModeBase
   m_temp
    m_alloc
     m_mark: 18446744073709551615
     m_size: 8
     m_ptr: 0x674910 "\r\r", '?' <repeats 16 times>, "?????????"
      *m_ptr: 13 '\r'
```

Note that *m_ptr is 13 as in the key

```
m_ptr: 0x674910 "\r\r", '?' <repeats 16 times>, "?????????"
  *m_ptr: 13 '\r'
```

At line 84: I do not really understand how it is debugged:

```
81      StringSource ss2(
82          cipher,
83          true,
84          new StreamTransformationFilter(
85              d,
86              // StringSink là 1 hàm sinh ra chuỗ
87              new StringSink(recovered)
```

StringSource will take cipher as the input and decrypt it by using the StreamTransformationFilter object. The StreamTransformationFilter will base on the type of d, which is currently DES_CBC, and use d's key and iv to decrypt into the recorvered.

### 3. Exercise 3 (file bt3.cpp):

Use some libraries like this:

```
#include <locale>
#include <codecvt>
#include <string>
#include <io.h>
#include <fcntl.h>
#include <stdio.h>
```

The main idea is using the 'wstring' datatype which supports UTF-16 for input then converting 'wstring' to 'string' to encrypt the plaintext. The last step is converting the 'string' recovered text decrypted from ciphertext to 'wstring' to display UTF-16 output.

Now, let's go through the details:

a. The first thing we need is declaring the variables

```
std::wstring wplain = L"你好";
std::wstring_convert<std::codecvt_utf8<wchar_t>> converter;
std::string plain = converter.to_bytes(wplain);


string cipher, encoded, recovered;
std::wstring wencoded;
```

We have both wstring and string datatype. To convert wstring to string datatype, I have used 'wstring_convert' and 'codecvt_utf8' class to create converter. After using it, I have had 'plain' string datatype converted from wstring

b. Then, we just use the DES algorithm to encrypt the plaintext.
c. In addition, we need to use _setmode() funtion with 'stdout' mode and 'std::wcout' to display our UTF-16 paintext.

```
_setmode(_fileno(stdout), _O_U16TEXT); // <=== Windows madness
try
{
    std::wcout << "plain text: " << wplain << std::endl;
```

d. At the last step, I used 'wstring_convert' and 'codecvt_utf8_utf16' class to create converter1 and used it to convert cipher text in 'encoded' variable and recovered text to UTF-16 and display them with wcout.

```
); // StringSource
std::wstring_convert<std::codecvt_utf8_utf16<wchar_t>> converter1;
wencoded = converter1.from_bytes(encoded);

std::wcout << "cipher text: " << wencoded << endl;
```

```
    std::wstring wrecovered = converter1.from_bytes(recovered);

    std::wcout << "recovered text: " << wrecovered << endl;
```

(anything printed by 'cout' below _setmode() function can not be displayed on terminal so that I have to also convert ciphertext to UTF-16 and use 'wcout')

e. And this is a result:

```
quanp  Cryptography_Lab1  488ms  .\sample_DES_CBC.exe
key: 0E4F64A7225846DB
iv: C02370243F343C7F
plain text: 你好
cipher text: AAA6D7904F7B94D6
recovered text: 你好
```

## 4. Exercise 4 (file bt4.cpp):

Change the default plaintext ("CBC Mode Test") into the code below:

```
string plain;
cout << "Nhập plaintext: ";
std::getline(std::cin, plain);
```

In my gcc (12.2), gdb (13.1), the last input (Enter, or '\n') has been automatically ignored, so it is not necessary to add cin.ignore function.
Other codes are the same.

## 5. Exercise 5 (file bt5.cpp):

Type key: The key is SecByteBlock type, a special Byte array of CryptoPP. Type a string named keystring and use reinterpret_cast to cast to a SecByteBlock object, named key.

