班级：2022211301　　**姓名：**卢安来　　**学号：**2022212720

根据课堂教学内容，更正 hmac.c，hmac.h，test.cpp 和 Makefile。

问题一：内存管理。

在原有的 hmac 函数中，返回结果的内存空间在函数 hmac 内分配，将在 hmac 外被释放。

这违背了「应该让调用者自己管理资源」的原则。故应当将内存分配和释放都交由调用者负责，而 hmac 则通过参数获取可以存放结果的内存空间地址，即修改 hmac 函数的声明如下。

```
/**
 * Computes the HMAC (Hash-based Message Authentication Code) of a
given string using a specified key and algorithm.
 *
 * @param str The input string to compute the HMAC for.
 * @param str_length The length of the input string.
 * @param key The key used for the HMAC computation.
 * @param key_length The length of the key.
 * @param algorithm The name of the hash algorithm to use (e.g.,
"SHA256").
 * @param buffer The buffer to store the resulting HMAC string in
hexadecimal format.
 * @param buffer_size The size of the buffer.
 * @return Pointer to the buffer containing the HMAC string on success,
or NULL on failure.
 */
extern char *hmac(
    const char *const str, const size_t str_length,
    const char *const key, const size_t key_length,
    const char *const algorithm,
    char *const buffer, const size_t buffer_size);
```

其中 buffer 和 buffer_size 即为调用者提供的用于存放结果的内存空间。

`test` 中对应修改如下，即

```
void test_hmac(const char *const algorithm) {
    const char *const src = "kjhdskfhdskfjhdskjfdskfdskfjsdkfjds";
    const char *const key = "kjshfkds";

    const std::size_t buffer_size = 256;
    char *const buffer = new char[buffer_size];

    const char *const result = hmac(
        src, std::strlen(src),
        key, std::strlen(key),
        algorithm,
        buffer, buffer_size);

    if (result)
        printf("%s result = [%s]\n", algorithm, result);
    else
        printf("ERROR: %s\n", buffer);

    delete[] buffer;

    return;
}
```

问题二：错误处理。

在原有的 hmac 函数中，如果出现错误，则 hmac 函数会将一些错误信息直接输出到标准输出 stdout。

这违背了「库函数的接口应该安静地返回不同的返回值，让调用者根据返回值做不同的处理」的原则。故应当将错误报告的逻辑进行调整，当出现错误时，hmac 函数将返回空指针，此时错误信息将被输出到 buffer 中，为防止 buffer 溢出，使用 snprintf 限定其至多输出 buffer_size 个字符（包括结尾的 \0），下面给出一个修改的案例。

```
if (!engine) {
    snprintf(buffer, buffer_size,
                "invalid algorithm %s for hmac", algorithm);
    return NULL;
}
```

错误信息末尾的多余换行被移除，以方便调用者自行进行格式化输出控制。

问题三：弃用的库函数。

在原有的 hmac 函数中，使用了 HMAC_CTX_new, HMAC_Init_ex, HMAC_CTX_free, HMAC_Update, HMAC_Final 这些在 OpenSSL 3.0 及以上版本中被弃用的 API。

