```
void example_levels(){
    print_example_banner("Example: Levels");
```

BFV/CKKS 에서 공통으로 언급되는 level 에 대해 알아보겠다. 이를 SEAL 에서 어케다루는지도...

parms_id

- **Encryption 파라미터** (Rand.Num.Generator 포함)는 비트길이 256 인 Hash 로 식별된다.
- 이 hash 를 parms_id 라 하고, 파라미터 바뀌면 바로 변한다. 언제든 print 하여 접근가능, .

modulus switching chain

- **Context** 만들 때, 자동으로 생성되는 얘는 원본파라미터로부터 파생된 다른 파라미터들의 "chain"임.
- mod.sw.ch 의 파라미터는 원본과 같고, coeff.mod 의 크기만 줄어들뿐이다.
- 정확히 말하면, 각 chain 은 이전 chain 의 파라미터에서 coeff.mod 이루는 소수中마지막 prime 을 제거하기를 시도한다. 이는 "plain.mod>coeff.mod"인 유효하지 않은 파라미터 상태까지 지속됨.
- 어쨌든 각 chain, chain 내 parameter 접근 쉬움.

chain index

- 각 chain 파라미터들은 chain 역순으로 ch.idx 가짐
- 첫번째 chain 파라미터 set 의 ch.idx 제일크고, 마지막 chain 파라미터 set 의 ch.idx=0 임.

level

- ch.idx 가 높으면 (뒤에 파생된 것이 아닌 원본쪽에 가까워) level 이 높다고 얘기함.

ContextDATA

- Context 만들 때 파라미터 set 이 관여하는 pre-computations 수행되어 ContextDat 에 저장된다.

➔ [mod.sw.chain]은 연결리스트로, data=[ContextData]객체, node= [parms_id]

➔ [mod.sw.chain]의 노드간 근본적 차이는 coeff.mod 의 변화임

→ 리스트에 새 노드 추가시 이전노드의 **coeff.mod 이루는 마지막 소수제거하므로** coeff.mod 구성하는 "소수들의 순서" 중요하다.

→ 원본파라미터의 coeff.mod 이루는 마지막 소수는 스페셜한 의미있어 "**special prime**"라고 함.

* 파생된 파라미터 set 은 이전의 set 의 coeff.mod 이루는 마지막 소수를 제거한 결과이므로 원본의 마지막 소수는 오로지 "원본 set" 즉, mod.sw.ch 의 1st set 에만 존재.

* 키생성은 1st set (원본 ≡ higher level), 데이터(암호문등)는 lower level 에서 생김.

→ "**special prime**"은 coeff.mod 이루는 가장 큰 소수와 같아야 한다 (loosely).

➔ [mod.sw.chain]의 각 노드마다 coeff.mod 변화와 Lv 명명, 주요한 3 개 lv 에 접근방법

(따로 access 없는건 context_data = context_data->next_context_data 로 찾아감)

coeff_modulus	ch.idx	note	access
{50,30,30,50,**50**}	Lv.4	Key Lv.	SEALContext::**key**_context_data()
{50,30,30,50,50}	Lv.3	Highest Data Lv.	SEALContext::**first**_context_data()
{50,30,30,50,50}	Lv.2		
{50,30,30,50,50}	Lv.1		
{50,30,30,50,50}	Lv.0	Lowest Lv.	SEALContext::**last**_context_data()

기본세팅 : **[1] Enc파라미터 [2] Context [3]키 [4]암호기 [5]Evaluator [6]복호기**

[1] Encryption 파라미터 생성

① poly.mod.deg = 2^{13} =8192

② coeff.mod : "CoeffModulus::Create"로 비트길이 {50,30,30,50,**50**}인 5 개 소수의 곱으로 한다.

- Σ(bit-length)i=210 으로 coeff.mod 비트길이상한(@poly.mod.deg=8192) =218 보다 작다.
- 가장 큰 소수와 special prime 이 같을 수 있도록 비트길이 신경썼음.

③ plain.mod :별 역할안한다. 그냥 reasonable 한 값으로 설정함.

```
EncryptionParameters parms(scheme_type::bfv);
size_t poly_modulus_degree = 8192;
parms.set_poly_modulus_degree(poly_modulus_degree);
parms.set_coeff_modulus(CoeffModulus::Create(poly_modulus_degree, { 50, 30, 30, 50, 50 }));
parms.set_plain_modulus(PlainModulus::Batching(poly_modulus_degree, 20));
```

[2] Context 생성 파라미터 정보확인 후, 각 Lv 에 접근, 정보를 알아보자.

