Lab 2. Stacks and Queues

Release: **29th January 2024**
Supervision: **6th February 2024**
Due: **19th February 2024**

Vladimir Tarasov

1 Task: Parenthesis Checker

1.1 Introduction

In this task, you will implement a parentheses checker using an array-based representation of a stack. A parenthesis checker evaluates parentheses in arithmetic expression.

1.2 Parentheses Checker

A parentheses checker takes as input a string expression of parentheses containing only the characters '(', ')', '[', ']', '{' and '}', and determines if the input string is valid. It prints "valid expression" if the string is balanced, otherwise prints "invalid expression".

An input string is balanced if:

1. An open bracket is closed by the same type of bracket,
2. Open brackets are closed in the correct order.

A program to implement such a parentheses checker using an *array-based*representation of a stack is given in section 7.7.2 in Thareja's book. The steps to do the evaluation are given below.

Steps to evaluate a parentheses expression

1. Define a character stack.
2. Traverse the arithmetic expression.
   - If the current character is a starting bracket '(', '{' or '[', push it to the stack.
   - If the current character is a closing bracket ')', '}' or ']', pop it from the stack. If the popped character matches the starting bracket then the brackets are balanced, otherwise the brackets are not balanced.
3. After a complete traversal, if there are some starting bracket left in the stack, then the input parentheses are unbalance, otherwise they are balanced.

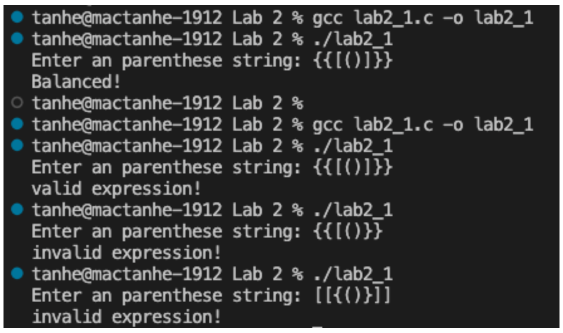
Now extend the program provided in Thareja's book, so that the checker also can evaluate whether an input expression satisfies the following *additional constraints*.

Additional constraints for the checker

- Parentheses `( )` can contain only `( )` brackets.
- Square brackets `[ ]` can contain only `[ ]` and `( )` brackets.
- Curly braces `{ }` can contain `{ }`, `[ ]` and `( )` brackets.

1.3 Requirements

Once your program is implemented correctly, you should be able to produce output like the example shown in Fig 1, where the size of the stack is set to 10. You can choose a different value for the size. This depends on how long input string you allow a user to input.



```
● tanhe@mactanhe-1912 Lab 2 % gcc lab2_1.c -o lab2_1
● tanhe@mactanhe-1912 Lab 2 % ./lab2_1
  Enter an parenthese string: {{[()]}}
  Balanced!
○ tanhe@mactanhe-1912 Lab 2 %
● tanhe@mactanhe-1912 Lab 2 % gcc lab2_1.c -o lab2_1
● tanhe@mactanhe-1912 Lab 2 % ./lab2_1
  Enter an parenthese string: {{[()]}}
  valid expression!
● tanhe@mactanhe-1912 Lab 2 % ./lab2_1
  Enter an parenthese string: {{[()}}
  invalid expression!
● tanhe@mactanhe-1912 Lab 2 % ./lab2_1
  Enter an parenthese string: [[{()}]]
  invalid expression!
```

Figure 1: example output from the checker

1.4 Deliverables

The deliverable is a source code in a zipped file in which the checker described above is implemented. It is up to you to decide how you code the `main()` function. Variables in your code should have proper names, and the code has to include sufficient comments for readability.

The source code has to be thoroughly tested as well as to adhere to the recommendations on *Standard C and portability*, which you can find in Canvas.

