实验五：RSA实现

17341125 蒙亚愿 信息安全

一、实验目的

> 了解RSA密码体制的原理，通过OAEP最有非对称加密填充实现RSA加密，并且给出解密函数进行验证。

二、实验要求

> 1. p和q是两个1024的安全素数。（每个人按照自己的学号 最后一位的数字选择，例如12345678选择第8组）
> 2. 密钥：私钥为学号的下一个素数，生成公钥
> 3. 明文：中山大学英文（Sun Yat-sen University）进行OAEP填充到2048位
>
> *填充时使用到的两个Hash函数为实验4实现的SHA-1的扩展（hash六次，前面补0到1024）

三、实验介绍

3.1 实验综述

RSA的原理很简单，加解密过程都是模指数运算，但是我们需要计算出密钥。实验已经给出了一组安全素数q、p，我们就可以算出$n = p \times q$作为模数。私钥d是学号的下一位素数$NextPrime(NetID)$，则公钥e就是d在模$\phi(n)$下的逆，即$e \times d = 1 \ mod \ \phi(n)$。计算出密钥之后就可以用于之后的RSA加解密：

$$C = M^e \ mod \ n$$
$$M = C^d \ mod \ n$$

OAEP填充是为了把明文拓展到2048位。首先对于用于填充的信息m，现在高位补充0至1024位得到$left$，同时，1024bit的随机数$right$经过$Hash$函数得到的散列值与$left$1异或，得到2048bit的前半部分L。

L经过$Hash$函数得到的散列值与随机数$right$异或得到明文的后半部分，组成了明文的2048bit。

而OAEP去填充中，密文的后半部分与明文的前半部经过$Hash$函数得到的散列值异或得到$right2$，密文的前半部分再与$right2$经过$Hash$函数得到的散列值异或，得到1024bit的密文，除去前面的0，剩下的就是明文。

3.2 开发环境

运行环境：Windows

使用语言：C++

四、算法原理以及代码实现

由于是大数运算，我使用了gmp库，对于大数使用了mpz_class类型来存储，使得实现偏于C++语言的结构。其中的主要的函数、运算还是自己实现。为了方便计算，密钥的变量都声明为全局变量。

4.1 RSA部分

4.1.1 初始化密钥

我们使用gmp大数库实现。已知一组安全素数q、p，以及私钥key_d。首先计算并初始化密钥：

$$n = pq$$
$$\phi(n) = (p-1)(q-1)$$
$$ed = 1 \ mod \ \phi(n)$$

在计算key_e时，可以使用gmp库中的 mpz_invert函数实现，或者参考实验二中有限域求逆（利用了贝祖定理）即可算出。

```
    mpz_invert(key_e.get_mpz_t(), key_d.get_mpz_t(), n2.get_mpz_t()) ;
```

或者利用贝祖定理计算key_e：

```
mpz_class get_e(mpz_class d, mpz_class n2){  //求xmodn的逆
    mpz_class x=d, n=n2 ;
    mpz_class s=0, t=0 ;
    mpz_class s0=1, s1=0, t0=0, t1=1 ;
    mpz_class r = n%x, q=n/x ;

    while(r){   //贝祖定理
        s = s0-s1*q ;
        t = t0-t1*q ;
        s0 = s1; s1 = s ;
        t0 = t1; t1 = t ;
        n = x ; x = r; r = n%x; q = n/x ;
    }
    return (t+n2)%n2 ;
}
```

初始化密钥：

```
int initial_key(){
    p =
"400882764648455319657432720444581949903421946570574002533113867683669157585114
065901173683653597618159046918837756471765402538115857219375370975542140826693766
907877225294394660196920399062015019060101577249910878397898264680611633866920000
476131145299094113562044184957827175941213256314411457604124324110339" ;
    q =
"226361709166406372611941516071143565417626970204633140130253356906493708694863247
127408771640298681442210214662454386808304013021530309926922058832716592793182884
268579818138519883459067842338738539465516678584076601952899052926066557484379345
762007854345253021301605355478134911941153226370715545582767416231" ;
    key_d = "17341127" ;

    key_n = p*q;
    n2 = (p-1)*(q-1) ;

    mpz_invert(key_e.get_mpz_t(), key_d.get_mpz_t(), n2.get_mpz_t()) ;
   // key_e = get_e(key_d, n2) ;
```

```
    cout<<" 公钥: "<<"\n n = "<<key_n<<"\n e = "<<key_e<<endl<<endl;
    cout<<" 私钥: "<<"\n n = "<<key_n<<"\n d = "<<key_d<<endl<<endl;

    return 0 ;
}
```

