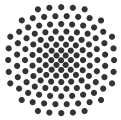

University of Stuttgart

Programming Paradigms Prof. Dr. Michael Pradel

University of Stuttgart, Summer 2022

Exercise 5: Control Abstraction

(Deadline for uploading solutions: June 29, 2022, 11:59pm Stuttgart time)

The materials provided for this homework are:

- a pdf file with the text of the homework (this);
- a zip file with **the folder structure and the templates that must be used for the submission**.

The folder structure is shown in Figure 1.

```
exercise5.zip
├─ task_1/
│  └─ task1.csv (Insert your solution in this file)
├─ task_2/
│  └─ task2.csv (Insert your solution in this file)
├─ task_3/
│  ├─ 1
│  │  └─ main.c (Insert your solution in this file)
│  ├─ 2
│  │  └─ main.c (Insert your solution in this file)
│  └─ 3
│     └─ main.c (Insert your solution in this file)
└─ task_4/
   ├─ 1
   │  └─ main.js (Insert your solution in this file)
   ├─ 2
   │  └─ main.js (Insert your solution in this file)
   └─ 3
      └─ main.js (Insert your solution in this file)
```

Figure 1: Directory structure in the provided .zip file and in your uploaded solution file.

The submission must be compressed in a zip file (not rar, 7z, gz, or anything else) using exactly the folder structure and names in Figure 1. To ensure that your submission has the expected structure, check it with the validator.jar tool provided on the course page, using the following command (where exercise5.zip is the original file provided to you, and your_exercise5.zip is your modified version of it):

```
java -jar validator.jar exercise5.zip your_exercise5.zip
```

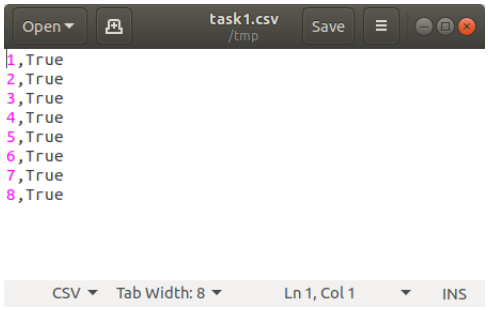

Figure 2: Example of the file containing the answer of Task I.

1 Task I (10% of total points of the exercise)

Which of the following statements are correct? Your answer, either **True** or **False**, must be written into the file *exercise5/task_1/task1.csv* one answer per line as illustrated in Figure 2.

1. Calling a function recursively multiple times can cause a stack overflow error.
2. The best way to avoid a stack smashing vulnerability is to declare variables with larger size than expected.
3. When you pass an argument by reference, you have to manually return the value when the function ends.
4. All programming languages permit passing parameters both by value and by reference.
5. During the execution, when an exception is raised without a *catch* statement, the stack is contracted and all the local variables are destroyed.
6. During the execution of code, the presence of try/catch statements can avoid the crash of the program.
7. In JavaScript, the `Promise()` constructor permits multi-threading execution.
8. In JavaScript, the `catch()` registers a callback to be invoked when the promise is rejected or if an error occurs during the execution.

2 Task II (15% of total points of the exercise)

This task is to check your understanding of some concepts about passing parameters to functions. For that, we give you a code snippet written in pseudo-language.

Code Snippet for Task II

```
1  x = 0
2  y = 1
3  func foo1(x,y):
4          z = x
5          x = y
6          y = z
7          print(x) // Submit your answer
8
9  func foo2(x,y):
10         x = x + 2
11         y = y + 2
12         print(x) // Submit your answer
13
14 foo1(x,y)
15 foo2(x,y)
16 x = x + y
17 print(x) // Submit your answer
```

You have to guess the prints in line 7, 12 and 17 for three different scenarios.

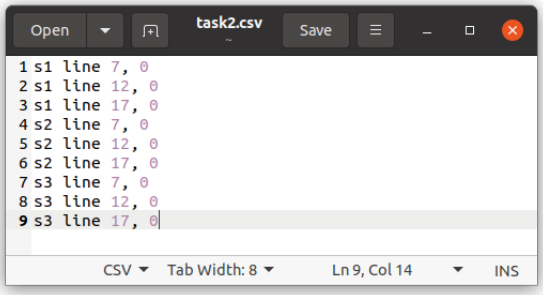

Figure 3: Example of the file containing the answer of Task II.

1. The first scenario (s1) is that *foo1* gets parameters by value and *foo2* gets parameters by reference.
2. The second scenario (s2) is that *foo1* gets parameters by reference and *foo2* gets parameters by value.
3. The third scenario (s3) is that both *foo1* and *foo2* get parameters by value.

To submit your answer, please fill in the file exercise5/task_2/task2.csv. One row corresponds to one print, which corresponds to one answer. In total there are nine rows: three prints for the first scenario (s1), three prints for the second scenario (s2), and three prints for the third scenario (s3). Figure 3 shows an example.

