```
// Crypto.cpp - CRYPTOGRAPHIC FUNCTIONS FOR HIDE AND SEEK
//
// MODULE INDEX
// NAME                          CONTENTS
// EncryptBlock                  Placeholder function to encrypt a block
// DecryptBlock                  Placeholder function to decrypt a block
// GetRandomByte           Placeholder function to get a random byte
// Encrypt                 Encrypt a page
// Decrypt                 Decrypt a page
// CryHmac_c::CryHmac_c          Hmac constructor
// CryHmac_c::CryHmacStart  Start hmac generation
// CryHmac_c::CryHmacFinish Finish the hmac calculation
// CryHmac_c::op =          Copy operator
// CrySha1_c::CrySha1Transform     Transform the current block
// CrySha1_c::CryHashStart  Start hash calculation
// CrySha1_c::CryHashWrite  Write data to the hash algorithm
// CrySha1_c::CryHashFinish Finish the hash calculation
// GetSignature                  Derive digital signature
//
// MAINTENANCE HISTORY
// DATE             PROGRAMMER AND DETAILS
// 17-09-14   JS     Original
// 08-10-14   STT    Corrected Hash len when compiled as 64-bit code
// 13-10-14   MPF    Corrected to comply with programming standards
// 13-10-14   MPF    Exercise GetSignature
//
//-----------------------------------------------------------------------------

#include <cstring>                // C-style string manipulation functions
#include <cstdlib>                // C-style standard library
#include <cstdio>                 // C-style standard input/output
#include <iostream>               // C++ I/O stream declarations
using namespace std;              // Expand the standard namespace
#include <stdint.h>               // Standard integer type declarations

//-----------------------------------------------------------------------------

// DEFINITIONS

#define       BLOCK_LEN           16     // Block cipher block length
#define       PLAIN_LEN           976    // Plaintext page length

//-----------------------------------------------------------------------------

// BLOCK SHIFT

namespace {
    const unsigned char BLOCK_SHIFT_ARR[BLOCK_LEN] = {
       0x48, 0x31, 0x0a, 0x51,
       0xd5, 0xfb, 0x35, 0x46,
       0x47, 0x05, 0xd0, 0x2e,
       0x86, 0x46, 0x3e, 0x93
    };
};

//-----------------------------------------------------------------------------

// RANDOM NUMBER GENERATOR STATE
```

```
namespace {
    unsigned short randState[3] = {0xd908, 0x3d06, 0x9359};
};

//-------------------------------------------------------------------------

// HMAC, KEY-HASHING FOR MESSAGE AUTHENTICATION

#define       CRY_HMAC_MAX_BLK_LEN 64     // Max block length (octets)
#define       CRY_HMAC_MAX_HASH_LEN       20     // Max hash length (octets)
#define       CRY_HMAC_IPAD        0x36   // Inner pad char
#define       CRY_HMAC_OPAD        0x5c   // Outer pad char

//-------------------------------------------------------------------------

// GENERALISED HASHING CLASS

class CryHash_c {

    // Public Constants
public:
    const size_t     CRY_HASH_BLK_LEN;
                           // Block length (octets)
    const size_t     CRY_HASH_HASH_LEN;
                           // Hash length (octets)
    // Public Methods
public:
    CryHash_c (size_t blkLen, size_t hashLen) : CRY_HASH_BLK_LEN(blkLen),
       CRY_HASH_HASH_LEN(hashLen) {}
                           // Constructor
    virtual void CryHashStart () = 0;
                           // Start hash calculation
    virtual void CryHashWrite (const void *buf, size_t len) = 0;
                           // Write data to the hash algorithm
    virtual void CryHashFinish (void *hashValue) = 0;
                           // Finish hash calculation
    CryHash_c &operator = (const class CryHash_c &/*src*/) { return *this; }
                           // Assignment operator
};

//-------------------------------------------------------------------------

// CLASS DECLARATION

class CryHmac_c {

    // Private Variables
private:
    unsigned char    cryHmacEffKey[CRY_HMAC_MAX_BLK_LEN]; // Effective key