You can compile your code using more strict flags `-Wall` together with `-pedantic` before submitting the code, e.g.

```
$ gcc -Wall -pedantic program.c -o program
```

Please include a comment in your program (all `.c` files) with the year and your name to indicate authorship:

```
// 2024 Your Name
```

2 Task: Printer Simulation

2.1 Introduction

Access to shared resources in computer systems is often controlled by a queueing mechanism. A common example is printers. In this lab task you will simulate the scenario of a laboratory printer (as shown in Fig 2).

2.2 Laboratory Printer Scenario

On average there will be one print task in N seconds. The length of the print tasks ranges from 1 to M pages. The printer in the lab can process P pages per minute, i.e., on average a printing task is finished in $pages * (60/P)$ seconds, at good quality.

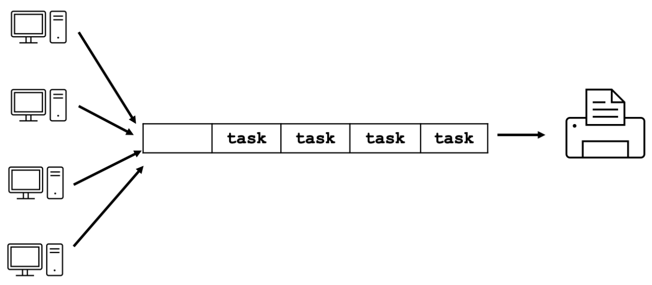


Figure 2: Laboratory printer scenario

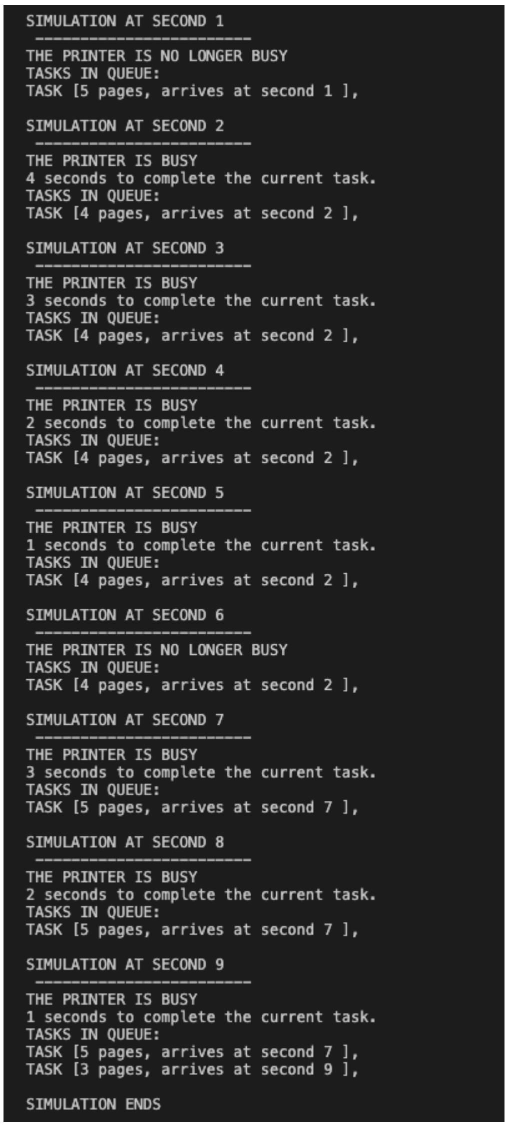
The simulation runs roughly as follows.

Simulation of a laboratory printer

1. Set the printer state to empty.
2. Create a queue for handling print tasks.
3. While (the clock limit is not reached)
    - Is a new print task created? If so, add it to the queue with the current second as the timestamp upon its arrival.
    - If the printer is not busy and if a task is waiting, remove the front task from the queue and assign it to the printer.
    - The printer now does one second of printing if necessary, i.e., subtracts one second from the time required for the task. If the task has been completed, in other words the time required has reached zero, the printer is no longer busy.
4. Simulation ends.

