



**МІНІСТЕРСТВО ОСВІТИ, НАУКИ, МОЛОДІ ТА СПОРТУ УКРАЇНИ  
НАЦІОНАЛЬНИЙ ТЕХНІЧНИЙ УНІВЕРСИТЕТ УКРАЇНИ  
«КИЇВСЬКИЙ ПОЛІТЕХНІЧНИЙ ІНСТИТУТ»  
ФІЗИКО-ТЕХНІЧНИЙ ІНСТИТУТ**

**Лабораторна робота 1**

**Аналіз програмного коду мов високого рівня**

***Варіант №5***

**Підготував:**

студент 3 курсу

групи ФІ-84

Коломієць Андрій Юрійович

**Email:** *andkol-ipt22@lil.kpi.ua*

**Викладач:**

**Київ – 2021**

## **Лабораторна робота 1**

### **Аналіз програмного коду мов високого рівня**

#### **Мета роботи**

Отримати навички розпізнавання констукцій мов високого рівня в машинному коді для архітектур **x86/x64** та **ARM/ARM64** на прикладі **C/C++**.

#### **Постановка задачі**

Дослідити машинний код, що відповідає синтаксичним конструкціям **C/C++**.

#### **Завдання**

1. Проаналізувати машинний код прикладу **hanoi.c** для **Windows x64, ARM, ARM64 (MSVC)**, для **Linux amd64, arm, arm64 (GCC)**, для **Linux amd64 (LLVM clang, <http://llvm.org/>)**;

2. Реалізувати мовою **C/C++**, проаналізувати результати компіляції (за варіантом), для платформ **i686, amd64, arm, aarch64**:

- Комбінаторні алгоритми, будь-який на вибір з вказаного класу: **на графах-пошук найкоротшого шляху**.
- Криптографічні алгоритми, алгоритми кодування та контролю цілісності: **Blowfish**;

3. Реалізація функцій стандартної бібліотеки **C**. Бібліотека за варіантами, функції всі зазначені (за наявності реалізації), версія бібліотеки остання стабільна на момент початку курсу: **musl**.

Функції:

- стандартного ввід-виводу **printf, puts**;
- роботи з файлами **fopen, fread, fwrite, feof, fclose**;
- виконання команд операційної системи **system**.

Зверніть увагу на відмінності системних викликів у різних **ОС Linux** та **Windows** для різних архітектур (**x86, x86\_64, ARM**)

#### **Зауваження перед переглядом протоколу**

Аналіз, опис, реалізація згенерованого коду наведені в файлах архіву. Пояснення виділені окремо в вигляді коментарів. Також включено файли, але вже без коментарів, тобто такі, котрі були отримані після генерації (для порівняння). Виконані команди подаються в спрощеному вигляді, але робиться виноска з всіма можливими варіантами вводу.

## Виконання лабораторної роботи

### Завдання 1

Проаналізуємо машинний код прикладу **hanoi.c** для **Windows x64, ARM, ARM64 (MSVC)**, для **Linux amd64, arm, arm64 (GCC)**, для **Linux amd64 (LLVM clang, <http://llvm.org>)**;

#### Розв'язок:

#### MSVC

```
C:\Program Files (x86)\Microsoft Visual Studio\2019\Community\VC\Auxiliary\Build > vcvarsall.bat /help
```

Syntax

```
vcvarsall.bat [arch] [platform_type] [w insdk_version] [-vcvars_ver=vc_version] [-vcvars_spectre_libs=spectre_mode]
```

where :

```
[ arch ]: x86 | amd64 | x86_amd64 | x86_arm | x86_arm64 | amd64_x86 | amd64_arm | amd64_arm64
```

```
////////// additional information //////////
```

```
C:\Program Files(x86)\Microsoft Visual Studio\2019\Community\VC\Auxiliary\Build>vcvarsall.bat [architecture]
```

```
*****
```

```
** Visual Studio 2019 Developer Command Prompt v16 .4.6
```

```
** Copyright (c) 2019 Microsoft Corporation
```

```
*****
```

```
[vcvarsall.bat] Environment initialized for : '[architecture]'
```

[architecture]:  
x64, ARM, ARM64

```
>cl /FAcsu hanoi.c
```

```
Microsoft(R) C/C++ Optimizing Compiler Version 19.24.28319 for x86
```

```
Copyright(C) Microsoft Corporation . All rights reserved .
```

```
hanoi.c
```

```
Microsoft(c) Incremental Linker Version 14.24.28319.0
```

```
Copyright(c) Microsoft Corporation. All rights reserved .
```

```
/out: hanoi.exe
```

```
hanoi.obj
```

## GCC

Можна отримати машинний код двома шляхами, але в данному прикладі створювався асемблерний лістинг.

