МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ
ХАРКІВСЬКИЙ НАЦІОНАЛЬНИЙ УНІВЕРСИТЕТ РАДІОЕЛЕКТРОНІКИ

Кафедра Програмної інженерії

Звіт
з лабораторної роботи № 3
з дисципліни: «Операційні системи»
з теми: «Створення та використання бібліотек. Частина 2»

Виконав:
ст. гр. ПЗПІ-23-2
Ситник Є. С.

Перевірила:
доц. каф. ПІ
Мельнікова Р. В.

Харків – 2025

3 СТВОРЕННЯ ТА ВИКОРИСТАННЯ БІБЛІОТЕК. ЧАСТИНА 2

3.1 Мета роботи

Вивчити розширені прийоми та методи створення та використання різних типів динамічних бібліотек.

3.2 Хід роботи

3.2.1 Створити новий варіант бібліотеки «rsa» із застосуванням «.def» файлу, та динамічного завантаження під час виконання програми.

Файли із розширенням «.def» використовуються для позначення «символів», що будуть експортовані із «.dll» бібліотеки для зовнішнього використання. Зараз застосування таких файлів можна зустріти дуже рідко, адже його майже повністю замінили атрибути «__declspec(dllexport)» та «__declspec(dllimport)». Оскільки такі атрибути були використані під час виконання попередньої лабораторної роботи нам необхідно їх прибрати із заголовочного файлу нової версії бібліотеки, після цього створимо файл «rsa.def» в який додамо назви функцій, що мають бути експортовані.

```
#ifndef RSA_LIB
#define RSA_LIB

#ifdef __cplusplus
extern "C" {
#endif

typedef struct {
    __int128_t d;
    __int128_t e;
    __int128_t n;
} rsa_Pair;

rsa_Pair rsa_gen_pair(__int128_t p, __int128_t q, __int128_t e);

__int128_t rsa_encrypt(__int128_t data, __int128_t e, __int128_t n);
__int128_t rsa_decrypt(__int128_t data, __int128_t d, __int128_t n);

#ifdef __cplusplus
}
#endif

#endif
```

Рисунок 3.1 – Оновлений вміст файлу «rsa.h»

```
1  LIBRARY rsa.dll
2  EXPORTS
3      rsa_gen_pair @1
4      rsa_encrypt @2
5      rsa_decrypt @3
```

Рисунок 3.2 – Вміст файлу «rsa.def»

Також необхідно передати створений «rsa.def» файл компілятору. Для цього оновимо команду компіляції, додавши шлях до відповідного файлу.

```
@build-rsa-shared-lib: mkdir-build
  {{cc}} {{cc_flags}} --shared rsa-shared/lib/rsa.cpp rsa-shared/lib/rsa.def -o build/rsa.dll -Wl,--out-implib,build/rsa.lib
```

Рисунок 3.3 – Оновлений «рецепт» компіляції бібліотеки «rsa.dll»

Щоб перевірити, що потрібні «символи» експортуються із «rsa.dll» можемо скористатись програмою «dumpbin».

```
~/D/N/O/lb-3/code > dumpbin /exports build/rsa.dll
Microsoft (R) COFF/PE Dumper Version 14.40.33811.0
Copyright (C) Microsoft Corporation.  All rights reserved.


Dump of file build\rsa.dll

File Type: DLL

  Section contains the following exports for rsa.dll

    00000000 characteristics
    67FBDEE6 time date stamp Sun Apr 13 18:57:26 2025
        0.00 version
           1 ordinal base
           3 number of functions
           3 number of names

    ordinal hint RVA      name

          3    0 000021B0 rsa_decrypt
          2    1 000020C0 rsa_encrypt
          1    2 00001F10 rsa_gen_pair
```

Рисунок 3.4 – Перевірка зі допомогою «dumpbin» експортованих із бібліотеки «символів»

Можемо побачити, що потрібні «символи» експортуються із «rsa.dll».