    CryHash_c        *cryHmacHash; // Hashing instance

    // Public Methods
public:
    CryHmac_c (CryHash_c *hash);
                           // Constructor
    void CryHmacStart (const void *keyBuf, size_t keyLen);
                           // Start HMAC calculation
```

```
    void CryHmacWrite (const void *buf, size_t len)
       { cryHmacHash->CryHashWrite (buf, len); }
                            // Write data to the HMAC algorithm
    void CryHmacFinish (void *hashValue);
                            // Finish HMAC calculation
    CryHmac_c &operator = (const class CryHmac_c &src);
                            // Assignment operator
};

//---------------------------------------------------------------------------

// SECURE HASH SHA1 DEFINITIONS

#define       CRY_SHA1_BLK_LEN     64     // Block length (chars)
#define       CRY_SHA1_LONG_PER_BLK       (CRY_SHA1_BLK_LEN/sizeof(uint32_t))
                                  // Longs per block
#define       CRY_SHA1_HASH_LEN    20     // Hash length (chars)
#define       CRY_SHA1_LONG_PER_HASH      (CRY_SHA1_HASH_LEN/sizeof(uint32_t))
                                  // Longs per hash

//---------------------------------------------------------------------------

// CLASS DECLARATION

class CrySha1_c :
    public CryHash_c
{
    // Class Instance Data

    uint32_t         crySha1HashReg[CRY_SHA1_LONG_PER_HASH]; // Hash register
    unsigned char    crySha1BlkBuf[CRY_SHA1_BLK_LEN]; // Block buffer
    unsigned char    crySha1BlkLen;       // Block length
    unsigned long    crySha1HashLen;      // Current hashed length (in bytes)

    // Private Methods

    void CrySha1Transform ();
                            // Transform the current block
    // Public Methods
public:
    CrySha1_c () : CryHash_c (CRY_SHA1_BLK_LEN, CRY_SHA1_HASH_LEN) {}
                            // Construct the base class
    void CryHashStart ();
                            // Start hash calculation
    void CryHashWrite (const void *buf, size_t len);
                            // Write data to the hash algorithm
    void CryHashFinish (void *hashValue);
                            // Finish hash calculation
};

//---------------------------------------------------------------------------

// KEY HASHING FOR MESSAGE AUTHENTICATION

class CryHmacSha1_c :
    private CrySha1_c,
    public CryHmac_c
{
```

```
public:
    CryHmacSha1_c () : CrySha1_c (), CryHmac_c (this) {}
                            // Construct the base class
};

//---------------------------------------------------------------------------

// PLACEHOLDER FUNCTION TO ENCRYPT A BLOCK
// Warning: This function is a trivial Ceasar cipher for testing purposes
// only.  It must not be used to secure confidential data.

void
EncryptBlock (
    const char              *key,         // Encryption key
    const unsigned char     *plainBlock,  // Plaintext block
    unsigned char    *cipherBlock) // Ciphertext block
{
    size_t           i;            // General purpose index
    const char       *p;           // General purpose pointer
    unsigned char    r;            // A register
    unsigned char    c;            // Carry

    c = 0;
    for (i = 0; i < BLOCK_LEN; i++) {
       r = plainBlock[i] + BLOCK_SHIFT_ARR[i] + c;
       c = r < plainBlock[i] || (r == plainBlock[i] && c != 0);
       cipherBlock[i] = r;
    }
    if (*key != '\0') {
       p = key;
       c = 0;
       for (i = 0; i < BLOCK_LEN; i++) {
           r = cipherBlock[i] + static_cast<unsigned char>(*p) + c;
           c = r < cipherBlock[i] || (r==cipherBlock[i] && c != 0);
           cipherBlock[i] = r;
           p ++ ;
           if (*p == '\0') p = key;
       }
    }
}

//---------------------------------------------------------------------------

// PLACEHOLDER FUNCTION TO DECRYPT A BLOCK
// Warning: This function is a trivial Ceasar cipher for testing purposes
// only.  It must not be used to secure confidential data.