Компіляція:

№	Вибір компілятора	Команди котрі потрібні виконати для одного вибраного компілятора в послідовності згідно вказаної колонки номерів
1	\$gcc \$i686-linux-gnu-gcc \$arm-linux-gnueabi-gcc \$aarch64-linux-gnu-gcc	-c hanoi.c
2		-S hanoi.c
3		hanoi.c -o hanoi
4		objdump -D hanoi.o > hanoi.[architecture].lst

Асемблерний лістинг:

[architecture]:  
amd64, arm, arm64

```
$ gcc -Wa, -adhln -g hanoi.c > hanoi.amd64.lst  
$ i686-linux-gnu-gcc -Wa, -adhln -g hanoi.c > hanoi.i686.lst  
$ arm-linux-gnueabi-gcc -Wa, -adhln -g hanoi.c > hanoi.arm.lst  
$ aarch64-linux-gnu-gcc -Wa, -adhln -g hanoi.c > hanoi.aarch64.lst
```

## LLVM

Компіляція:

```
$clang -c hanoi.c  
$clang -S hanoi.c  
$clang hanoi.c -o hanoi  
$objdump -D hanoi.o > hanoi.amd64.lst
```

В теці з файлами, наявні всі файли компіляції, та дизасемблювання. Також наведено коментарі, щодо коду асемблера для різних компіляторів та різних архітектур, тобто зроблений аналіз.

**Теоретичні відомості:**

*Yurichev Dennis. Reverse Engineering for Beginners. — 2020. — Режим доступу: <https://beginners.re/>*

## Завдання 2

Реалізуємо мовою C/C++, проаналізуємо результати компіляції, для платформ **i686, amd64, arm, aarch64**:

- Комбінаторні алгоритми, будь-який на вибір з вказаного класу: **на графах-пошук найкоротшого шляху**.
- Криптографічні алгоритми, алгоритми кодування та контролю цілісності: **Blowfish**;

**Розв'язок:**

### Комбінаторні алгоритми, будь-який на вибір з вказаного класу: на графах-пошук найкоротшого шляху

Всі наведені команди компіляції можна зробити наступним чином:

№	Вибір компілятора	Команди котрі потрібні виконати для одного вибраного компілятора в послідовності згідно вказаної колонки номерів
1	\$gcc \$i686-linux-gnu-gcc \$arm-linux-gnueabi-gcc \$aarch64-linux-gnu-gcc	-c Dijkstra.c
2		-S Dijkstra.c
3		Dijkstra.c -o Dijkstra
4		-D Dijkstra.o > Dijkstra.[architecture].lst

[architecture]:  
i686, amd64, arm, aarch64

Асемблерний лістинг можливий, але не потребує завдання лабораторної роботи, тому що аналіз коду стає лекшим.

```
$ gcc -Wa , - adhl -g Dijkstra . c > Dijkstra. amd64 . lst  
$ i686 - linux - gnu - gcc -Wa , - adhl -g Dijkstra. c > Dijkstra . i686 . lst  
$ arm - linux - gnueabi - gcc -Wa , - adhl -g Dijkstra. c > Dijkstra . arm . lst  
$ aarch64 - linux - gnu - gcc -Wa , - adhl -g Dijkstra . c > Dijkstra. aarch64 . Lst
```

### Криптографічні алгоритми, алгоритми кодування та контролю цілісності: Blowfish

№	Вибір компілятора	Команди котрі потрібні виконати для одного вибраного компілятора в послідовності згідно вказаної колонки номерів
1	\$gcc \$i686-linux-gnu-gcc \$arm-linux-gnueabi-gcc \$aarch64-linux-gnu-gcc	-c blowfish.c
2		-S blowfish.c
3		blowfish.c -o blowfish
4		-D blowfish.o > blowfish.[architecture].lst