```
string keystring;
cout << "Type 8-byte key: ";
std::getline(std::cin, keystring);
// Convert string  to SecByteBlock
// &keystring[0]: the beginning of the data
// keystring.size: the size of the array
SecByteBlock key(reinterpret_cast<const byte*>(&keystring[0]), keystring.size());
```

string to SecByteBlock

This code will not set the keystring length, so I must type 8 characters to fit the SecByteBlock length or an error is thrown out.

Type iv: It is possible to import a value to a byte element with cin. The input value is a char, and compiler will convert automatically to a byte.

```
byte iv[8];
for (size_t i = 0; i < 8; i++){
    cout << i << ": ";
    std::cin >> iv[i];
}
```

```
Type 8-byte key: abcdabcd
0: 1
1: 2
2: 3
3: 4
4: 5
5: 6
6: 7
7: 8
cipher text: ÷oÿ╕vVïe4iö¢¡S,
Recovered text: CBC Mode Test
```

example of import key and iv

The code does not encode cipher in Hex format, so it is not easy to look at.

## 6. Bài tập luyện tập 1:

Thanks to the benchmark_aes_cbc sample from Lab 2, I can now create a benchmark for DES_CBC. From the original code, create the IV:

```
byte iv[DES::BLOCKSIZE];
prng.GenerateBlock(iv, sizeof(iv));
```

and change “CTR_MODE <AES>” to “CBC_MODE <DES>” and change key to iv.

```
CTR_Mode< AES >::Encryption cipher;
cipher.SetKeyWithIV(key, key.size(), key);
```

Situation 1: Data < 64-bit (File Bai_luyen_tap_1_1.cpp)

I refer that “Data” is plaintext. In the code, I regconize variable “buf” is the plaintext:

```
do
{
    blocks *= 2;
    for (; i<blocks; i++)
        cipher.ProcessString(buf, BUF_SIZE);
    elapsedTimeInSeconds = timer.ElapsedTimeAsDouble();
}
while (elapsedTimeInSeconds < runTimeInSeconds);
```

due to the ProcessString function will encrypt it:

### ◆ ProcessString() [1/2]

```
void StreamTransformation::ProcessString ( byte *  inoutString,
                                           size_t  length
                                         )
```

Encrypt or decrypt a string of bytes.

**Parameters**

**inoutString** the string to process

**length** the size of the inoutString, in bytes

now I just need to change the BUF_SIZE to create a random plaintext. Its original value of the code is 2048U, which means 2048 bytes.

```
const int BUF_SIZE = RoundUpToMultipleOf(2048U,
    dynamic_cast<StreamTransformation&>(cipher).OptimalBlockSize());

AlignedSecByteBlock buf(BUF_SIZE);
prng.GenerateBlock(buf, buf.size());
```

now change to 3U, which means 3 bytes = 24 bit.

```
const int BUF_SIZE = RoundUpToMultipleOf(3U,
    dynamic_cast<StreamTransformation&>(cipher).OptimalBlockSize());
```

Then evaluate the DES-CBC algorithm in case data < 64 bit:

```
PS D:\Move\UIT\HK4\MMH\ThucHanh\Lab1\Code> .\bai_luyen_tap_1_1.exe
DES/CBC benchmarks...
  2.7 GHz cpu frequency
  24.9885 cycles per byte (cpb)
  103.044 MiB per second (MiB)
```

Situation 2: UTF-16 Data (File Bai_luyen_tap_1_3.cpp)

To use UTF-16 data with buf, I changed the data type of buf from 'AlignedSecByteBlock' to 'std::u16string'.

```
std::u16string buf(BUF_SIZE, 0); // allocate space for UTF-16 data
```

buf' is allocated as a 'std::u16string' with size 'BUF_SIZE'. The UTF-16 data is then copied into the 'buf' using the 'std::copy function'

```
33    // populate buf with UTF-16 data
34    const char16_t* utf16_data = u"Đây là chuỗi UTF-16";
35    std::copy(utf16_data, utf16_data + std::char_traits<char16_t>::length(utf16_data), buf.begin());
36
```

To pass 'buf' to 'cipher.ProcessString', I cast the 'buf.data()' to a 'byte*'. This can be done using 'reinterpret_cast'