这违背了接口对兼容性和安全性的要求。故应当针对 OpenSSL 的不同版本进行兼容，这需要使用 C 语言的条件编译功能，即使用 #if 和 #else 针对 OpenSSL 版本进行条件编译，对于 OpenSSL 3.0 及以上版本换用更新的 API。

```
    const EVP_MD *engine = EVP_get_digestbyname(algorithm);
    if (!engine) {
        snprintf(buffer, buffer_size,
                 "invalid algorithm %s for hmac", algorithm);
        return NULL;
    }

#if OPENSSL_VERSION_NUMBER >= 0x30000000L
    // OpenSSL 3.0 and above
    EVP_MAC *mac = EVP_MAC_fetch(NULL, "HMAC", NULL);
    if (!mac) {
        snprintf(buffer, buffer_size,
                 "EVP_MAC_fetch fail");
        return NULL;
    }

    EVP_MAC_CTX *ctx = EVP_MAC_CTX_new(mac);
    if (!ctx) {
        EVP_MAC_free(mac);
        snprintf(buffer, buffer_size,
                 "EVP_MAC_CTX_new fail");
        return NULL;
    }

    OSSL_PARAM params[3];
    params[0] = OSSL_PARAM_construct_utf8_string(
        "digest", (char *)algorithm, 0);
    params[1] = OSSL_PARAM_construct_octet_string(
        "key", (unsigned char *)key, key_length);
    params[2] = OSSL_PARAM_construct_end();

    if (!EVP_MAC_init(ctx, NULL, 0, params)) {
        EVP_MAC_CTX_free(ctx);
        EVP_MAC_free(mac);
        snprintf(buffer, buffer_size,
                 "EVP_MAC_init fail");
        return NULL;
    }

    if (!EVP_MAC_update(ctx, (unsigned char *)str, str_length)) {
        snprintf(buffer, buffer_size,
```

```
                "EVP_MAC_update fail");
        EVP_MAC_CTX_free(ctx);
        EVP_MAC_free(mac);
        return NULL;
    }

    unsigned char output[EVP_MAX_MD_SIZE];
    size_t output_length = sizeof(output);
    if (!EVP_MAC_final(ctx, output, &output_length, sizeof(output))) {
        snprintf(buffer, buffer_size,
                "EVP_MAC_final fail");
        EVP_MAC_CTX_free(ctx);
        EVP_MAC_free(mac);
        return NULL;
    }

    EVP_MAC_CTX_free(ctx);
    EVP_MAC_free(mac);
#else
    // OpenSSL < 3.0
    HMAC_CTX *ctx = HMAC_CTX_new();
    if (HMAC_Init_ex(ctx, key, key_length, engine, NULL) <= 0) {
        snprintf(buffer, buffer_size,
                "HMAC_Init_ex fail");
        HMAC_CTX_free(ctx);
        return NULL;
    }

    if (HMAC_Update(ctx, (unsigned char *)str, str_length) <= 0) {
        snprintf(buffer, buffer_size,
                "HMAC_Update fail");
        HMAC_CTX_free(ctx);
        return NULL;
    }

    unsigned char output[EVP_MAX_MD_SIZE];
    unsigned int output_length;
    if (HMAC_Final(ctx, output, &output_length) <= 0) {
        snprintf(buffer, buffer_size,
                "HMAC_Final fail");
        HMAC_CTX_free(ctx);
        return NULL;
    }

    if (output_length > EVP_MAX_MD_SIZE) {
        snprintf(buffer, buffer_size,
                "HMAC_Final return invalid output length %u\n",
output_length);
        HMAC_CTX_free(ctx);
        return NULL;
    }

    HMAC_CTX_free(ctx);
#endif
```

问题四：不一致的注释。

在 `hmac.h` 原有的注释中，曾说明支持的算法为 md5, sha1, sha256, sha512，这与实际支持的情况不符。

这违背了接口注释的准确性和一致性的原则，即注释与代码的实际状态并未保持一致。故应该修改注释，告知调用者如何获取所有可用的算法（使用命令 `openssl list -digest-algorithms` 或调用 OpenSSL API），即

```
/**
 * Computes the HMAC (Hash-based Message Authentication Code) of a given string using a specified key and algorithm.
 * This function utilizes the OpenSSL library to perform the HMAC computation. It is important to ensure that
 * the OpenSSL library is properly initialized and that the specified algorithm is supported by the library.
 *
 * The function takes the following parameters:
 *
 * @param str The input string to compute the HMAC for. This is the data that will be authenticated.
 * @param str_length The length of the input string. It is important to provide the correct length to avoid buffer overflows.
 * @param key The key used for the HMAC computation. This should be a secret key known only to the sender and receiver.
 * @param key_length The length of the key. Like the input string length, this is crucial for preventing buffer overflows.
 * @param algorithm The name of the hash algorithm to use (e.g., "SHA256"). This should correspond to a valid and available
 *                  hash algorithm in the OpenSSL library. You can check the available hash algorithms using the OpenSSL
 *                  command-line tool or by using EVP_MD_fetch in the EVP library to enumerate them.
 * @param buffer The buffer to store the resulting HMAC string in hexadecimal format. The buffer must be large enough to
 *               hold the HMAC output, which is typically the size of the hash algorithm's output (e.g., 32 bytes for
SHA256).
 * @param buffer_size The size of the buffer. It must be at least as large as the HMAC output size to avoid buffer overflows.
 *
 * @return Pointer to the buffer containing the HMAC string on success, or NULL on failure. If the function fails, it may be
 *         due to reasons such as an unsupported algorithm, incorrect buffer size, or memory allocation issues.
 */
extern char *hmac(
    const char *const str, const size_t str_length,
    const char *const key, const size_t key_length,
    const char *const algorithm,
    char *const buffer, const size_t buffer_size);
```