4.1.2 RSA加解密:

根据公式:

$$C = M^e \ mod \ n$$
$$M = C^d \ mod \ n$$

直接计算加解密之后的数据。当然，其中的C是经过OAEP填充之后的“明文”，M是还没有去填充的“密文”。调用gmp库里的函数mpz_powm进行模幂运算

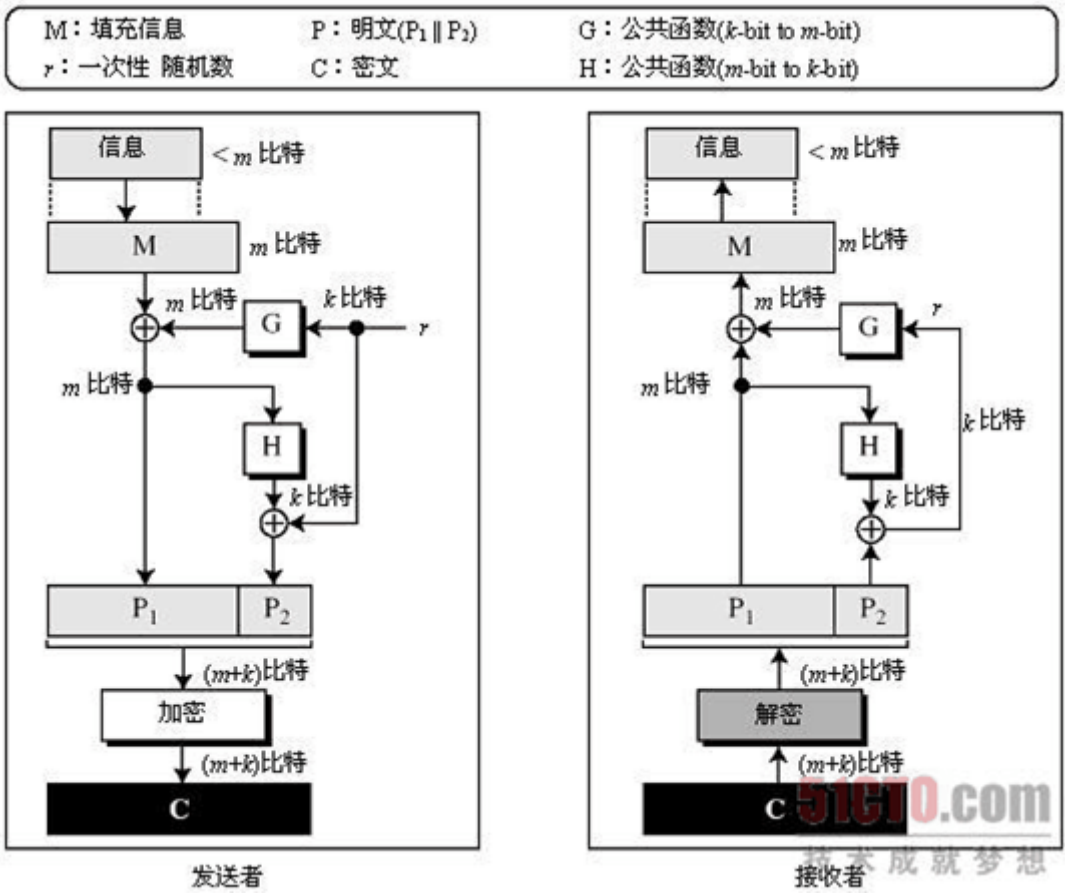
```
    mpz_class plaint_text(num) ;
    cout<<"\n 填充之后: \n"<<plaint_text<<endl;
  //  mpz_class cipher_text = exp_mod(plaint_text, key_e) ;  //2048bit   对的
    mpz_class cipher_text ;
    mpz_powm_sec(cipher_text.get_mpz_t(), plaint_text.get_mpz_t(),
key_e.get_mpz_t(), key_n.get_mpz_t()) ;

    cout<<"\n 加密后的密文(十进制): \n"<<cipher_text<<endl;
  //  mpz_class meg = exp_mod(cipher_text, key_d) ;  //2048bit  对的
    mpz_class meg ;
    mpz_powm_sec(meg.get_mpz_t(), cipher_text.get_mpz_t(), key_d.get_mpz_t(),
key_n.get_mpz_t()) ;
    cout<<"\n 解密之后，还没有去填充得到的: "<<endl;
    cout<<meg<<endl;
```