void
DecryptBlock (
    const char              *key,         // Encryption key
    const unsigned char     *cipherBlock, // Ciphertext block
    unsigned char    *plainBlock)  // Plaintext block
{
    size_t           i;            // General purpose index
    const char              *p;           // General purpose pointer
    unsigned char    r;            // A register
    unsigned char    c;            // Carry
```

```
    if (*key != '\0') {
       p = key;
       c = 0;
       for (i = 0; i < BLOCK_LEN; i++) {
           r = cipherBlock[i] - static_cast<unsigned char>(*p) - c;
           c = r > cipherBlock[i] || (r==cipherBlock[i] && c!=0);
           plainBlock[i] = r;
           p ++ ;
           if (*p == '\0') p = key;
       }
    }
    c = 0;
    for (i = 0; i < BLOCK_LEN; i++) {
       r = plainBlock[i] - BLOCK_SHIFT_ARR[i] - c;
       c = r > plainBlock[i] || (r == plainBlock[i] && c != 0);
       plainBlock[i] = r;
    }
}

//-----------------------------------------------------------------------------

// PLACEHOLDER FUNCTION TO GET A RANDOM BYTE
// Warning: This function uses a linear congruential random number generator.
// It is not cryptographically secure.

unsigned char
GetRandomByte ()
{
    unsigned short   y[3];         // Final state
    unsigned long    r;            // A register

    r = static_cast<unsigned long>(randState[0]) * 0xe66dUL + 0xb;
    y[0] = static_cast<unsigned short>(r);
    r >>= 16;
    r += static_cast<unsigned long>(randState[0]) * 0xdeecUL +
       static_cast<unsigned long>(randState[1]) * 0xe66dUL;
    y[1] = static_cast<unsigned short>(r);
    r >>= 16;
    r += static_cast<unsigned long>(randState[0]) * 0x0005UL +
       static_cast<unsigned long>(randState[2]) * 0xe66dUL +
       static_cast<unsigned long>(randState[1]) * 0xdeecUL;
    y[2] = static_cast<unsigned short>(r);
    randState[0] = y[0];
    randState[1] = y[1];
    randState[2] = y[2];
    return (randState[1] >> 1) & 0xff;
}

//-----------------------------------------------------------------------------

// ENCRYPT A PAGE

void
Encrypt (
    const char              *key,         // Encryption key
    const void              *plainText,   // Plaintext buffer
    void            *cipherText)  // Ciphertext buffer
{
```

```
    unsigned char    *iv;          // Pointer to initialisation vector
    unsigned char    xorBlock[BLOCK_LEN]; // Result of exclusive or
    const unsigned char     *plainBlock;  // Pointer to plaintext block
    unsigned char    *cipherBlock; // Pointer to ciphertext block
    size_t           i, j;         // General purpose indices

    iv = static_cast<unsigned char*>(cipherText);
    for (i = 0; i < BLOCK_LEN; i++) iv[i] = GetRandomByte();
    plainBlock = static_cast<const unsigned char*>(plainText);
    cipherBlock = iv + BLOCK_LEN;
    for (j = 0; j < PLAIN_LEN; j += BLOCK_LEN) {
       for (i = 0; i < BLOCK_LEN; i++)
           xorBlock[i] = iv[i] ^ plainBlock[i];
       EncryptBlock (key, xorBlock, cipherBlock);
       iv += BLOCK_LEN;
       plainBlock += BLOCK_LEN;
       cipherBlock += BLOCK_LEN;
    }
}

//---------------------------------------------------------------------------

// DECRYPT A PAGE

void
Decrypt (
    const char              *key,         // Encryption key
    const void              *cipherText,  // Ciphertext buffer
    void             *plainText)   // Plaintext buffer
{
    const unsigned char     *iv;          // Pointer to initialisation vector