```
    blocks *= 2;
    for (; i<blocks; i++)
        cipher.ProcessString(reinterpret_cast<byte*>(buf.data()), BUF_SIZE);
    elapsedTimeInSeconds = timer.ElapsedTimeAsDouble();
}
```

Then evaluate the DES-CBC algorithm in case UTF-16 data:

```
94.9797 MiB per second (MiB)
quanp  Cryptography_Lab1  45.665s
                                    .\sample_DES_CBC.exe
DES/CBC benchmarks...
  2.7 GHz cpu frequency
  26.0101 cycles per byte (cpb)
  98.997 MiB per second (MiB)
```

Situation 3: Data > 1 MB (File Bai_luyen_tap_1_3.cpp)

1 MB = 1024B. So I change 3U to 1025U

```
const int BUF_SIZE = RoundUpToMultipleOf(1025U,
    dynamic_cast<StreamTransformation&>(cipher).OptimalBlockSize());
```

Then evaluate the DES-CBC algorithm in case data > 1MB:

```
PS D:\Move\UIT\HK4\MMH\ThucHanh\Lab1\Code> .\bai_luyen_tap_1_3.exe
DES/CBC benchmarks...
  2.7 GHz cpu frequency
  26.2763 cycles per byte (cpb)
  97.9941 MiB per second (MiB)
```

## 7. Bài tập luyện tập 3 (file bai_luyen_tap3.cpp):

Reference:
https://www.cs.jhu.edu/~rubin/courses/sp03/papers/voydock.kent.pdf

One weakness of CBC is that if the attacker know the IV, he can implement a chosen plaintext attack.

Scenario: the attacker can catch the transmission between a user X and X's mail box on host H amd even can mail to X. Each uniqe IV and key is used for the session (the paper used the word "association") and not hidden, but all packets use the same IV.

The attacker can:

- Know the IV of the session.

- see the encrypted messages.

Assume:

- Session (or association): A

- IV of the session A: $IV_A$

- first block of the i-th packet: $U_i$

- ciphertext: E(X) with X is the result of the XOR 2 byte strings.

Logic of the attack:

- Because the attacker can know ciphertext, it means he knows $E(IV_A\ xor\ U_i)$

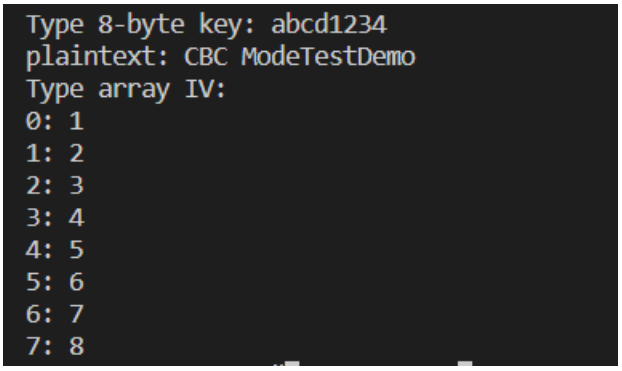
- Because he can send message to user X, it means he can determine the message, including the first patterns $K_i$ that will create the first block the i-th packet, which means he knows $E(IV_A\ xor\ K_i)$ and will find $K_m$ through bruteforce until $E(IV_A\ xor\ K_m) = E(IV_A\ xor\ U_i)$ with all i packets.