Тепер створимо нову версію програми, в якій використаємо динамічне завантаження бібліотеки під час виконання. Для цього змінимо файл «main.cpp». На самому початку виконання програми нам необхідно завантажити «rsa.dll» файл та експортувати із нього функції, типи яких оголосимо заздалегідь. Також додамо виклик «FreeLibrary» перед кожним місцем завершення програми після завантаження бібліотеки.

```
typedef rsa_Pair (*rsa_gen_pair_t)(__int128_t, __int128_t, __int128_t);
typedef __int128_t (*rsa_encrypt_t)(__int128_t, __int128_t, __int128_t);
typedef __int128_t (*rsa_decrypt_t)(__int128_t, __int128_t, __int128_t);
```

Рисунок 3.5 – Оголошені типи функцій для експорту

```
int main() {
    HMODULE hLib = LoadLibraryA(lpLibFileName: "rsa.dll");
    if (!hLib) {
        fprintf(File: stderr, Format: "Error: Can't load rsa.dll.\n");
        return 1;
    }

#pragma GCC diagnostic push
#pragma GCC diagnostic ignored "-Wcast-function-type"
    rsa_gen_pair_t rsa_gen_pair = (rsa_gen_pair_t)(GetProcAddress(hModule: hLib, lpProcName: "rsa_gen_pair"));
    rsa_encrypt_t rsa_encrypt = (rsa_encrypt_t)(GetProcAddress(hModule: hLib, lpProcName: "rsa_encrypt"));
    rsa_decrypt_t rsa_decrypt = (rsa_decrypt_t)(GetProcAddress(hModule: hLib, lpProcName: "rsa_decrypt"));
#pragma GCC diagnostic pop

    if (!rsa_gen_pair || !rsa_encrypt || !rsa_decrypt) {
        printf(Format: "Error: Can't load functions from rsa.dll\n");

        FreeLibrary(hLibModule: hLib);
        return 1;
    }
```

Рисунок 3.6 – Завантаження бібліотеки та імпорту функцій із неї

Також не забудемо оновити «рецепт» компіляції програми.

```
@build-rsa-shared-app: build-rsa-shared-lib
 {{cc}} {{cc_flags}} rsa-shared/app/main.cpp -o build/rsa-shared.exe
```

Рисунок 3.7 – Оновлений «рецепт» компіляції програми

Перевіримо роботу програми.

