Dr_Quine

Summary: This project is about making you discover the recursion theorem of Kleene!

Version: 4.0

Contents

Chapter I

Foreword

Chapter II

Introduction

A quine is a computer program (a kind of metaprogram) whose output and source code are identical. As a challenge or for fun, some programmers try to write the shortest quine in a given programming language.

The operation that consists of simply opening the source file and displaying it is considered cheating. More generally, a program that uses any data entry cannot be considered a valid quine. A trivial solution is a program whose source code is empty. Indeed, the execution of such a program produces for most languages no output; that is to say, the source code of the program.

Chapter III

Objectives

This project invite you to confront the principle of self-reproduction and the problems that derive from it. It is a perfect introduction to more complex projects, particularly malware projects.

For the curious ones, we strongly recommend that you watch everything related to fixed points!

Chapter IV

General Instructions

- This project will be corrected by humans only.
- **Both the C and the Assembly implementations are required** for all mandatory programs. Your repository must contain two top-level folders named `C` and `ASM`. Each folder must contain its own `Makefile` with the usual rules.
- Your `Makefile` must compile the project and must contain the usual rules. It must recompile and re-link the program only if necessary.
- You have to handle errors carefully. In no way can your program quit in an unexpected manner (Segmentation fault, bus error, double free, etc.).
- You can ask your questions on the forum, on slack...

Assembly part

- Expected Assembly architecture: x86-64 running on **Linux**.
- For example, with NASM:

```
nasm -f elf64 Colleen.s -o Colleen.o && gcc Colleen.o -o Colleen
```

Chapter V

Mandatory Part

For this project, you will have to recode three different programs, each with different properties. Each program will have to be coded in C and in Assembly, placed respectively in folders named `C` and `ASM`, each folder containing its own `Makefile` with the usual rules.

Both the C and the Assembly implementations are mandatory for validation. The absence of either implementation for any mandatory program results in a failure of the mandatory part.

Program #1 — Colleen

- The executable must be named `Colleen`.
- When executed, the program must display on the standard output an output identical to the source code of the file used to compile the program.
- The **C source code** must contain at least:
  - A `main` function.
  - Two different comments.
  - One of the comments must be present in the `main` function.
  - One of the comments must be present outside of your program.
  - Another function in addition to the `main` function (which of course will be called).
- The **Assembly source code** must contain at least:
  - A clear entry point (e.g., `_start` or a symbol linked to `main` as per your toolchain).
  - Two different comments.
  - One comment must be present within the entry point or its immediately-called routine.

- One comment must be present outside of the entry point routine.
- One additional routine/function called from the entry point.

C example:

```
$> ls -al
total 12
drwxr-xr-x 2 root root 4096 Feb 2 13:26 .
drwxr-xr-x 4 root root 4096 Feb 2 13:26 ..
-rw-r--r-- 1 root root 647 Feb 2 13:26 Colleen.c
$> clang -Wall -Wextra -Werror -o Colleen Colleen.c; ./Colleen > tmp_Colleen ; diff tmp_Colleen
    Colleen.c
$> _
```

Assembly example:

```
$> ls -al
total 12
drwxr-xr-x 2 root root 4096 Feb 2 13:26 .
drwxr-xr-x 4 root root 4096 Feb 2 13:26 ..
-rw-r--r-- 1 root root 712 Feb 2 13:26 Colleen.s
$> nasm -f elf64 Colleen.s -o Colleen.o && gcc Colleen.o -o Colleen
$> ./Colleen > tmp_Colleen ; diff tmp_Colleen Colleen.s
$> _
```

Program #2 — Grace

- The executable must be named `Grace`.
- When executed, the program writes in a file named `Grace_kid.c` / `Grace_kid.s` the source code of the file used to compile the program.
- The **C source code** must strictly contain:
  - No `main` declared (no functions declared at all).
  - Exactly three `#define`.
  - One comment.
- The program will run by calling a macro.
- The **Assembly source code** must strictly contain:
  - No extra routines beyond the entry point needed to perform the task (no additional procedures).
  - Exactly three macros (or the closest equivalent for your assembler).
  - One comment.