3 Task III (30% of total points of the exercise)

This task is to check your understanding of some concepts about memory management and stack smashing vulnerabilities. We provide three programs written in C that suffer from a memory error. You have to fix and run the code. To fix the code, you have to modify the existing programs without adding or removing entire lines of code. You can use GCC to compile the source code provided.[1] We use GCC version 9.4.0 for our tests. You can compile the source code with the command:

Compile C program

```
gcc -o main main.c
```

Then, you can run the compiled source code with the command :

Run compiled C program

```
./main
```

For this task you have to fix three C programs.

1. The first program is in exercise5/task_3/1/main.c (Figure 4) and if you try to compile it, you receive a warning: "__builtin_memcpy" and if you run the compiled file you get the error: "stack smashing detected". You have to fix the program, so that it compiles and runs without errors.

[1] https://gcc.gnu.org/

Code for task 3.1

```c
#include <stdio.h>
#include <string.h>
#define L 15

int main() {
        char string[L];
        int n, i;

        strcpy(string, "Stack Smashing error?");

        n = 20;

        for(i=0; i < n; i++)
        printf("%3d - %s\n", i, string);

        return 0;
}
```

2. The second program is in exercise5/task_3/2/main.c (Figure 5) and it compiles without warnings, but if you run the compiled file you get the error: ”stack smashing detected”. You have to fix the program, so that it compiles and runs without errors.

Code for task 3.2

```c
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <time.h>

void main()
{
        char div[] = "is divisible";
        char divNot[] = "is not divisible";

        srand(time(NULL));   // Initialization
        int r = rand() % (1000 + 1 - 0) + 0; // Pseudo-random integer

        if ((r % 25) == 0) {
                strcat(div, " by 25.  ");
                printf("%d %s\n", r, div);
        } else{
                strcat(divNot, " by 25.  ");
                printf("%d %s\n", r,  divNot);
        }

        return;
}
```

3. The third program is in exercise5/task_3/3/main.c (Figure 6) and it compiles without warnings, but if you run the compiled file you get the error: ”malloc(): corrupted top size”. You have to fix the program, so that it compiles and runs without errors.

Code for task 3.3

```c
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <time.h>

char *stringEdit(const char *s, const char *t)
{
        char *r = (char *) malloc(strlen(s));
        char *p = r;

        while(*r++ = *s++);
        r--;
        while(*r++ = *t++);
        *r = '\0';

        return p;
}

void main()
{
        char message1[] = "Program Paradigm, Exercise 5: ";
        char message2[] = "Control Abstraction";
        char *result = stringEdit(message1, message2);

        printf("%s", result);

        free(result);

        return;
}
```

Evaluation Criteria: Your solution will be run to check if the memory errors disappear and the output of the program will be checked. Moreover, we will check if you remove or add entire lines of code (which you should not, as explained above).

4 Task IV (45% of total points of the exercise)

This task is to check your understanding of some concepts of promises in JavaScript and error handling. We provide three programs written in JavaScript with specific edits to perform. You have to modify and run the code. You can run JavaScript code using the Node.js runtime with the terminal command: "node main.js".[2] We use Node.js version 14.17.3.
For this task, you have to edit three JavaScript programs.

1. The first program is in exercise5/task_4/1/main.js (Figure 7). If you run the code, you get three times '1' as output. However, we would like a program that reuses the `return` value of the previous `then` in order to have as final output: "1", "2", and "4".

[2] https://nodejs.org/en/download/

Code for task 4.1

```
let promise = new Promise(function(resolve, reject) {
        setTimeout(() => resolve(1), 1000);
});

promise.then(function(result) {
        console.log(result); // 1
        return result * 2;
});

promise.then(function(result) {
        console.log(result); // 1
        return result * 2;
});

promise.then(function(result) {
        console.log(result); // 1
        return result * 2;
});
```

2. The second program is in exercise5/task_4/2/main.js (Figure 8). If you run the code, the last city ('Stuttgart') is not printed, because the function `CreateCities()` is slower than `getCities`. To fix this problem, you have to use the keywords `async` and `await`, so `getCities()` can wait until the end of `createCities()`.

Code for task 4.2

```
const cities = [
{ city: 'Rome', country:'Italy'},
{ city: 'Paris', country: 'France'}
]

function getCities(){
        setTimeout(() => {
                cities.forEach((city, index) => {
                        console.log(city.city)
                })
        }, 1000);
}

function createCities(city){
        setTimeout(() => {
                cities.push(city)
        }, 2000);
}

function init(){
        createCities({ city: 'Stuttgart', country:'Germany' });
        getCities();
}

init();
```

3. The second program is in exercise5/task_4/3/main.js (Figure 9). If you run this code, you get the error: "UnhandledPromiseRejectionWarning". To fix this problem, you have to use the keywords `catch()` method of a promise object to print "Error code 11" and the keyword `finally` to print the content of variable `p`.

Code for task 4.3

```
p = new Promise(function(resolve, reject) {
        reject('Something went wrong!')
});

p.then(function() {
        console.log('Hello World!');
        return result;
});
```

Evaluation Criteria: Your solution will be run and the output will be compared with our master solutions. Moreover, we will check if you achieve the desired behavior by using the requested language constructs, such as promises or `await`.