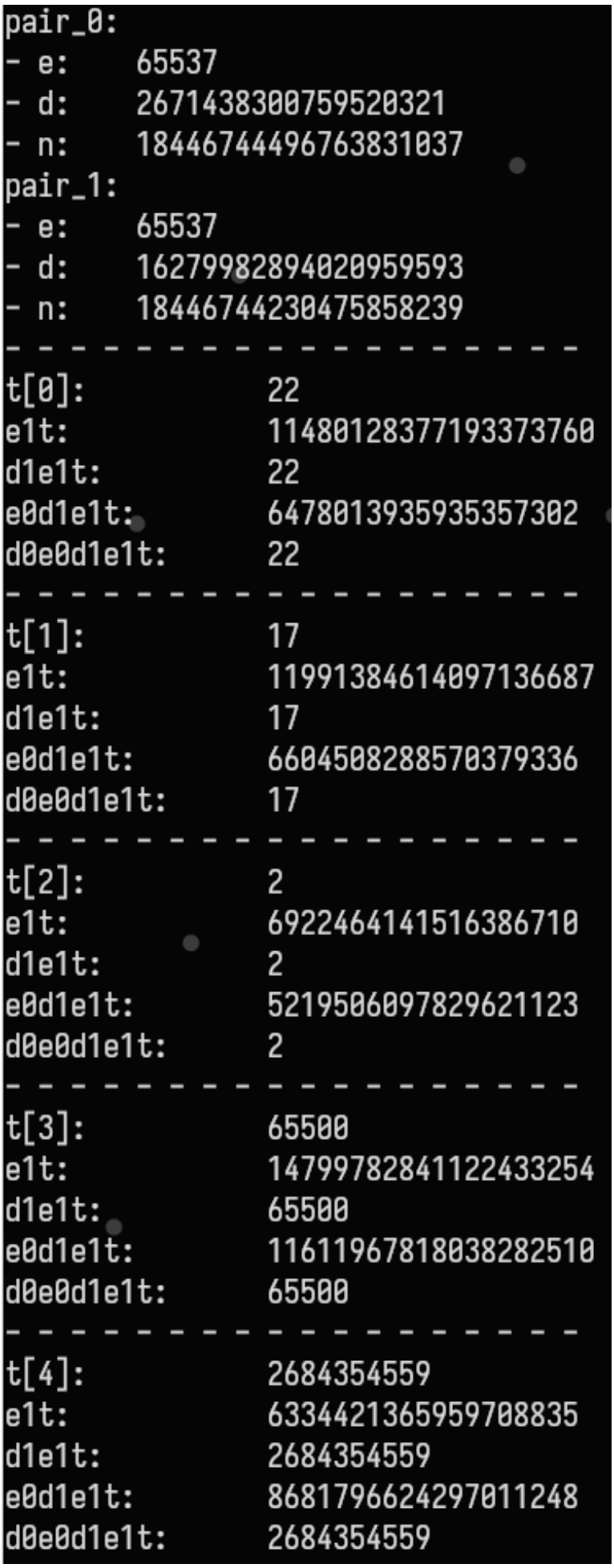

Рисунок 3.8 – Результат виконання програми

Програма працює як і раніше.

3.2.2 Створити файли ресурсів із таблицями рядків українською та англійською мовою, та програму для їх використання.

Файли ресурсів це файли, які містять скрипти, що описують різні не-кодові елементи, які використовуються програмою (наприклад рядки, іконки, бітові карти та інше). Файли ресурсів мають розширення «.rc» та створюються із використанням специфічного синтаксису. Створимо файли «locale.cpp» та «locale.rc» для української та англійської локалізацій, спільний файл «locale.h» для позначення ID ресурсів, та файл «main.cpp» для написання основної програми.

В файлі заголовків визначимо псевдоніми, що ми будемо використовувати замість числових ID.

```
1  #ifndef LOCALE
2  #define LOCALE
3
4  #define ID_SURNAME 100
5  #define ID_FACULTY 101
6  #define ID_GROUP 102
7  #define ID_DISCIPLINE 103
8
9  #endif
```

Рисунок 3.9 – Вміст файлу «locale.h»

В файлах «locale.cpp» оголосимо функції «DLLMain», що викликаються під час завантаження/вивантаження бібліотеки, та надрукуємо інформацію для відлагодження.

```c
#include <stdio.h>
#include <windows.h>

BOOL WINAPI DllMain(HINSTANCE hInstDLL, DWORD fdwReason, LPVOID lpvReserved) {
  switch (fdwReason) {
  case DLL_PROCESS_ATTACH:
    fprintf(File: stderr,
            Format: "DLL loaded into process. hinstDLL = %p, lpvReserved = %p\n",
            hInstDLL, lpvReserved);
    break;

  case DLL_THREAD_ATTACH:
    fprintf(File: stderr,
            Format: "New thread created in the process using the DLL. "
            "hinstDLL = %p, lpvReserved = %p\n",
            hInstDLL, lpvReserved);
    break;

  case DLL_THREAD_DETACH:
    fprintf(File: stderr,
            Format: "Thread using the DLL is terminating. hinstDLL = %p, "
            "lpvReserved = %p\n",
            hInstDLL, lpvReserved);
    break;

  case DLL_PROCESS_DETACH:
    fprintf(File: stderr,
            Format: "DLL unloaded from process. hinstDLL = %p, lpvReserved = "
            "%p\n",
            hInstDLL, lpvReserved);
    break;
  }

  return TRUE;
}
```

Рисунок 3.10 – Вміст файлів «locale.cpp»

В файлах «locale.rc» оголосимо строки, що відповідатимуть прізвищу, факультету, групі та дисципліні двома мовами, використаємо оголошені раніше псевдоніми в якості ID.

```
#include "../locale.h"
#include <windows.h>

LANGUAGE LANG_UKRAINIAN, SUBLANG_DEFAULT

STRINGTABLE
BEGIN
    ID_SURNAME    "Ситник"
    ID_FACULTY    "Комп'ютерних наук"
    ID_GROUP      "ПЗПІ-23-2"
    ID_DISCIPLINE "Операційні Системи"
END
```