2.3 Requirements

You should implement the interfaces for the printer ADT, task ADT and queue ADT as defined in the header files (in `printer_simulation_headers.zip`), and a main function for the simulation process. As defined in the queue ADT interface, the queue should be implemented using *linked list*. Regarding the main function it is up to you to decide how to code it. Once your program is implemented correctly, you should be able to produce output like the example shown in Fig 3, where $N = 2$, $M = 5$, $P = 60$ and the simulation runs for 10 seconds. You can choose different values for the simulation parameters in your program.



```
SIMULATION AT SECOND 1
 ------------------------
THE PRINTER IS NO LONGER BUSY
TASKS IN QUEUE:
TASK [5 pages, arrives at second 1 ],

SIMULATION AT SECOND 2
 ------------------------
THE PRINTER IS BUSY
4 seconds to complete the current task.
TASKS IN QUEUE:
TASK [4 pages, arrives at second 2 ],

SIMULATION AT SECOND 3
 ------------------------
THE PRINTER IS BUSY
3 seconds to complete the current task.
TASKS IN QUEUE:
TASK [4 pages, arrives at second 2 ],

SIMULATION AT SECOND 4
 ------------------------
THE PRINTER IS BUSY
2 seconds to complete the current task.
TASKS IN QUEUE:
TASK [4 pages, arrives at second 2 ],

SIMULATION AT SECOND 5
 ------------------------
THE PRINTER IS BUSY
1 seconds to complete the current task.
TASKS IN QUEUE:
TASK [4 pages, arrives at second 2 ],

SIMULATION AT SECOND 6
 ------------------------
THE PRINTER IS NO LONGER BUSY
TASKS IN QUEUE:
TASK [4 pages, arrives at second 2 ],

SIMULATION AT SECOND 7
 ------------------------
THE PRINTER IS BUSY
3 seconds to complete the current task.
TASKS IN QUEUE:
TASK [5 pages, arrives at second 7 ],

SIMULATION AT SECOND 8
 ------------------------
THE PRINTER IS BUSY
2 seconds to complete the current task.
TASKS IN QUEUE:
TASK [5 pages, arrives at second 7 ],

SIMULATION AT SECOND 9
 ------------------------
THE PRINTER IS BUSY
1 seconds to complete the current task.
TASKS IN QUEUE:
TASK [5 pages, arrives at second 7 ],
TASK [3 pages, arrives at second 9 ],

SIMULATION ENDS
```

Figure 3: example program output

Hints

- A program to implement queue using linked list can be found in section 8.3 in Thareja's book.
- Generate the length for a printing task using random function, i.e., `1 + rand() % M`.
- Check when to generate a printing task using random function, i.e., `num = 1 + rand() % TASK_IN_N_SECOND`, if `num` equals to `TASK_IN_N_SECOND` then a new printing task get created.

2.4 Deliverables

The deliverable is a zip file in which the files are organized as in Fig 4. Variables in your code should have proper names, and the code has to include sufficient comments for readability.

The provided header files *must be included* in your implementation. It is up to you to decide how you code the `main()` function.

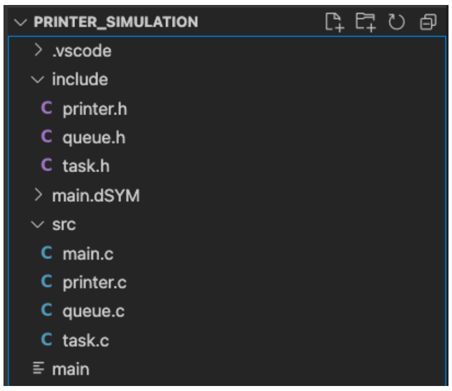
The source code has to be thoroughly tested as well as to adhere to the recommendations on *Standard C and portability*, which you can find in Canvas.

You can compile your code using more strict flags `-Wall` together with `-pedantic` before submitting the code, e.g.

```
$ gcc -Wall -pedantic program.c -o program
```

Please include a comment in your program (all `.c` files) with the year and your name to indicate authorship:

```
// 2024 Your Name
```



```
PRINTER_SIMULATION
  > .vscode
  v include
      printer.h
      queue.h
      task.h
  > main.dSYM
  v src
      main.c
      printer.c
      queue.c
      task.c
    main
```

Figure 4: file organization