C example:

```
$> ls -al
total 12
drwxr-xr-x 2 root root 4096 Feb 2 13:30 .
drwxr-xr-x 4 root root 4096 Feb 2 13:29 ..
-rw-r--r-- 1 root root 362 Feb 2 13:30 Grace.c
$> clang -Wall -Wextra -Werror -o Grace Grace.c; ./Grace ; diff Grace.c Grace_kid.c
$> ls -al
total 24
drwxr-xr-x 2 root root 4096 Feb 2 13:30 .
drwxr-xr-x 4 root root 4096 Feb 2 13:29 ..
-rwxr-xr-x 1 root root 7240 Feb 2 13:30 Grace
-rw-r--r-- 1 root root 362 Feb 2 13:30 Grace.c
-rw-r--r-- 1 root root 362 Feb 2 13:30 Grace_kid.c
$> _
```

Assembly example:

```
$> ls -al
total 12
drwxr-xr-x 2 root root 4096 Feb 2 13:30 .
drwxr-xr-x 4 root root 4096 Feb 2 13:29 ..
-rw-r--r-- 1 root root 388 Feb 2 13:30 Grace.s
$> nasm -f elf64 Grace.s -o Grace.o && gcc Grace.o -o Grace
$> rm -f Grace_kid.s ; ./Grace ; diff Grace_kid.s Grace.s
$> ls -al
total 24
drwxr-xr-x 2 root root 4096 Feb 2 13:30 .
```

```
drwxr-xr-x 4 root root 4096 Feb 2 13:29 ..
-rwxr-xr-x 1 root root 7240 Feb 2 13:30 Grace
-rw-r--r-- 1 root root 388 Feb 2 13:30 Grace.s
-rw-r--r-- 1 root root 388 Feb 2 13:30 Grace_kid.s
$> _
```

Program #3 — Sully

- The executable must be named `Sully`.
- When executed the program writes in a file named `Sully_X.c` / `Sully_X.s`. The X will be an integer given in the source. Once the file is created, the program compiles this file and then runs the new program (which will have the name of its source file).
- Stopping the program depends on the file name: the resulting program will be executed only if the integer X is greater or equals than 0.
- An integer is therefore present in the source of your program and will have to evolve by decrementing every time you create a source file from the execution of the program.
- You have no constraints on the source code, apart from the integer that will be set to 5 at first.

C example:

```
$> clang -Wall -Wextra -Werror ../Sully.c -o Sully ; ./Sully
$> ls -al | grep Sully | wc -l
13
$> diff ../Sully.c Sully_0.c
1c1
< int i = 5;
---
> int i = 0;
$> diff Sully_3.c Sully_2.c
1c1
< int i = 3;
---
> int i = 2;
$> _
```

Assembly example:

```
$> nasm -f elf64 ../Sully.s -o Sully.o && gcc Sully.o -o Sully
$> ./Sully
$> ls -al | grep Sully | wc -l
13
$> diff ../Sully.s Sully_0.s
1c1
< ; i = 5
---
> ; i = 0
$> diff Sully_3.s Sully_2.s
1c1
< ; i = 3
---
> ; i = 2
$> _
```

A comment must look like:

```
$> nl comment.c
 1   /*
 2      This program will print its own source when run.
 3   */
```

A program without a declared main must look like:

```
$> nl macro.c
 1  #include
 2  #define FT(x)int main(){ /* code */ }
    [..]
 5  FT(xxxxxxxx)
```

Using advanced macros is strongly recommended for this project.

For the smarty pants (or not)... just reading the source and displaying it is considered to be cheating. The use of argv/argc is also considered cheating.

Chapter VI

Bonus part

The only Bonus accepted during the evaluation is to have redone this project entirely in the language of your choice.

In case of a language without define/macro, you will naturally have to adapt the program accordingly.

For the smarty pants (or not)... just copying the mandatory C code to .cpp files and calling it a different language will not count as bonus.

The bonus part will only be assessed if the mandatory part is PERFECT. Perfect means the mandatory part has been integrally done and works without malfunctioning. If you have not passed ALL the mandatory requirements, your bonus part will not be evaluated at all.

Chapter VII

Submission and peer-evaluation

Turn in your assignment in your `Git` repository as usual. Only the work inside your repository will be evaluated during the defense. Don't hesitate to double check the names of your folders and files to ensure they are correct.