Рисунок 3.11 – Вміст файлу «locale.rc» із українською локалізацій

```
#include "../locale.h"
#include <windows.h>

LANGUAGE LANG_ENGLISH, SUBLANG_ENGLISH_US

STRINGTABLE
BEGIN
    ID_SURNAME     "Sytnyk"
    ID_FACULTY     "Computer science"
    ID_GROUP       "PZPI-23-2"
    ID_DISCIPLINE  "Operating Systems"
END
```

Рисунок 3.12 – Вміст файлу «locale.rc» із англійською локалізацією

Створимо «рецепти» для компіляції створених ресурсних файлів. Для перетворення файлів «.rc» у бінарний формат використовується програма «windres», за її допомогою файли «.rc» перетворюються в об’єктні файли «.res», які потім передаються компілятору для створення «.dll». Оскільки я використовую операційну систему ArchLinux я маю додати деякі аргументи для «windres», а саме «-c 65001» для встановлення кодування UTF-8, та «-O coff», щоб вказати формат бінарних файлів, що відповідає формату Windows.

```
@build-locale-ua: mkdir-build
  {{windres}} -c 65001 locale/ua/locale.rc -O coff -o build/locale-en.res
  {{cc}} {{cc_flags}} --shared -Wl,--out-implib,build/locale.lib locale/ua/locale.cpp build/locale-en.res -o build/locale-ua.dll

@build-locale-en: mkdir-build
  {{windres}} -c 65001 locale/en/locale.rc -O coff -o build/locale-en.res
  {{cc}} {{cc_flags}} --shared -Wl,--out-implib,build/locale.lib locale/en/locale.cpp build/locale-en.res -o build/locale-en.dll
```

Рисунок 3.13 – «Рецепти» компіляції файлів ресурсів

В файлі «main.cpp» запитаємо у користувача мову, та завантажимо відповідний цій мові «.dll» файл, після чого завантажимо з цього файлу строки використовуючи визначені раніше псевдоніми, та виведемо ці строки в консоль.

```c
#include "../locale.h"
#include <locale.h>
#include <windows.h>

void print_res_str(HINSTANCE hLib, size_t id) {
    wchar_t buffer[256] = {[0]=0};
    if (LoadStringW(hInstance: hLib, uID: id, lpBuffer: buffer, cchBufferMax: sizeof(buffer) / sizeof(wchar_t)))
        wprintf(Format: L"%ls\n", buffer);
    else
        fprintf(File: stderr, Format: "Error: Can't load string from dll\n");
}

int main() {
    setlocale(Category: LC_ALL, Locale: "ukr");

    printf(Format: "Choose language to use\n");
    printf(Format: "\t1 - ua\n");
    printf(Format: "\t2 - en\n");
    printf(Format: "choice: ");

    int lang = getchar() - '0';

    HMODULE hLib = 0;

    if (lang == 1)
        hLib = LoadLibrary(lpLibFileName: "locale-ua.dll");
    else if (lang == 2)
        hLib = LoadLibrary(lpLibFileName: "locale-en.dll");
    else
        fprintf(File: stderr, Format: "Error: No such language\n");

    if (!hLib) {
        fprintf(File: stderr, Format: "Error: Can't load dll\n");
        return 1;
    }

    printf(Format: "Surname:\t");
    print_res_str(hLib, id: ID_SURNAME);

    printf(Format: "Faculty:\t");
    print_res_str(hLib, id: ID_FACULTY);

    printf(Format: "Group:\t\t");
    print_res_str(hLib, id: ID_GROUP);

    printf(Format: "Discipline:\t");
    print_res_str(hLib, id: ID_DISCIPLINE);

    FreeLibrary(hLibModule: hLib);
    return 0;
}
```

Рисунок 3.14 – Вміст файлу «main.cpp»

Створимо «рецепт» для компіляції програми.

```
@build-locale-app: build-locale-ua build-locale-en
    {{cc}} {{cc_flags}} locale/app/main.cpp -o build/locale.exe
```