① usual

```
SEALContext context(parms);
print_parameters(context);
cout << endl;
```

```
| Encryption parameters :
        |   scheme                : BFV
        |   poly_modulus_degree: 8192
        |   coeff_modulus size   : 210 (50 + 30 + 30 + 50 + 50) bits
        |   plain_modulus        : 1032193
```

② key level : parms_id, coeff.mod 소수들

```
print_line(__LINE__);
cout << "Print the modulus switching chain." << endl;
auto context_data = context.key_context_data();
cout << "----> Level (chain index): " << context_data->chain_index();
cout << " ..... key_context_data()" << endl;
cout << "        parms_id: " << context_data->parms_id() << endl;
cout << "        coeff_modulus primes: ";
cout << hex;
for (const auto &prime : context_data->parms().coeff_modulus()){
    cout << prime.value() << " ";}
cout << dec << endl;
```

```
Line 106 --> Print the modulus switching chain.
----> Level (chain index): 4 ..... key_context_data()
        parms_id: 26d0ad92b6a78b12 667d7d6411d19434 18ade70427566279 84e0aa06442af302
        coeff_modulus primes: 3ffffffef4001 3ffe8001 3fff4001 3ffffffcc001 3ffffffffc001
```

③ first level → last level : parms_id, coeff.mod 소수들

```
context_data = context.first_context_data();
while (context_data)
{
    cout << " Level (chain index): " << context_data->chain_index();
    if (context_data->parms_id() == context.first_parms_id()){
        cout << " ..... first_context_data()" << endl;          }
    else if (context_data->parms_id() == context.last_parms_id()){
        cout << " ..... last_context_data()" << endl;}
    else{
        cout << endl;}
    cout << "        parms_id: " << context_data->parms_id() << endl;
    cout << "        coeff_modulus primes: ";
    cout << hex;
    for (const auto &prime : context_data->parms().coeff_modulus()){
        cout << prime.value() << " "; }
    cout << dec << endl;
    context_data = context_data->next_context_data();
}
cout << " End of chain reached" << endl << endl;
```

```
--> Level (chain index): 3 ..... first_context_data()
    parms_id: 211ee2c43ec16b18 2c176ee3b851d741 490eacf1dd5930b3 3212f104b7a60a0c
    coeff_modulus primes: 3ffffffef4001 3ffe8001 3fff4001 3ffffffcc001
--> Level (chain index): 2
    parms_id: 85626ad91458073f e186437698f5ff4e a1e71da26dabe039 9b66f4ab523b9be1
    coeff_modulus primes: 3ffffffef4001 3ffe8001 3fff4001
--> Level (chain index): 1
    parms_id: 73b7dc26d10a15b9 56ce8bdd07324dfa 7ff7b8ec16a6f20f b80f7319f2a28ac1
    coeff_modulus primes: 3ffffffef4001 3ffe8001
--> Level (chain index): 0 ..... last_context_data()
    parms_id: af7f6dac55528cf7 2f532a7e2362ab73 03aeaedd1059515e a515111177a581ca
    coeff_modulus primes: 3ffffffef4001
--> End of chain reached
```

[3] 키생성 key 레벨에 생성되는지 보자.

```
KeyGenerator keygen(context);
auto secret_key = keygen.secret_key();
PublicKey public_key;
keygen.create_public_key(public_key);
RelinKeys relin_keys;
keygen.create_relin_keys(relin_keys);
print_line(__LINE__);
cout << "Print the parameter IDs of generated elements." << endl;
cout << "    + public_key:   " << public_key.parms_id() << endl;
cout << "    + secret_key:   " << secret_key.parms_id() << endl;
cout << "    + relin_keys:   " << relin_keys.parms_id() << endl;
```

```
Line 173 --> Print the parameter IDs of generated elements.
    + public_key:   26d0ad92b6a78b12 667d7d6411d19434 18ade70427566279 84e0aa06442af302
    + secret_key:   26d0ad92b6a78b12 667d7d6411d19434 18ade70427566279 84e0aa06442af302
    + relin_keys:   26d0ad92b6a78b12 667d7d6411d19434 18ade70427566279 84e0aa06442af302
```

[4] 암호기 생성

```
Encryptor encryptor(context, public_key);
```

[5] Evaluator 생성

```
Evaluator evaluator(context);
```

[6] 복호기 생성

```
Decryptor decryptor(context, secret_key);
```

메세지→평문

- bfv 에서 평문은 parms_id 수반하지 않는다.

```
Plaintext plain("1x^3 + 2x^2 + 3x^1 + 4");
cout << "    + plain:        " << plain.parms_id() << " (not set in BFV)" << endl;
```

```
+ plain:        0000000000000000 0000000000000000 0000000000000000 0000000000000000   (not set in BFV)
```