    unsigned char    xorBlock[BLOCK_LEN]; // Result of exclusive or
    unsigned char    *plainBlock;  // Pointer to plaintext block
    const unsigned char     *cipherBlock; // Pointer to ciphertext block
    size_t           i, j;         // General purpose indices

    iv = static_cast<const unsigned char*>(cipherText);
    plainBlock = static_cast<unsigned char*>(plainText);
    cipherBlock = iv + BLOCK_LEN;
    for (j = 0; j < PLAIN_LEN; j += BLOCK_LEN) {
       DecryptBlock (key, cipherBlock, xorBlock);
       for (i = 0; i < BLOCK_LEN; i++)
           plainBlock[i] = iv[i] ^ xorBlock[i];
       iv += BLOCK_LEN;
       plainBlock += BLOCK_LEN;
       cipherBlock += BLOCK_LEN;
    }
}

//---------------------------------------------------------------------------

// HMAC CONSTRUCTOR

CryHmac_c::CryHmac_c (
    CryHash_c        *hash)        // Secure hash instance
:
    cryHmacHash (hash)
{
```

```
    if (hash->CRY_HASH_BLK_LEN > CRY_HMAC_MAX_BLK_LEN) {
       cerr << "CryHmac_c::CryHmac_c: block length too long"
           << hash->CRY_HASH_BLK_LEN << '\n';
       abort ();
    }
    if (hash->CRY_HASH_HASH_LEN > CRY_HMAC_MAX_HASH_LEN) {
       cerr << "CryHmac_c::CryHmac_c: hash length too long"
           << hash->CRY_HASH_HASH_LEN << '\n';
       abort ();
    }
}

//------------------------------------------------------------------------

// START HMAC GENERATION

void
CryHmac_c::CryHmacStart (
    const void              *keyBuf,      // Key buffer
    size_t           keyLen)              // Key length
{
    unsigned char    keyXorIpad[CRY_HMAC_MAX_BLK_LEN]; // Key XOR inner-pad
    size_t           i;            // General purpose index

    // If the key length is greater than the block size, use the hash
    // of the key instead of the key itself.

    if (keyLen > cryHmacHash->CRY_HASH_BLK_LEN) {
       cryHmacHash->CryHashStart ();
       cryHmacHash->CryHashWrite (keyBuf, keyLen);
       cryHmacHash->CryHashFinish (cryHmacEffKey);
       memset (cryHmacEffKey+cryHmacHash->CRY_HASH_HASH_LEN, 0,
           cryHmacHash->CRY_HASH_BLK_LEN-cryHmacHash->CRY_HASH_HASH_LEN);
    }

    // If the key length is reasonable, the key is the key.

    else {
       memcpy (cryHmacEffKey, keyBuf, keyLen);
       memset (cryHmacEffKey+keyLen, 0, cryHmacHash->CRY_HASH_BLK_LEN-keyLen);
    }

    // Prepare the hash function to receive the protected text

    for (i = 0; i < cryHmacHash->CRY_HASH_BLK_LEN; i++)
       keyXorIpad[i] = cryHmacEffKey[i] ^ CRY_HMAC_IPAD;
    cryHmacHash->CryHashStart ();
    cryHmacHash->CryHashWrite (keyXorIpad, cryHmacHash->CRY_HASH_BLK_LEN);
}

//------------------------------------------------------------------------

// FINISH THE HMAC CALCULATION

void
CryHmac_c::CryHmacFinish (
    void             *hmacValue)   // Calculated hash value
{
```

```
    unsigned char    keyXorOpad[CRY_HMAC_MAX_BLK_LEN]; // Key XOR inner-pad
    unsigned char    firstHash[CRY_HMAC_MAX_HASH_LEN]; // Hash of first pass
    size_t           i;            // General purpose index

    cryHmacHash->CryHashFinish (firstHash);
    for (i = 0; i < cryHmacHash->CRY_HASH_BLK_LEN; i++)
       keyXorOpad[i] = cryHmacEffKey[i] ^ CRY_HMAC_OPAD;
    cryHmacHash->CryHashStart ();
    cryHmacHash->CryHashWrite (keyXorOpad, cryHmacHash->CRY_HASH_BLK_LEN);
    cryHmacHash->CryHashWrite (firstHash, cryHmacHash->CRY_HASH_HASH_LEN);
    cryHmacHash->CryHashFinish (hmacValue);
}

//---------------------------------------------------------------------------