Рисунок 3.15 – «Рецепти» компіляції програми

Перевіримо роботу програми із обома мовами.

```
Choose language to use
        1 - ua
        2 - en
choice: 1
DLL loaded into process. hinstDLL = 00006FFFFC6D0000, lpvReserved = 0000000000000000
Surname:        Ситник
Faculty:        Комп'ютерних наук
Group:          ПЗПІ-23-2
Discipline:     Операційні Системи
DLL unloaded from process. hinstDLL = 00006FFFFC6D0000, lpvReserved = 0000000000000000
```

Рисунок 3.16 – Результат виконання із українською мовою

```
Choose language to use
        1 - ua
        2 - en
choice: 2
DLL loaded into process. hinstDLL = 00006FFFFC6D0000, lpvReserved = 0000000000000000
Surname:        Sytnyk
Faculty:        Computer science
Group:          PZPI-23-2
Discipline:     Operating Systems
DLL unloaded from process. hinstDLL = 00006FFFFC6D0000, lpvReserved = 0000000000000000
```

Рисунок 3.17 – Результат виконання із англійською мовою

3.2.3 Створити новий проект для підрахунку контрольної суми за допомогою алгоритму «crc32» та інтегрувати його в бібліотеку «rsa».

Алгоритм «crc32» – це алгоритм обчислення контрольної суми, який використовується для перевірки цілісності даних. Його основна мета це виявлення випадкових помилок передачі та зберігання (перевірка цілісності) даних, але не захист від навмисних змін даних шахраями, адже існують способи навмисно змінити дані зберігши при цьому ту саму контрольну суму.

Створимо новий проєкт, що складатиметься із бібліотеки та консольного застосунку. Бібліотека буде надавати функції для генерації контрольної суми,

додавання згенерованої контрольної суми до файлу, та перевірки цілісності файлу на основі доданої до нього контрольної суми. Консольний застосунок виконуватиме 2 команди – додавання контрольної суми до файлу, та перевірка файлу.

Створимо файл заголовків «crc.h», в якому оголосимо відповідні функції, та файл «crc.def», для експорту функцій із «.dll».

```
#ifndef CRC_LIB
#define CRC_LIB

#include <stdint.h>

#ifdef __cplusplus
extern "C" {
#endif

uint32_t crc_compute(uint8_t *buf, size_t size);

bool crc_append(char *path);

bool crc_verify(char *path);

#ifdef __cplusplus
}
#endif
```

Рисунок 3.18 – Вміст файлу «crc.h»

```
LIBRARY crc.dll
EXPORTS
    crc_compute @1
    crc_append @2
    crc_verify @3
```

Рисунок 3.19 – Вміст файлу «crc.def»

Алгоритм «crc32» передбачає побітове ділення даних на поліном, що може бути досить повільним процесом, особливо для великих обсягів даних. Для

прискорення цієї операції перед початком алгоритму можна обрахувати проміжні значення та зберегти їх в таблицю (для 1 байту це всього 256 унікальних значень). Створимо таблицю для збереження попередньо обрахованих значень, та функцію, що їх обрахує.

```
uint32_t crc_sym_table[256];

bool crc_sym_table_computed = false;

void crc_make_table(void) {
  for (size_t n = 0; n < 256; n++) {
    uint32_t c = n;

    for (size_t k = 0; k < 8; k++) {
      if (c & 1)
        c = 0xEDB88320 ^ (c >> 1);
      else
        c >>= 1;
    }

    crc_sym_table[n] = c;
  }

  crc_sym_table_computed = true;
}
```

Рисунок 3.20 – Функція наповнення таблиці

Реалізуємо функцію, що буде обраховувати контрольну суму для наданого буферу.

```c
uint32_t crc_update(uint32_t crc, uint8_t *buf, size_t size) {
  uint32_t c = crc ^ 0xFFFFFFFF;

  if (!crc_sym_table_computed)
    crc_make_table();

  for (size_t n = 0; n < size; n++)
    c = crc_sym_table[(c ^ buf[n]) & 0xFF] ^ (c >> 8);

  return c ^ 0xFFFFFFFF;
}

uint32_t crc_compute(uint8_t *buf, size_t size) {
  return crc_update(crc: 0, buf, size);
}
```