当然模幂运算(exp_mod)也可以使用平方乘算法，参考有限域中的模幂运算，实现如下:

```
mpz_class exp_mod(mpz_class base, mpz_class exp){
    mpz_class n = key_n ;
    mpz_class e = exp, b = base ;
    mpz_class temp = 1, ans = 0 ;

    while(e>=1){
        if(e==1){
            ans = (temp*b)%n ;
            return ans ;
        }
        else if(e%2==0){  //指数为偶数，底数平方取模，指数/2
            e /= 2 ;
            b = (b*b)%n ;
        }
        else if(e%2==1){  //指数为奇数，先提取一个底数相乘并取模，指数-1
            temp = (b*temp)%n ;
            e--;
        }
    }

}
```

4.2 OAEP填充

算法原理图：



4.2.1 加密填充

首先，将不满1024bit的消息填充到1024bit，即在前面补充0，得到M。对于m，使用$unsigned\ long$数组存储，$unsigned\ long$有32位，但是在存储信息m时，只有低8位存储了信息，剩下的高24位都是0。同时，得到的1024bit的随机数也是以相同的方式存储，32位中，只有低8位有意义，这样就可以与$Hash$函数的接口一致。

在最开始的消息m扩充成1024bit的M时，是用32位都有意义的$unsigned\ long$数组存储。已知明文是Sun Yat-sen University，有22个字节。所以在大小为32的$unsigned\ long$数组中，前22个都是0，明文中的"S"和"u"是第26个字节的低16位，之后的每4个字节组合串在一起，形成一个32位的数。

```
int len = 6 ;
for(int i=0;i+len<32;i++)  left1[i]=0 ;
left1[26] = (m[0]<<8)^m[1] ;
int flag=1 ;
for(int i=27;i<32;i++){
    left1[i] = (m[++flag]<<24)^(m[++flag]<<16)^(m[++flag]<<8)^m[++flag] ;
}
```

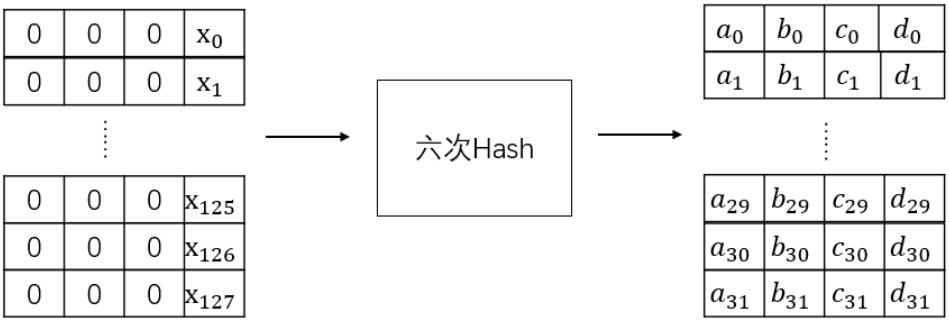
得到的随机数r会有两种存储方式，32位都有意义的数组是用于之后的异或运算；只有低8位有意义的数组用于Hash函数计算散列值。我们先得到低8位有意义的数组，然后通过四个数"串联"得到32位都有意义的数组。

```
//生成一个1024bit的随机数  1024/8=128  空空的32bit  1024/8=128
unsigned long right1[128] ;
```

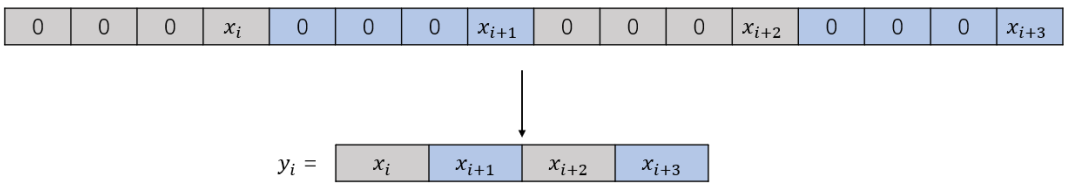
```
    unsigned long temp_right[32] ;  //之后用来和left2异或的满满的32
    unsigned seed = time(0) ;
    srand(seed) ;

    for(int j=0;j<128;j++){  //空空的
        bitset<9> t ;
        for(int k=0;k<8;k++){
            if(!j && !k) t[k] = 1;  //1024bit
            else t[k] = rand()%2 ;
        }
        right1[j] = t.to_ulong() ;
    }

    for(int j=0;j<128;j+=4){  //满满的
        temp_right[j/4] = (right1[j]<<24)^(right1[j+1]<<16)^(right1[j+2]
<<8)^(right1[j+3]) ;
    }
```