[architecture]:  
i686, amd64, arm, aarch64

Асемблерний лістинг можливий, але не потребує завдання лабораторної роботи:

```
$ gcc -Wa , - adhl -g blowfish. c > blowfish. amd64 . lst  
$ i686 - linux - gnu - gcc -Wa , - adhl -g blowfish. c > blowfish . i686 . lst  
$ arm - linux - gnueabi - gcc -Wa , - adhl -g blowfish. c > blowfish . arm . lst  
$ aarch64 - linux - gnu - gcc -Wa , - adhl -g blowfish . c > blowfish. aarch64 . Lst
```

## Компіляція

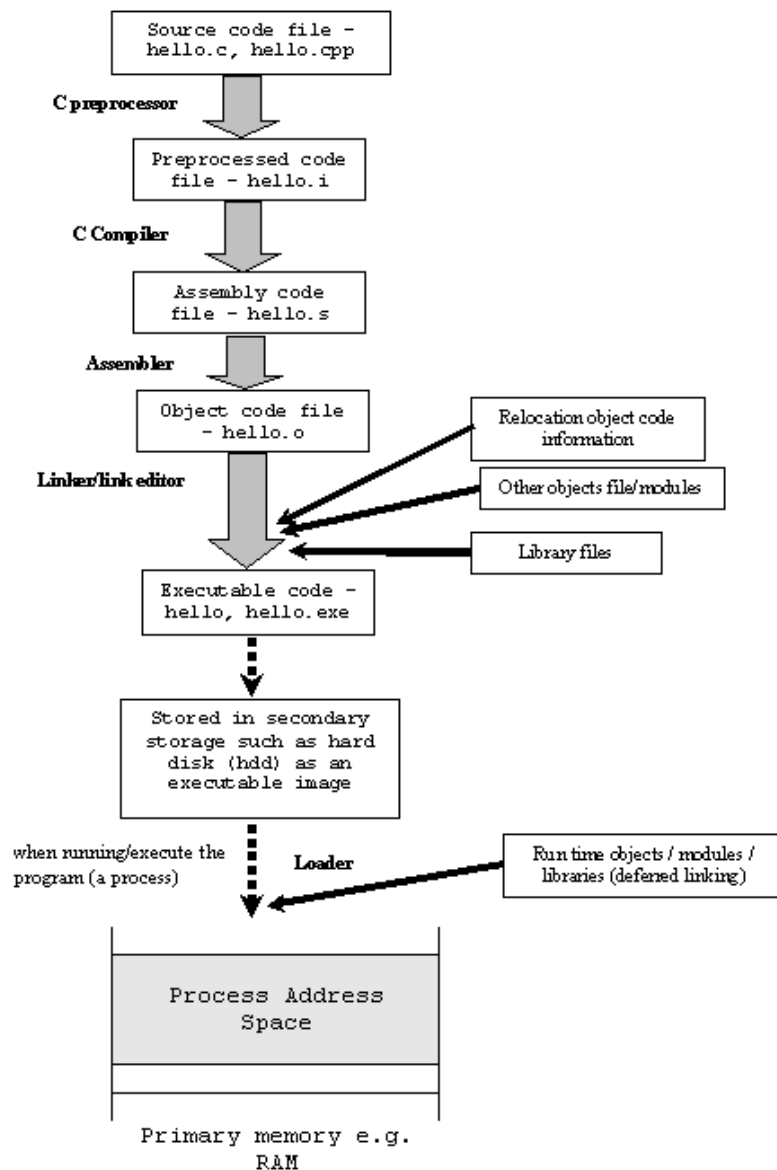
*Компіляція* - це процес перетворення наших програмних файлів C у виконуваний файл.

*Виконаний файл* - це файл особливого типу, який містить машинні інструкції (одиниці та нулі), і запуск цього файлу змушує комп'ютер виконувати ці вказівки.

Процес компіляції включає чотири етапи і використовує різні "інструменти", такі як:

- препроцесор;
- компілятор;
- асемблер;
- компоновальник.