Рисунок 3.21 – Функція обрахунку контрольної суми для буферу

Створимо функцію, що відкриватиме файл за вказаною адресою, обраховуватиме його контрольну суму, та додаватиме обраховане значення в кінець файлу.

```
bool crc_append(char *path) {
  FILE *input_file = fopen(Filename: path, Mode: "rb");
  if (!input_file) { ↙ 4

  fseek(File: input_file, Offset: 0, Origin: SEEK_END);
  size_t file_size = ftell(File: input_file);
  fseek(File: input_file, Offset: 0, Origin: SEEK_SET);

  uint8_t *buf = (uint8_t *)malloc(Size: file_size);
  if (!buf) { ↙ 5

  size_t read_size = fread(DstBuf: buf, ElementSize: 1, Count: file_size, File: input_file);
  if (read_size ≠ file_size) { ↙ 6

  fclose(File: input_file);

  uint32_t computed = crc_compute(buf, size: file_size);
  free(Memory: buf);

  fprintf(File: stderr, Format: "Info: Computed RSA is %u\n", computed);

  FILE *output_file = fopen(Filename: path, Mode: "ab");
  if (!output_file) { ↙ 4

  if (fwrite(Str: &computed, Size: sizeof(computed), Count: 1, File: output_file) ≠ 1) { ↙ 5

  fclose(File: output_file);
  return true;
}
```

Рисунок 3.22 – Функція додавання контрольної суми до файлу

Створимо функцію, що відкриватиме файл за вказаною адресою, зчитуватиме збережену в ньому контрольну суму, обраховуватиме контрольну суму повторно, та порівнюватиме збережене та обраховане значення.

```
bool crc_verify(char *path) {
  FILE *input_file = fopen(Filename: path, Mode: "rb");
  if (!input_file) { ↙ 4

  fseek(File: input_file, Offset: 0, Origin: SEEK_END);
  size_t file_size = ftell(File: input_file);
  if (file_size < 4) { ↙ 6

  size_t data_size = file_size - 4;
  fseek(File: input_file, Offset: 0, Origin: SEEK_SET);

  uint8_t *buf = (uint8_t *)malloc(Size: data_size);
  if (!buf) { ↙ 5

  size_t read_size = fread(DstBuf: buf, ElementSize: 1, Count: data_size, File: input_file);
  if (read_size ≠ data_size) { ↙ 8

  uint32_t computed = crc_compute(buf, size: data_size);
  uint32_t stored = 0;

  if (fread(DstBuf: &stored, ElementSize: sizeof(stored), Count: 1, File: input_file) ≠ 1) { ↙ 6

  fclose(File: input_file);
  free(Memory: buf);

  fprintf(File: stderr, Format: "Info: Computed RSA is %u\n", computed);
  fprintf(File: stderr, Format: "Info: Stored RSA is %u\n", stored);

  return computed == stored;
}
```

Рисунок 3.23 – Функція перевірки цілісності файлу

Створимо консольний застосунок, що прийматиме 2 аргументи – режим роботи та шлях до файлу, та викликатиме відповідні режиму функції.

```c
#include "../lib/crc.h"
#include <stdio.h>
#include <string.h>

void usage() {
  fprintf(File: stderr, Format: "Usage: checksum <COMMAND> <INPUT>\n");
  fprintf(File: stderr, Format: "Commands:\n");
  fprintf(File: stderr, Format: "\tappend - Compute checksum of the file and append it to the end of file.\n");
  fprintf(File: stderr, Format: "\tverify - Verify checksum of the file.\n");
  fprintf(File: stderr, Format: "\n");
  fprintf(File: stderr, Format: "Input:\n");
  fprintf(File: stderr, Format: "\tPath to input file.\n");
}

enum Command {
  CommandUnknown,
  CommandAppend,
  CommandVerify,
};

int main(int argc, char *argv[]) {
  Command command;

  if (argc < 3)
    command = CommandUnknown;
  else if (strcmp(Str1: argv[1], Str2: "append") == 0)
    command = CommandAppend;
  else if (strcmp(Str1: argv[1], Str2: "verify") == 0)
    command = CommandVerify;
  else
    command = CommandUnknown;

  switch (command) {
  case CommandAppend:
    if (crc_append(path: argv[2])) { ↙ 5
    break;

  case CommandVerify:
    if (crc_verify(path: argv[2])) { ↙ 5
    break;

  case CommandUnknown:
  default:
    usage();
    break;
  }

  return 0;
}
```