经过$Hash$函数之后的散列值也是个$unsigned\ long$的数组，但是此时32位全都占用了，即32位都有意义。低8位有意义的$unsigned\ long$数组与32位都有意义的数组异或时，也是需要先移位补充成32位都有意义的数组。

所以最后的补充后得到的“明文”就是大小为64的、32位都有意义的$unsigned\ long$数组（64*32=2048）。简单来说，低8位有意义的数组才能散列，得到32位都有意义的数组。



由于之后还需要异或，只有低8位有意义的数组就需要移位，一一对应之后才能与32位都有意义的数组异或。每4个串在一起就可以。



```
    unsigned long left2[32] ;   //这是hash之后的满满的32bit  1024/32=32
    unsigned long temp_left[128] ;   //hash之前是空空的32bit
    for(int i=0;i<128;i+=4){  //满满变空空
        unsigned long t = left1[i/4] ;
        temp_left[i] = t>>24 ;
        temp_left[i+1] = (t>>16)^(temp_left[i]<<8) ;
        temp_left[i+2] = (t>>8)^(temp_left[i+1]<<8)^(temp_left[i]<<16) ;
        temp_left[i+3] = t^(temp_left[i]<<24)^(temp_left[i+1]
<<16)^(temp_left[i+2]<<8) ;
    }
```

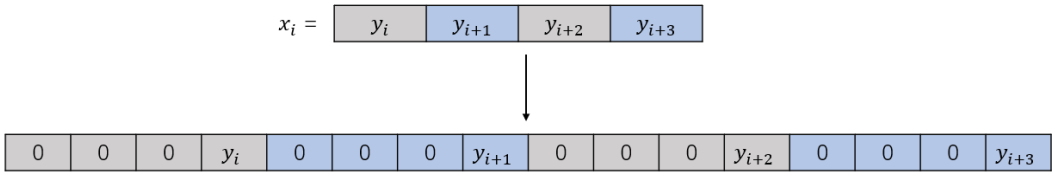
具体的加密填充实现：

```
void OAEP_Padding_Encrption(const char m[], unsigned long plaint_text[]){

    unsigned long left1[32] ;  //满满的32bit  1024/32=32

    //  满满的32bit
    int len = 6 ;
    for(int i=0;i+len<32;i++)  left1[i]=0 ;
    left1[26] = (m[0]<<8)^m[1] ;
    int flag=1 ;
    for(int i=27;i<32;i++){
        left1[i] = (m[++flag]<<24)^(m[++flag]<<16)^(m[++flag]<<8)^m[++flag] ;
    }

    //生成一个1024bit的随机数  1024/8=128  空空的32bit  1024/8=128
    unsigned long right1[128] ;
    unsigned long temp_right[32] ;  //之后用来和left2异或的满满的32
    unsigned seed = time(0) ;
    srand(seed) ;

    for(int j=0;j<128;j++){  //空空的
        bitset<9> t ;
        for(int k=0;k<8;k++){
            if(!j && !k) t[k] = 1;  //1024bit  保证随机数有1024bit
            else t[k] = rand()%2 ;
        }
        right1[j] = t.to_ulong() ;
    }

    for(int j=0;j<128;j+=4){  //满满的
        temp_right[j/4] = (right1[j]<<24)^(right1[j+1]<<16)^(right1[j+2]
<<8)^(right1[j+3]) ;
    }

    //hash函数
    unsigned long right2[32] ;  //hash出来之后是满满的32bit  1024/32=32
    h(right1, right2) ;

    for(int j=0;j<32;j++){
        left1[j] ^= right2[j] ;
    }
```

```
    unsigned long left2[32] ;   //这是hash之后的满满的32bit  1024/32=32
    unsigned long temp_left[128] ;   //hash之前是空空的32bit
    for(int i=0;i<128;i+=4){  //满满变空空
        unsigned long t = left1[i/4] ;
        temp_left[i] = t>>24 ;
        temp_left[i+1] = (t>>16)^(temp_left[i]<<8) ;
        temp_left[i+2] = (t>>8)^(temp_left[i+1]<<8)^(temp_left[i]<<16) ;
        temp_left[i+3] = t^(temp_left[i]<<24)^(temp_left[i+1]
<<16)^(temp_left[i+2]<<8) ;
    }

    h(temp_left, left2) ;

    for(int j=0;j<32;j++){
        left2[j] ^= temp_right[j] ;
    }


    for(int j=0;j<32;j++){
        plaint_text[j] = left1[j] ;
        plaint_text[32+j] = left2[j] ;
    }

}
```