Детальний процес ілюструє картинка.



*Препроцесор*- це перший прохід будь -якої компіляції **C**. Він видаляє коментарі, розширює файли включення та макроси, а також обробляє інструкції умовної компіляції. Це можна вивести як файл з розширенням **.i**.

*Компіляція* - це другий прохід. Він бере вихідні дані препроцесора та вихідний код та генерує вихідний код асемблера (**hello.s**). Мова асемблера-це мова програмування низького рівня (навіть нижча за **C**), яка все ще читається людиною, але складається з мнемонічних інструкцій, які мають чітке відповідність машинним інструкціям.

*Асемблювання*- це третій етап складання. Він бере вихідний код збірки і створює файл об'єкта **hello.o**, який містить фактичні машинні інструкції та символи (наприклад, назви функцій), які більше не читаються людиною, оскільки вони є бітами.

*Компонувальник*- є завершальним етапом компіляції. Він бере один або декілька об'єктних файлів або бібліотек як вхідні дані та об'єднує їх для створення одного (зазвичай виконуваного) файлу. При цьому він вирішує посилання на зовнішні символи, призначає кінцеві адреси процедурам/функціям та змінним, а також переглядає код та дані для відображення нових адрес (процес називається переміщенням).

*Всі файли компіляції наведено в теках, але також було зроблено по можливості дизасемблювання, тому що аналіз мови асемблера і результатів компіляцію відразу стають очевидними по аналогії з першим завданням.*

### **Теоретичні відомості:**

1. <https://automotivetechis.wordpress.com/2012/06/07/memory-map-in-c/>
2. <http://cs.brown.edu/courses/csci1310/2020/>
3. <https://stackoverflow.com/questions/36002159/how-to-open-o-file/>
4. [https://en.wikipedia.org/wiki/Comparison\\_of\\_cryptography\\_libraries/](https://en.wikipedia.org/wiki/Comparison_of_cryptography_libraries/)
5. [https://en.wikipedia.org/wiki/Symmetric-key\\_algorithm/](https://en.wikipedia.org/wiki/Symmetric-key_algorithm/)
6. [https://en.wikipedia.org/wiki/List\\_of\\_algorithms/](https://en.wikipedia.org/wiki/List_of_algorithms/)

### **Завдання 3**

Реалізація функцій стандартної бібліотеки **C**. Бібліотека за варіантами, функції всі зазначені (за наявності реалізації), версія бібліотеки остання стабільна на момент початку курсу: **musl**.

Функції:

- стандартного ввід-виводу **printf, puts**;
- роботи з файлами **fopen, fread, fwrite, feof, fclose**;
- виконання команд операційної системи **system**.

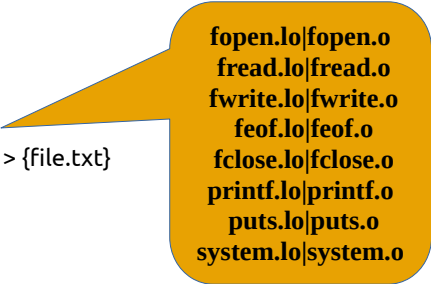
Зверніть увагу на відмінності системних викликів у різних **OC Linux** та **Windows** для різних архітектур (**x86, x86\_64, ARM**)

### **Розв'язок:**

У бібліотеці згідно варіанту, вже наявні файли з котрих можна видобути реалізацію зазначених функцій на асемблері.

Для цього можна виконати наступну команду.

```
$objdump -D {file.lo | file.o} > {file.txt}
```



fopen.lo|fopen.o  
fread.lo|fread.o  
fwrite.lo|fwrite.o  
feof.lo|feof.o  
fclose.lo|fclose.o  
printf.lo|printf.o  
puts.lo|puts.o  
system.lo|system.o

Реалізація функцій наявна вже була при завантаженні бібліотеки, з файлами з типом розширення **.lo**. Детальніше про системні виклики можна прочитати додаткову літературу, більшість операцій наведених в коді, аналогічні тим, що містилися в файлах першого, та другого завдання.

### **Теоретичні відомості:**

1. <https://musl.libc.org/>
2. <https://syscalls.w3challs.com/>