问题五：缺乏 C++ 支持。

在 hmac.h 中，函数 hmac 的声明未针对 C 和 C++ 的混合项目进行处理。

这违背了接口所需的通用性，即接口的适用范围被限制为纯 C 或纯 C++ 项目。故应该进行修正，用条件编译和 extern "C" 对 hmac 函数的声明进行包围，防止其在 C++ 编译单元中的名字被重新映射，即

```
#ifdef __cplusplus
extern "C" {
#include <cstddef>
#else
#include <stddef.h>
#endif

// The declaration of hmac

#ifdef __cplusplus
}
#endif
```

问题六：不恰当的数据类型。

在 hmac.h 中，函数 hmac 的声明中关于长度的变量的类型均为 int。

这损害了接口的健壮性和可移植性，若需要计算的字符串长度超过 $2^{31}-1$（约 2 GiB）或者其他平台上 int 的长度不够 32 位，则程序将会出现错误。故应该将关于长度的变量的类型修改为 size_t，之前已经展示过修改后的结果，故此处略去。

问题七：不正确的语言设定。

test.c 的代码中，为了释放由 malloc() 分配的内存，使用了 delete 而非 free()。

delete 是 C++ 的关键字，应该与 new 配套使用。故应该将 test.c 调整为 test.cpp，然后将内存申请部分进行修改，即

```
void test_hmac(const char *const algorithm) {
    const char *const src = "kjhdskfhdskfjhdskjfdskfdskfjsdkfjds";
    const char *const key = "kjshfkds";

    const std::size_t buffer_size = 256;
    char *const buffer = new char[buffer_size];

    const char *const result = hmac(
        src, std::strlen(src),
        key, std::strlen(key),
        algorithm,
        buffer, buffer_size);

    if (result)
        printf("%s result = [%s]\n", algorithm, result);
    else
        printf("ERROR: %s\n", buffer);

    delete[] buffer;

    return;
}
```

问题八：不合理的构建过程。

在提供的 `Makefile` 中，所有代码均使用 `g++` 编译且无变量控制，并且测试时对共享库 `libhmac` 的路径未指明。

这一方面与 `hmac` 是一个 C 语言库不匹配，另一方面也增加了后续修改调试的难度。故应该加上一些变量（`C`, `CFLAGS`, `CXX`, `CXXFLAGS`）以方便配置和编译选项（`-Wl,-rpath,'$ORIGIN'`）以正确加载共享库，即

```
C := gcc
CFLAGS := -Wall -Wextra -O2
CXX := g++
CXXFLAGS := -Wall -Wextra -O2

libhmac.so: hmac.c hmac.h
    $(C) $(CFLAGS) -shared -o libhmac.so hmac.c -lssl -lcrypto

test: test.cpp libhmac.so
    $(CXX) $(CXXFLAGS) -o test test.cpp -L. -lhmac -Wl,-
rpath,'$$ORIGIN'
    ./test

clean:
    rm -f test libhmac.so
```

建议一：接口测试应更全面。

在测试时，不仅应测试其功能，还需要保证其输出结果的正确性和可预测性。

建议二：测试结果识别应该更加自动化。

在测试时，测试结果不应该由人来检查是否正确，而应交由代码完成。

建议三：项目目录不够规范。

源代码应该放在 `src` 目录下，测试源代码应该放在 `test` 目录下，生成的可执行文件和共享库应该放在 `build` 目录下。

修改后的代码见上传文件中的压缩包。