// COPY OPERATOR

CryHmac_c &
CryHmac_c::operator = (
    const CryHmac_c  &src)         // Source instance
{
    memcpy (cryHmacEffKey, src.cryHmacEffKey, CRY_HMAC_MAX_BLK_LEN);
    return *this;
}

//---------------------------------------------------------------------------

// CIRCULAR SHIFT FUNCTION

#define CrySha1CircularShift(bits,word) \
    (((word) << (bits)) | ((word) >> (32-(bits))))

//---------------------------------------------------------------------------

// TRANSFORM THE CURRENT BLOCK

void
CrySha1_c::CrySha1Transform ()
{
    static const uint32_t K[] =    // Constants defined in SHA-1
       {0x5A827999, 0x6ED9EBA1, 0x8F1BBCDC, 0xCA62C1D6};
    int                    t;           // Loop counter
    uint32_t         temp;         // Temporary word value
    uint32_t         W[80];        // Word sequence
    uint32_t         A, B, C, D, E;       // Word buffers

    // Load the first 16 words in the array W

    for (t = 0; t < 16; t++) {
       W[t] = crySha1BlkBuf[t * 4] << 24;
       W[t] |= crySha1BlkBuf[t * 4 + 1] << 16;
       W[t] |= crySha1BlkBuf[t * 4 + 2] << 8;
       W[t] |= crySha1BlkBuf[t * 4 + 3];
    }

    for(t = 16; t < 80; t++)
       W[t] = CrySha1CircularShift (1, W[t-3] ^ W[t-8] ^ W[t-14] ^ W[t-16]);
```

```
    A = crySha1HashReg[0];
    B = crySha1HashReg[1];
    C = crySha1HashReg[2];
    D = crySha1HashReg[3];
    E = crySha1HashReg[4];

    for (t = 0; t < 20; t++) {
       temp = CrySha1CircularShift(5,A) +
              ((B & C) | ((~B) & D)) + E + W[t] + K[0];
       E = D;
       D = C;
       C = CrySha1CircularShift(30,B);
       B = A;
       A = temp;
    }

    for (t = 20; t < 40; t++) {
       temp = CrySha1CircularShift(5,A) + (B ^ C ^ D) + E + W[t] + K[1];
       E = D;
       D = C;
       C = CrySha1CircularShift(30,B);
       B = A;
       A = temp;
    }

    for (t = 40; t < 60; t++) {
       temp = CrySha1CircularShift(5,A) +
              ((B & C) | (B & D) | (C & D)) + E + W[t] + K[2];
       E = D;
       D = C;
       C = CrySha1CircularShift(30,B);
       B = A;
       A = temp;
    }

    for (t = 60; t < 80; t++) {
       temp = CrySha1CircularShift(5,A) + (B ^ C ^ D) + E + W[t] + K[3];
       E = D;
       D = C;
       C = CrySha1CircularShift(30,B);
       B = A;
       A = temp;
    }

    crySha1HashReg[0] += A;
    crySha1HashReg[1] += B;
    crySha1HashReg[2] += C;
    crySha1HashReg[3] += D;
    crySha1HashReg[4] += E;
}

//------------------------------------------------------------------------------

// START HASH CALCULATION

void
CrySha1_c::CryHashStart ()
{
```

```
    crySha1HashReg[0] = 0x67452301;
    crySha1HashReg[1] = 0xefcdab89;
    crySha1HashReg[2] = 0x98badcfe;
    crySha1HashReg[3] = 0x10325476;
    crySha1HashReg[4] = 0xc3d2e1f0;
    crySha1HashLen = 0;
    crySha1BlkLen = 0;
}

//-----------------------------------------------------------------------------

// WRITE DATA TO THE HASH ALGORITHM

void
CrySha1_c::CryHashWrite (
    const void              *buf,         // Data buffer
    size_t          len)          // Buffer length
{
#   if 0
    {
       size_t        i;            // General purpose index
       printf ("CryHashWrite:");
       for (i = 0; i < len; i++) {
           if (i != 0 && (i & 0x0f) == 0) printf ("\n             ");
           printf (" %02x", ((unsigned char*)buf)[i]);
       }
       printf ("\n");
    }
#   endif
    while (len-- != 0) {
       crySha1BlkBuf[crySha1BlkLen++] =
           *(*reinterpret_cast<const unsigned char**>(&buf))++;