4.2.2 OAEP解密去填充

首先传进来的即将要去填充的“明文”是一个大小为64的、32位都有意义的$unsigned\ long$数组，同样也是在进行Hash之前，要变成只有低8位有意义的数组。



同样，异或的时候也是保持有意义的位数一致，把32位都有意义的数组拓展成低8位有意义的数组。

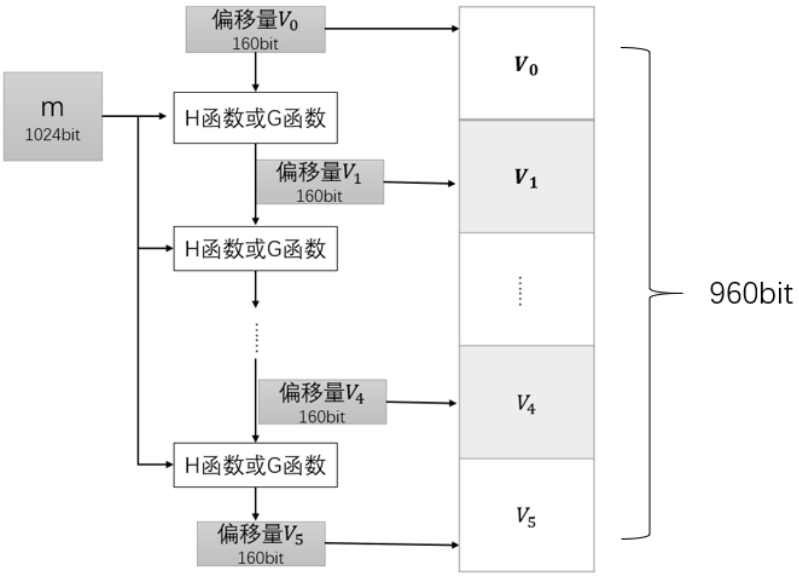
最后得到1024bit的M，从前面的第一个不是0开始，剩下所有的就是真正的明文。

```
void OAEP_Padding_Decryption(unsigned long buff[64], char ans[]){
    // 分成两个1024bit  满满的
    unsigned long left1[32], right1[32] ;
    for(int i=0;i<32;i++){
        left1[i] = buff[i] ;
        right1[i] = buff[i+32] ;
    }
    unsigned long left2[32], right2[32] ;


    unsigned long temp_left[128] ;
    for(int i=0;i<128;i+=4){  //满满变空空
        unsigned long t = left1[i/4] ;
```

```
        temp_left[i] = t>>24 ;
        temp_left[i+1] = (t>>16)^(temp_left[i]<<8) ;
        temp_left[i+2] = (t>>8)^(temp_left[i+1]<<8)^(temp_left[i]<<16) ;
        temp_left[i+3] = t^(temp_left[i]<<24)^(temp_left[i+1]
<<16)^(temp_left[i+2]<<8) ;
        bitset<32> ppp(t) ;
        bitset<8> a1(temp_left[i]), a2(temp_left[i+1]), a3(temp_left[i+2]),
a4(temp_left[i+3]) ;
    }

    h(temp_left, left2) ;

    //满满left2 异或 满满right1
    for(int i=0;i<32;i++){
        right1[i] ^= left2[i] ;
    }

    // 满满right1要先变成空空right1，再hash
    unsigned long temp_right[128] ;
    for(int i=0;i<128;i+=4){  //满满变空空
        unsigned long t = right1[i/4] ;
        temp_right[i] = t>>24 ;
        temp_right[i+1] = (t>>16)^(temp_right[i]<<8) ;
        temp_right[i+2] = (t>>8)^(temp_right[i+1]<<8)^(temp_right[i]<<16) ;
        temp_right[i+3] = t^(temp_right[i]<<24)^(temp_right[i+1]
<<16)^(temp_right[i+2]<<8) ;
    }

    h(temp_right, right2) ;

    // 满满left1 异或 满满right2
    for(int i=0;i<32;i++){
        left1[i] ^= right2[i] ;
    }


    char b[128] ;
    for(int i=0;i<128;i+=4){  //满满变空空
        unsigned long t = left1[i/4] ;
        b[i] = t>>24 ;
        b[i+1] = (t>>16)^(b[i]<<8) ;
        b[i+2] = (t>>8)^(b[i+1]<<8)^(b[i]<<16) ;
        b[i+3] = t^(b[i]<<24)^(b[i+1]<<16)^(b[i+2]<<8) ;
    }

    int flag=0, cnt=0;
    for(int i=0;i<128;i++){
        if(b[i]!=0 && !flag)  flag=1;
        if(flag)  ans[cnt++] = (char)b[i] ; // cout<<ans[cnt-1];
    }

}
```