Рисунок 3.24 – Вміст файлу «main.cpp»

Також додамо «рецепти» для компіляції створених бібліотеки та застосунку.

```
@build-checksum-lib: mkdir-build
 {{cc}} {{cc_flags}} --shared checksum/lib/crc.cpp checksum/lib/crc.def -o build/crc.dll -Wl,--out-implib,build/crc.lib

@build-checksum-app: build-checksum-lib
 {{cc}} {{cc_flags}} checksum/app/main.cpp -L build -l crc -o build/checksum.exe
```

Рисунок 3.25 – Створені «рецепти» компіляції

Перевіримо роботу програми на текстовому файлі.

```
~/D/N/O/lb-3/code > just build-checksum-app
~/D/N/O/lb-3/code > echo "test text" > test.txt
~/D/N/O/lb-3/code > build/checksum.exe append test.txt
Info: Computed RSA is 3326526264
Checksum appended.
~/D/N/O/lb-3/code > cat test.txt
test text
8F
~/D/N/O/lb-3/code > build/checksum.exe verify test.txt
Info: Computed RSA is 3326526264
Info: Stored RSA is 3326526264
Checksum verified.
~/D/N/O/lb-3/code > sed -i "s/test/test 2/" test.txt
~/D/N/O/lb-3/code > cat test.txt
test 2 text
8F
~/D/N/O/lb-3/code > build/checksum.exe verify test.txt
Info: Computed RSA is 4257420541
Info: Stored RSA is 3326526264
Failed to verify checksum.
```

Рисунок 3.26 – Перевірка роботи програми

Можемо побачити, що програма працює як задумано.

Тепер додамо перевірку цілісності в бібліотеку «rsa», використавши функцію перевірки із щойно створеної бібліотеки «crc».

```
BOOL WINAPI DllMain(HINSTANCE hInstDLL, DWORD fdwReason, LPVOID lpvReserved) {
  switch (fdwReason) {

  case DLL_PROCESS_ATTACH:
    fprintf(File: stderr,
            Format: "DLL loaded into process. hinstDLL = %p, lpvReserved = %p\n",
            hInstDLL, lpvReserved);

    {
      fprintf(File: stderr, Format: "Info: Checking integrity of rsa.dll.\n");

      char dll_path[MAX_PATH] = {[0]=0};
      if (GetModuleFileNameA(hModule: hInstDLL, lpFilename: dll_path, nSize: MAX_PATH) == 0) {
        fprintf(File: stderr, Format: "Error: Can't get path to rsa.dll file.\n");

        return FALSE;
      }

      if (!crc_verify(path: dll_path)) {
        fprintf(File: stderr, Format: "Error: Can't verify rsa.dll integrity.\n");

        return FALSE;
      } else {
        fprintf(File: stderr, Format: "Info: Integrity of rsa.dll is verified.\n");
      }
    }
    break;

  case DLL_THREAD_ATTACH: ↙ 7

  case DLL_THREAD_DETACH: ↙ 7

  case DLL_PROCESS_DETACH: ↙ 7
  }

  return TRUE;
}
```

Рисунок 3.27 – Перевірка роботи програми

Протестуємо перевірку цілісності. Для цього запустимо програму, щоб переконатись, що вона працює, потім за допомогою редактору змінимо вміст «rsa.dll» (наприклад видаливши визначення якогось із експортованих символів), та спробуємо запустити програму ще раз.