- When he has the first plaintext block $U_1$, he can replace IV by $U_1$ to find second, third, … n-th plaintext blocks of the packets he has caught.

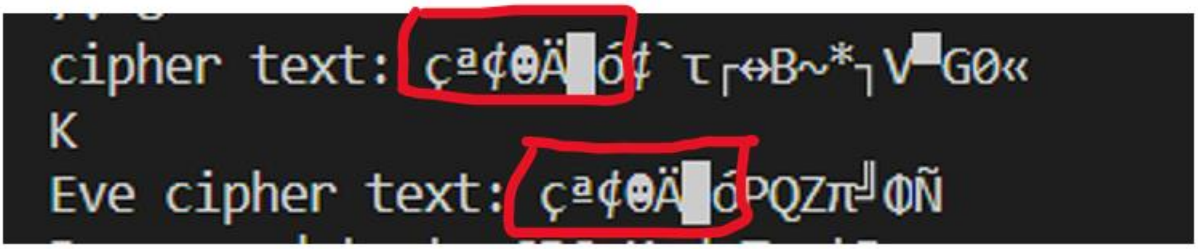
Demo program description:

- IV (var iv), key (var keystring), plaintext (var plain) is input from keyboard. Only one IV and one key are used throughout the program.

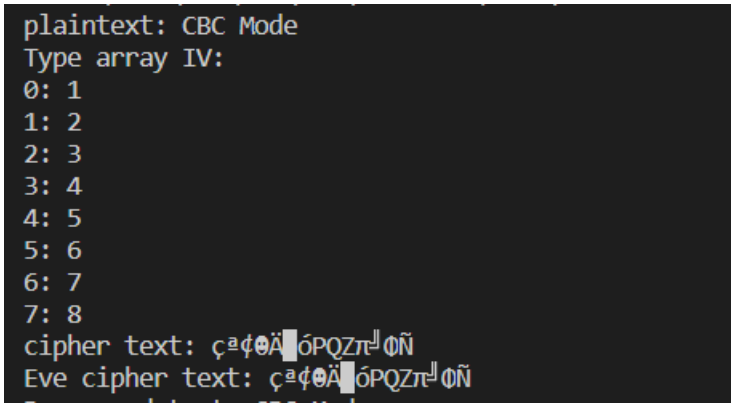
- The pattern $K_1$ (var eP1, which means eavesdropper first Plaintext block) is "CBC Mode".




Now we check whether the encryp message from eP1 has any parts like the original cipher text.




I have no idea why just a part of Eve cipher text is in the original cipher text, because according to the above logic, the whole of it must be the same as a part of the cipher text. Luckily, if the guessing plaintext is as the same as the original plaintext, then 2 ciphertexts is the same:




If the 2 ciphertexts is the same, the eavesdropper knows that he has guessed the right plaintext.

---

***Sinh viên đọc kỹ yêu cầu trình bày bên dưới trang này***

## YÊU CẦU CHUNG

- Sinh viên tìm hiểu và thực hành theo hướng dẫn.
- Nộp báo cáo kết quả chi tiết những việc (**Report**) bạn đã thực hiện, quan sát thấy và kèm ảnh chụp màn hình kết quả (nếu có); giải thích cho quan sát (nếu có).
- Sinh viên báo cáo kết quả thực hiện và nộp bài.

**Báo cáo:**

- File .PDF. Tập trung vào nội dung, không mô tả lý thuyết.
- Nội dung trình bày bằng Font chữ Times New Romans/ hoặc font chữ của mẫu báo cáo này (UTM Neo Sans Intel/UTM Viet Sach)– cỡ chữ 13. Canh đều (Justify) cho văn bản. Canh giữa (Center) cho ảnh chụp.
- Đặt tên theo định dạng: [Mã lớp]-SessionX_GroupY. (trong đó X là Thứ tự buổi Thực hành, Y là số thứ tự Nhóm Thực hành/Tên Cá nhân đã đăng ký với GV).

  *Ví dụ: [*NT101.K11.ANTT*]-Session1_Group3.*
- Nếu báo cáo có nhiều file, nén tất cả file vào file .ZIP với cùng tên file báo cáo.
- Không đặt tên đúng định dạng – yêu cầu, sẽ **KHÔNG** chấm điểm.
- Nộp file báo cáo trên theo thời gian đã thống nhất tại courses.uit.edu.vn.

**Đánh giá**: Sinh viên hiểu và tự thực hiện. Khuyến khích:

- Chuẩn bị tốt.
- Có nội dung mở rộng, ứng dụng trong kịch bản/câu hỏi phức tạp hơn, có đóng góp xây dựng.

*Bài sao chép, trễ, … sẽ được xử lý tùy mức độ vi phạm.*

**HẾT**