4.2.3 Hash函数的实现

Hash函数与sha-1中实现的函数几乎一样，只是加上了循环语句和串联960bit的语句。因为需要输出1024bit的散列值，所以可以改造原来的Hash函数：一共有六次迭代，第一次迭代使用的偏移量是sha-1密码体制的固定偏移量，而接下来的五次迭代使用的偏移量则是前一次迭代得到的结果，明文一直不变，一直是原来的，于是，6个散列值（160*6bit）+高位补充0=1024bit。然后将每一次迭代得到的散列值串在一起即可。




```
void h(const unsigned long x[128], unsigned long y[]){ // x 经过hash得到 y
    unsigned long h[5] ;
    unsigned long w[80] ;  // 一共有80个，每个32位，也就是4个字节
    unsigned long k[4] ;

    h[0] = 0x67452301 ;
    h[1] = 0xefcdab89 ;
    h[2] = 0x98badcfe ;
    h[3] = 0x10325476 ;
    h[4] = 0xc3d2e1f0 ;

    k[0] = 0x5a827999 ;
    k[1] = 0x6ed9eba1 ;
    k[2] = 0x8f1bbcdc ;
    k[3] = 0xca62c1d6 ;

    y[0] = y[1] = 0 ;  //在前面补0  (1024-960)/32=2
    int flag = 2 ;

    for(int u=0;u<6;u++){   //六次迭代
        // cal(x, mac) ;  真的计算hash值
        unsigned long block[64] ;  //64 *8 = 512bit
        int blockCnt = 1024/512 ;
        int cnt=0;

        for(int i=0;i<blockCnt;i++){
            for(int j=0;j<64;j++){  // 521/8=64
                block[j] = x[cnt++] ;
            }

            cal_w(block, w);
```

```
            unsigned long a=h[0], b=h[1], c=h[2], d=h[3], e=h[4] ;

            for(int t=0;t<80;t++){
                unsigned long fans = f(t,b,c,d) ;
                unsigned long temp = rotl(a,5) + fans + e + w[t] + k[t/20] ;
                e = d ;
                d = c ;
                c = rotl(b,30) ;
                b = a ;
                a = temp ;
            }

            h[0] += a ;
            h[1] += b ;
            h[2] += c ;
            h[3] += d ;
            h[4] += e ;

        }

        //一次迭代时候，得到160bit的散列值
        // 160*6/32 = 30
        for(int v=0;v<5;v++){
            y[flag] = h[v] ;
          //  printf("%08x",y[flag]) ;
            flag++ ;
        }

    }


}
```