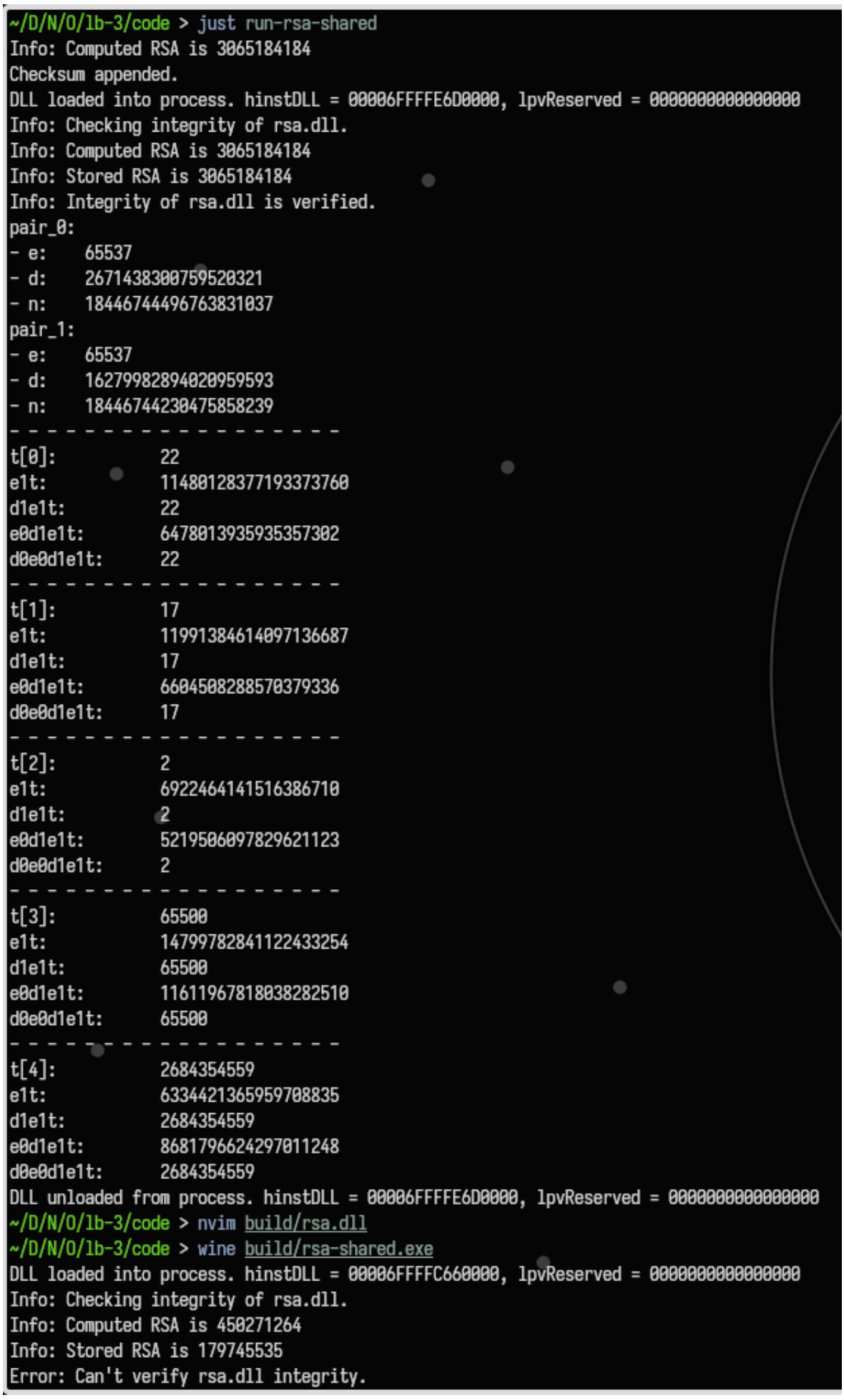

Рисунок 3.28 – Перевірка роботи програми

Можемо побачити, що перевірка цілісності працює, як і очікувалося.

3.3 Висновки

Під час виконання даної лабораторної роботи я вивчив розширені прийоми та методи створення та використання різних типів динамічних бібліотек.