       if (crySha1BlkLen == CRY_SHA1_BLK_LEN) {
           crySha1BlkLen = 0;
           CrySha1Transform ();
           crySha1HashLen += CRY_SHA1_BLK_LEN;
       }
    }
}

//-----------------------------------------------------------------------------

// FINISH THE HASH CALCULATION

void
CrySha1_c::CryHashFinish (
    void            *hashValue)   // Calculated hash value
{
    size_t          i;            // General purpose index
    uint32_t        r;            // Shift register
    unsigned long   grossLen;     // Gross length (bytes)

    // Calculate the total message length

    grossLen = crySha1HashLen + crySha1BlkLen;

    // Append the padding
```

```
    crySha1BlkBuf[crySha1BlkLen++] = 0x80;
    while (crySha1BlkLen != CRY_SHA1_BLK_LEN-8) {
       if (crySha1BlkLen >= CRY_SHA1_BLK_LEN) {
           CrySha1Transform ();
           crySha1BlkLen = 0;
       }
       crySha1BlkBuf[crySha1BlkLen++] = 0;
    }

    // Append the total length in bits

    crySha1BlkBuf[CRY_SHA1_BLK_LEN-8] = 0;
    crySha1BlkBuf[CRY_SHA1_BLK_LEN-7] = 0;
    crySha1BlkBuf[CRY_SHA1_BLK_LEN-6] = 0;
    crySha1BlkBuf[CRY_SHA1_BLK_LEN-5] =
       static_cast<unsigned char>(grossLen >> 29);
    crySha1BlkBuf[CRY_SHA1_BLK_LEN-4] =
       static_cast<unsigned char>(grossLen >> 21);
    crySha1BlkBuf[CRY_SHA1_BLK_LEN-3] =
       static_cast<unsigned char>(grossLen >> 13);
    crySha1BlkBuf[CRY_SHA1_BLK_LEN-2] =
       static_cast<unsigned char>(grossLen >> 5);
    crySha1BlkBuf[CRY_SHA1_BLK_LEN-1] =
       static_cast<unsigned char>(grossLen << 3);

    // Transform the final block

    CrySha1Transform ();

    // Generate the hash

    for (i = 0; i < CRY_SHA1_LONG_PER_HASH; i++) {
       r = crySha1HashReg[i];
       *(*reinterpret_cast<unsigned char**>(&hashValue))++ =
           static_cast<unsigned char>(r >> 24);
       *(*reinterpret_cast<unsigned char**>(&hashValue))++ =
           static_cast<unsigned char>(r >> 16);
       *(*reinterpret_cast<unsigned char**>(&hashValue))++ =
           static_cast<unsigned char>(r >> 8);
       *(*reinterpret_cast<unsigned char**>(&hashValue))++ =
           static_cast<unsigned char>(r);
    }
}

//-----------------------------------------------------------------------------

// DERIVE DIGITAL SIGNATURE

void
GetSignature (
    const char              *key,         // Signature key
    const void              *data,        // Data to be signed
    void            *signature)   // Constructed signature
{
    CrySha1_c       crySha1;      // SHA-1 implementation

    crySha1.CryHashStart();
    crySha1.CryHashWrite (key, strlen(key));
```

```
    crySha1.CryHashWrite (data, 1004);
    crySha1.CryHashFinish (signature);
}

//---------------------------------------------------------------------------

// TESTING MAIN LINE

#if 0
int
main ()
{
    unsigned char plainPage[PLAIN_LEN];
    unsigned char cipherPage[PLAIN_LEN+BLOCK_LEN];
    unsigned char decryptPage[PLAIN_LEN];
    unsigned char signature[20];
    size_t           i, j;

    cout << "Test started\n";
    for (i = 0; i < 1000; i++) {
       for (j = 0; j < BLOCK_LEN; j++)
           plainPage[j] = lrand48();
       Encrypt ("abc123", plainPage, cipherPage);
       GetSignature ("abc123", cipherPage, signature);
       Decrypt ("abc123", cipherPage, decryptPage);
       if (memcmp (plainPage, decryptPage, BLOCK_LEN) != 0)
           cerr << "Error: page mismatch\n";
    }
    cout << "Test completed\n";
    return 0;
}
#endif
```