5、结果测试

```
公钥:
n = 907445077811785770755938770664156563749646033218335989487272997157870413487321399274375856609388629736154950568496219885661746806678246455919682900655557201967134980599391453267067902828672443047697656551256602312711841041627312845366157771058005921589091920528898452670359235385444071641804936452556810328917895011755870385964845244254145888366079315373868608413318020007247963111390712731289594433446409402231959955081924233449658480024181463251478321667509204318076098684448272967573081736282791146745715793054844225033024161162615196168016461863882354362731911645812589362211331000757705249986810732323524594019 7
e = 524969747403644911071804200386577859962673214653534538062688694510195996197961538898877575248082500112897147540001464066774253534664819811018134401074740195839270844333325429589449761195848759684685141428274508483152455001796295155073715545829418349736454158170385790067723683864963375737198809348772338213848317927584838235198324475669601186482260565428541498529310106033645109128720200571997242638040535439527272255373352717076719827646838479710531009024233327352723169064189227860890568834523900780776832359854000602004359264679548405311496180169141331572178829622987883697561404371565452638018290889082023737711

私钥:
n = 907445077811785770755938770664156563749646033218335989487272997157870413487321399274375856609388629736154950568496219885661746806678246455919682900655557201967134980599391453267067902828672443047697656551256602312711841041627312845366157771058005921589091920528898452670359235385444071641804936452556810328917895011755870385964845244254145888366079315373868608413318020007247963111390712731289594433446409402231959955081924233449658480024181463251478321667509204318076098684448272967573081736282791146745715793054844225033024161162615196168016461863882354362731911645812589362211331000757705249986810732323524594019 7
d = 17341127

明文: Sun Yat-sen University

填充之后:
172401395322835918050115314644126725686022822508483572723877243010400326329303541511039698206033227015040614402124995337347396236800119432153859328873051262984704559124620104925053137930595425264596750484000992261900905240425917512176120091659161502375851197933268804602741200518230218384632772961020334567446160781034414831033094244892335046323650141347227421163437031051057254718361361893812310433291086260306295758743540448396113332280325628162540369553518295656524940347485811385891020161856571635468310213374360982060510569506017898550952675186003103166036399285900038957495276379555110895532980043818602401642163 3

加密后的密文(十进制):
817818615649810074952932533136839041800760889370848332288223936899601588742627597475411998611370156050934764961610210524231536858032408704245137573260682373580674974293263104786761219042442414850376791380688940185655481497899097570988943325883528338414006538659180183853475792649446107506074458009129130018141733503649702285992674266186394118117332415207154722908769738643902268389273732078076830422339406203471289795832453669875329350222060460586276046860696158445748250535211646921874849477071462628484183104373726372245133287268092842184300347350135527433706807984535153674206560271691460947461695394096639005

解密之后，还没有去填充得到的:
172401395322835918050115314644126725686022822508483572723877243010400326329303541511039698206033227015040614402124995337347396236800119432153859328873051262984704559124620104925053137930595425264596750484000992261900905240425917512176120091659161502375851197933268804602741200518230218384632772961020334567446160781034414831033094244892335046323650141347227421163437031051057254718361361893812310433291086260306295758743540448396113332280325628162540369553518295656524940347485811385891020161856571635468310213374360982060510569506017898550952675186003103166036399285900038957495276379555110895532980043818602401642163 3
```

```
明文：Sun Yat-sen University

填充之后：
172401395322835918050115314644126725686022822508483572723877243010400326329303545151039698206033227015040614402124995337347396236800119432153859328873051262984704559124620104925053137930595425264596750484000992261900905240425917512176120091659161502375851197933268804602741200518230218384632772901020334567446160781034414831033094244892335046323650141347227421163437031051057254718361361893812310433291086260306295758743540448396113332280525628162540369553518295656524940347485811385891020161856571635468310213374360982060510569506017898550952675186003103166036399285900038957495276379555110895532980043818602401642163300

加密后的密文(十进制)：
817818615649810074952932533136839041800760889370848332288223936899601588742627597475411998611370156050934764961610210524231536858032408704245137573260682373580674974293263104786761219042442414850376791380688940185655481497899097570988943325883528383414006538659180183853475792649446107506074458009129130018141733503649702285992674266186394118117332415207154722908769738643902264838927373207807683042233940620347128979583245366987532935022206046058627604686069615844574825053521164692187484947707146262848418310437372637224513328726809284218430034735013552743370680798453515367420656027169146094746169539409663390050

解密之后，还没有去填充得到的：
172401395322835918050115314644126725686022822508483572723877243010400326329303545151039698206033227015040614402124995337347396236800119432153859328873051262984704559124620104925053137930595425264596750484000992261900905240425917512176120091659161502375851197933268804602741200518230218384632772901020334567446160781034414831033094244892335046323650141347227421163437031051057254718361361893812310433291086260306295758743540448396113332280525628162540369553518295656524940347485811385891020161856571635468310213374360982060510569506017898550952675186003103166036399285900038957495276379555110895532980043818602401642163300

对以上密文解密：
Sun Yat-sen University


加解密一致，实验成功！

请按任意键继续. . .
```