평문→암호

- 암호문은 params_id 있고 fresh 암호문은 first Lv.에 있을것, 확인바람.

```
Ciphertext encrypted;
encryptor.encrypt(plain, encrypted);
cout << "    + encrypted:    " << encrypted.parms_id() << endl << endl;
```

```
+ encrypted:    211ee2c43ec16b18 2c176ee3b851d741 490eacf1dd5930b3 3212f104b7a60a0c
```

연산 : 연산없이 modulus_switch()만 **WRONG way**

first level→ last_level 에 걸쳐서 modulus_switch(encrypted) 시행 : parms_id, 노이즈예산 프린트

- (Lv.3→Lv.2) Evaluator::switch_to_next() : Δ= -50bits
- (Lv.2→Lv.1) Evaluator::switch_to_next() : Δ= -30bits
- (Lv.1→Lv.0) Evaluator::switch_to_next() : Δ= -30bits

➔ **결과**: 예산만 낭비하고 별 기능없어 보인다.

- Evaluator::modulus_switch()은 다음 level(↓방향만 가능)의 암호문관련 파라미터를 바꾸는 테크닉. 네?

Evaluator::mod_switch_to_next : the next level chain 으로 암호문전달

Evaluator::mod_switch_to : 원하는 parms_id 위치의 chain 으로 암호문전달.

```
print_line(__LINE__);
    cout << "Perform modulus switching on encrypted and print." << endl;
    context_data = context.first_context_data();
    cout << "---->";
    while (context_data->next_context_data())
    {
        cout << " Level (chain index): " << context_data->chain_index() << endl;
        cout << "         parms_id of encrypted: " << encrypted.parms_id() << endl;
        cout <<"   Noise budget at this level: "<< decryptor.invariant_noise_budget(encrypted)<<" bits" << endl;
        evaluator.mod_switch_to_next_inplace(encrypted);
        context_data = context_data->next_context_data();
    }
    cout << " Level (chain index): " << context_data->chain_index() << endl;
    cout << "         parms_id of encrypted: " << encrypted.parms_id() << endl;
    cout <<"   Noise budget at this level: "<< decryptor.invariant_noise_budget(encrypted)<<" bits" << endl;
    cout << " End of chain reached" << endl << endl;
```

```
Line 200 --> Perform modulus switching on encrypted and print.
----> Level (chain index): 3
        parms_id of encrypted: 211ee2c43ec16b18 2c176ee3b851d741 490eacf1dd5930b3 3212f104b7a60a0c
        Noise budget at this level: 132 bits
--> Level (chain index): 2
        parms_id of encrypted: 85626ad91458073f e186437698f5ff4e a1e71da26dabe039 9b66f4ab523b9be1
        Noise budget at this level: 82 bits
--> Level (chain index): 1
        parms_id of encrypted: 73b7dc26d10a15b9 56ce8bdd07324dfa 7ff7b8ec16a6f20f b80f7319f2a28ac1
        Noise budget at this level: 52 bits
--> Level (chain index): 0
        parms_id of encrypted: af7f6dac55528cf7 2f532a7e2362ab73 03aeaedd1059515e a515111177a581ca
        Noise budget at this level: 22 bits
--> End of chain reached
```

복호화

- 복호화하면 plain 얻게 되는데...

```
print_line(__LINE__);
    cout << "Decrypt still works after modulus switching." << endl;
    decryptor.decrypt(encrypted, plain);
    cout << "       + Decryption of encrypted: " << plain.to_string();
    cout << " ..... Correct." << endl << endl;
```

```
Line 226 --> Decrypt still works after modulus switching.
    + Decryption of encrypted: 1x^3 + 2x^2 + 3x^1 + 4 ..... Correct.
```

Evaluator::modulus_switch()

왜했냐면...

- **암호문길이는 coeff.mod 이루는 소수들의 갯수에 선형비례한다.**
- **그래서 암호문에 더이상 연산할 필요가 없어지면 (즉, 결과암호문) 의뢰인에게 복호화하라고 전송하기 전에, 얘를 coeff.mod 구성하는 소수갯수가 최소인 last level 로 보내는 것이 암호문줄이는 면에서 이득.**
- **연산이 남았더라도, 계산속도를 증가시키는 목적으로 modulus_switch() 할 수 있다.**
- **복호기는 암호문이 어떤 level 것이라도 복호가능하다.**
- **노이즈예산 소모되는 것은 "제대로"한다면 문제되지 않는다. no 소모**

연산 : modulus_switch(), 암호문(X)에 8제곱 **RIGHT way**

① $X=X^2$ → relinearize : Δ= -32bits
② $X=X^2$ → relinearize : Δ= -33bits
③ Evaluator::switch_to_next() : Δ=**0**bits
④ $X=X^2$ → relinearize : Δ= -33bits
⑤ Evaluator::switch_to_next() : Δ=**0**bits

