Secure and Dependable Systems
Jacobs University Bremen
Dr. Jürgen Schönwälder

Course: CO21-320203
Date: 2019-02-16
Due: 2019-02-28

SADS 2019 Problem Sheet #1

This problem sheet asks you to implement a simple reverse polish notation calculator in C. The emphasis of this assignment is on the unit testing and the test coverage calculation. The header files provided below give some guidance how your code should be structured. (Do not copy a random implementation from the Internet, you will not receive points if the implementation does not follow the guidelines given here and there are no proper unit tests.)

You have to submit your source code together with the test coverage report in a zip file (please remove all build files before creating the archive). You are encouraged to use `cmake` and expected to use the `check`[1] unit testing framework for C.

Problem 1.1: *stack implementation with unit tests* (2+2+1 = 5 points)

a) Implement a generic stack in C. The definition of the API of the generic stack is shown at the end of this sheet.

b) Write unit test cases for your stack first before writing the stack implementation.

c) Use the `gcc` coverage testing features to record the coverage of your test cases. (You may use the `gcd` C project discussed in class as a template.)

Problem 1.2: *reverse polish notation calculator with unit tests* (2+2+1 = 5 points)

a) Using the stack implementation from the first problem, implement a reverse polish calculator (`rpnc`) in C. Here are some example invocations in a shell (the quotes prevent the expansion of the `*` by the shell).

```
$ rpnc 42
42
$ rpnc 2 3 +
5
$ rpnc 2 3 '*'
6
$ rpnc 2 3 + 2 '*'
10
```

The calculator should implement integer arithmetic and the operators `+`, `-`, `*`, `/`, `%`. It should handle error cases:

```
$ rpnc foo
rpnc: invalid token 'foo'
$ rpnc 2 +
rpnc: missing operand
$ rpnc 2 4
rpnc: missing operator
$ rpnc 2 0 /
rpnc: arithmetic error
```

To enable unit testing of the calculator, implement the API defined in the following `rpn.h` header file shown at the end of this sheet.

b) Implement unit tests (ideally before writing the implementation of the calculator).

c) Determine the coverage of your unit test collection.

[1] https://libcheck.github.io/check/

```c
/*
 * rpn-stack.h --
 */

/** @file */

#ifndef _RPN_STACK_H
#define _RPN_STACK_H

/*
 * Prefixing all global symbols with rpn_ to avoid clashes with
 * symbols defined in the C library (e.g., signal stack definitions)
 * since we use this stack for building an RPN calculator.
 */

typedef struct _rpn_stack rpn_stack_t;

/**
 * \brief Create a new empty stack.
 *
 * \return A pointer acting as a handle for the new stack.
 */

rpn_stack_t*
rpn_stack_new();

/**
 * \brief Push data (allocated by the caller) on the stack.
 *
 * \param s The stack to push data on.
 * \param data The pointer to data to be pushed on the stack.
 */

void
rpn_stack_push(rpn_stack_t *s, void *data);

/**
 * \brief Pop data from the top of the stack.
 *
 * \param s The stack to pop data from.
 * \return The data or NULL if the stack is empty.
 */

void*
rpn_stack_pop(rpn_stack_t *s);

/**
 * \brief Peek on the data at the top of the stack.
 *
 * \param s The stack to peek on.
 * \return The data or NULL if the stack is empty.
 */

void*
rpn_stack_peek(rpn_stack_t *s);

/**
 * \brief Test whether a stack is empty.
 *
 * \param s The stack to test.
 * \return A non-zero value if the stack is empty, 0 otherwise.
 */
```

```
int
rpn_stack_empty(rpn_stack_t *s);

/**
 * \brief Delete a stack.
 *
 * \param s The stack to delete.
 */

void
rpn_stack_del(rpn_stack_t *s);

#endif
```

```c
/*
 * rpn.h --
 */

/** @file */

#ifndef _RPN_H
#define _RPN_H

/** RPN evaluation was successful. */
#define RPN_OK                  0
/** RPN evaluation failed due to an invalid token. */
#define RPN_INVALID_TOKEN       -1
/** RPN evaluation failed due to a missing operand. */
#define RPN_MISSING_OPERAND     -2
/** RPN evaluation failed due to a missing operator. */
#define RPN_MISSING_OPERATOR    -3
/** RPN evaluation failed due to an arithmetic error. */
#define RPN_ARITHMETIC_ERROR    -4

/**
 * \brief Evaluate an expression in reverse polish notation.
 *
 * This functions takes an array of strings that are interpreted as
 * tokens of an expression in reverse polish notation. The function
 * returns the results as a string via the second argument. The result
 * is allocated using malloc() and must be freed by the caller of this
 * function. Note that the function may return a NULL pointer if no
 * result was calculated. The return value of the function indicates
 * whether the evaluation of the expression in reverse polish notation
 * was successful or there were any errors.
 *
 * \param token Array of tokens making up the expression.
 * \param result Pointer to the string which will hold the result (malloced).
 * \result One of the error codes defined above.
 */

int rpn_eval_token(char *token[], char **result);

/**
 * \brief Evaluate an expression in reverse polish notation.
 *
 * This function splits the expression contained in the string expr
 * into an array of strings and then interpretes the array as tokens
 * of an expression in reverse polish notation. The function returns
 * the results as a string via the second argument. The result is
 * allocated using malloc() and must be freed by the caller of this
 * function. Note that the function may return a NULL pointer if no
 * result was calculated. The return value of the function indicates
 * whether the evaluation of the expression in reverse polish notation
 * was successful or there were any errors.
 *
 * \param expr The expression (whitespace separated numbers and operators)
 * \param result Pointer to the string which will hold the result (malloced).
 * \result One of the error codes defined above.
 */

int rpn_eval_expr(const char *expr, char **result);

#endif
```