在还没有去填充的时候就可发现，“明文”和“密文”模幂计算得到的数是一样的，就可知道加解密计算正确。然后就需要验证去填充算法的正确性。

每次加密的结果都是不一样的，因为要和一个随机数异或，所以RSA很难破解。

```
明文：Sun Yat-sen University

填充之后：
949836921086358743319039131612136857541400617901405215946632913502670122919191938594827692351044796299866530240044808211349218132005971782434562408030670875391184417078709181072434218441982071024407509818259932557332537208828538323263427971797102303397393411930575487642859005885356585125105197111960399792987242455080208140846022957600622829132795336341445836999823132380524164335314568817168120330996306567066262424101033990270481700089001211768082927975162373500150409340228184207588721348825042129774828883474879228012382042234628779020879419905198091031207771657932041420704015544237405253907040431394803254707731 3

加密后的密文(十进制)：
511321591747214207932093306792918180622920833539160402993419910592676014321278605350922150046272361979692077996421811060665186451004314053274840601408208549997468848208991064913324738075173820208987488017080649417234875632786586575849566721029237335246247963429669717660533707426964777819466914721972232673796306937932684054371151771347604959981491476953257253146059406851226051415199795896951130070581916291945675862237750482898945904440813130985763045646090399012094960429622019966768665669531969543145464800164913689360406220645783370968214074404376734665495385812078434153151379405904089102095390318190522279 87

解密之后，还没有去填充得到的：
949836921086358743319039131612136857541400617901405215946632913502670122919191938594827692351044796299866530240044808211349218132005971782434562408030670875391184417078709181072434218441982071024407509818259932557332537208828538323263427971797102303397393411930575487642859005885356585125105197111960399792987242455080208140846022957600622829132795336341445836999823132380524164335314568817168120330996306567066262424101033990270481700089001211768082927975162373500150409340228184207588721348825042129774828883474879228012382042234628779020879419905198091031207771657932041420704015544237405253907040431394803254707731 3

对以上密文解密：
Sun Yat-sen University


加解密一致，实验成功！

请按任意键继续. . .
```

因为这次实验已经给出了明文，不需要输入，所以这个程序会比较有特殊性，只能对特定的明文进行加解密，对于其他明文修改一下填充m的部分就可以。

六、总结与思考

1. 最开始的时候本来想用数组实现模幂运算，到那时想了想并不是很方便实现，之后看到别人的建议使用了gmp库里的函数，但事实上gmp库好难安装……耗费了三天安装，尝试了dev、vs都不行，最后在codeblocks可以了，不过codeblocks界面真的不好看，而且经常抽风，影响我写程序。

2. 在大数的类型上也纠结了一下，本来想使用mpz_t的类型，但是它的加减乘除并没有直接写符号方便，后来改成了mpz_class类型，比较像c++的语法，很方便运算。不过这个类型的函数有点少，但是之后发现可以用get_mpz_t函数转换成mpz_t类型，然后就可以很方便的使用很多函数，就不需要自己实现了，之前写好的模幂运算、求逆运算虽然是对的，但是也许不够简洁，直接用mpz_t里的函数就很快。

3. 在想六次hash是怎么回事的时候自闭了好久，1024bit怎么6次迭代，后来想法刹住了车，突然想到用前一次散列值作为下一次的偏移量，再用原来的明文hash一次不就可以，害，以后不能太死板。

4. 由于之前在sha-1实现的是用unsigned long类型的数组，所以接口我也用了那个类型，但是令我没想到的是里面的异或需要换一下原来数组的存储方式，结果OAEP填充因为这个突然变得很麻

烦，捋清楚了就是在hash之前需要的是比较空的、只有低8位有数的数组，hash出来的就是满满的32位都有意义的数组，异或时也要注意两个数组的存储方式要对应。害，很麻烦……

5. 算出OAEP填充之后的数组，要转成大数，首先想到的是数组里面的数是多少就直接串联起来组成一个大数，然后加密、解密，可是对于解密之后的数要转回unsigned long类型的数组，就是要看位数的，从二进制来看，就是每32位成一个十进制的数组，存储到数组，也就是10位十进制10位十进制地看，然后就有了溢出。我就很疑惑，为什么数组转成大数中的数组都没有发生溢出，而大数转回数组就溢出了。我这才想到之前数组转成大数的方式不对，应该是32bit完整的转换，即有些十进制数不满10位数就在前面补充0，除了第一个不需要补充，否则大数类型会异常。这样，转换成数组的时候就是从最后开始，10位10位的往前补充。

6. 这次试验其实如果只是实现RSA加解密的话还是很简单的，对我来说困难的是gmp库的安装和OAEP填充过渡到之前写好的sha-1密码的hash函数的接口和过渡到加解密的大数，捋清楚细节，再加上codeblock不抽风，就可以“很轻松”的实现。总而言之，收获很大，对于数字的理解范围突然变大了，不再局限于比较小的数字的局限性，也深刻地从计算机存储数据的方式，也认识了原来有一个这么好用的库，RSA的实现可以变得很简单。