➔ **결과**: coeff.mod 몇개쯤 제거해도 암호문줄이면서 노이즈변화없어 no harm!!

```
print_line(__LINE__);
cout << "Compute the 8th power." << endl;
encryptor.encrypt(plain, encrypted);
cout <<"  Noise budget at this level: "<< decryptor.invariant_noise_budget(encrypted)<<" bits" << endl;
evaluator.square_inplace(encrypted);
evaluator.relinearize_inplace(encrypted, relin_keys);
cout<<"+Noise budget of the 2nd power: "<< decryptor.invariant_noise_budget(encrypted)<<" bits" << endl;
evaluator.square_inplace(encrypted);
evaluator.relinearize_inplace(encrypted, relin_keys);
cout <<"+Noise budget of the 4th power: "<< decryptor.invariant_noise_budget(encrypted)<<" bits" << endl;
evaluator.mod_switch_to_next_inplace(encrypted);
cout<<"+Noise budget after modulus swtiching:"<<decryptor.invariant_noise_budget(encrypted)<<"bits"<<endl;
evaluator.square_inplace(encrypted);
evaluator.relinearize_inplace(encrypted, relin_keys);
cout <<"+Noise budget of the 8th power: "<< decryptor.invariant_noise_budget(encrypted)<<" bits" << endl;
evaluator.mod_switch_to_next_inplace(encrypted);
cout<<"+Noise budget after modulus switching: "<<decryptor.invariant_noise_budget(encrypted)<<"bits"<<endl;
```

```
Line 246 --> Compute the 8th power.
    + Noise budget fresh:                    132 bits
    + Noise budget of the 2nd power:          100 bits
    + Noise budget of the 4th power:          67 bits
    + Noise budget after modulus switching:   67 bits
    + Noise budget of the 8th power:          34 bits
    + Noise budget after modulus switching:   34 bits
```

복호화

- 복호기는 암호문이 어떤 level 에 있건 복호가능하다.
- 따라서 결과를 바꾸지 않는 선에서 암호문길이를 최대한 줄여서 주는게 계산과 복호에 좋다.

```
decryptor.decrypt(encrypted, plain);
    cout << "    + Decryption of the 8th power (hexadecimal) ..... Correct." << endl;
    cout << "    " << plain.to_string() << endl << endl;
```

```
+ Decryption of the 8th power (hexadecimal) ..... Correct.
    1x^24 + 10x^23 + 88x^22 + 330x^21 + EFCx^20 + 3A30x^19 + C0B8x^18 + 22BB0x^17 + 58666x^16 + C88D0x^15 +
9C377x^14 + F4C0Ex^13 + E8B38x^12 + 5EE89x^11 + F8BFFx^10 + 30304x^9 + 5B9D4x^8 + 12653x^7 + 4DFB5x^6 + 879F8x^5 +
825FBx^4 + F1FFEx^3 + 3FFFFx^2 + 60000x^1 + 10000
```

"key level, first level" 두개만 필요?

```
    context = SEALContext(parms, false);
```

확인해보자. coeff.mod 이루는 소수 5 개인데 5levels 아닌 상위 2level 만 생겼는지...

```
    cout << "Optionally disable modulus switching chain expansion." << endl;
    print_line(__LINE__);
    cout << "Print the modulus switching chain." << endl;
    cout << "---->";
    for (context_data = context.key_context_data(); context_data; context_data = context_data-
>next_context_data())
    {
        cout << " Level (chain index): " << context_data->chain_index() << endl;
        cout << "         parms_id: " << context_data->parms_id() << endl;
        cout << "         coeff_modulus primes: ";
        cout << hex;
        for (const auto &prime : context_data->parms().coeff_modulus())
        {
            cout << prime.value() << " ";
        }
        cout << dec << endl;
}
    cout << " End of chain reached" << endl << endl;
}
```

```
Optionally disable modulus switching chain expansion.
Line 306 --> Print the modulus switching chain.
----> Level (chain index): 1
        parms_id: 26d0ad92b6a78b12 667d7d6411d19434 18ade70427566279 84e0aa06442af302
        coeff_modulus primes: 3ffffffef4001 3ffe8001 3fff4001 3ffffffcc001 3ffffffffc001
--> Level (chain index): 0
        parms_id: 211ee2c43ec16b18 2c176ee3b851d741 490eacf1dd5930b3 3212f104b7a60a0c
        coeff_modulus primes: 3ffffffef4001 3ffe8001 3fff4001 3ffffffcc001
--> End of chain reached
```

다음 예제 이해하고 싶으면 이번 예제 숙지하기를 바람.