EEE2020-03 **Spring 2021**

Assignment 4: Stack

Due: Sunday, May 16, 2021, 11:59PM

1 Introduction

- The objective of this assignment is to implement a postfix calculator using stacks.
- You will have to complete the missing parts of `class stack_t` in `stack.cc` and two missing functions of postfix calculator, `convert_infix_to_postfix()` and `calculate_postfix()`, in `solver.cc`.
- In Ubuntu or Mac terminal, use `wget` to download a copy of skeleton code compressed into `stack.tar`.

```
$ wget https://icsl.yonsei.ac.kr/wp-content/uploads/stack.tar
```

- Decompress the `tar` file, and go to the `stack/` directory. Then, type `ls` to find the following list of files.

```
$ tar xf stack.tar
$ cd stack/
$ ls
buffer.h  main.cc   solver.cc  stack.cc  tar.sh
data.h    Makefile  solver.h   stack.h   test.sh
```

- From the listed files, you have to work on two files, `stack.cc` and `solver.cc`
- You are allowed to change other files for your own testing and validation, but they will revert to the original state when your assignment is graded.
  - `data.h` defines the data type as *string* such that `typedef std::string data_t`.
  - `stack.*` files define `class stack_t`.
  - `buffer.h` defines `class buffer_t`. The buffer class is made on top of the stack class.
  - `main.cc` has the `main()` function.
  - `solver.*` files define `class solver_t` that parses a string expression, converts the infix expression to postfix, and calculates the postfix expression.
  - `test.sh` is a script file that tests the postfix calculator with six different expressions.
- To compile the code, you can simply type `make` in the terminal. `Makefile` that comes along with the skeleton code has automated all the compiling scripts for you.

```
$ make
g++ -Wall -Werror -g -o main.o   -c main.cc
g++ -Wall -Werror -g -o solver.o -c solver.cc
g++ -Wall -Werror -g -o stack.o  -c stack.cc
g++ -o postfix main.o solver.o stack.o
```

- Executing the skeleton code should print the following output.

```
$ ./postfix
Usage: ./postfix 'expression'

$ ./postfix '1 + 2 * 3'
Error: empty infix buffer: 1 + 2 * 3
```

2 Implementation

- `stack.h` defines `class stack_t` based on the lecture slides of `4_stack_queue.pdf`.
- The class definition is enclosed in the namespace of `ds` to avoid naming conflicts with standard libraries.
- `using namespace std` in the code can potentially cause compile errors, so you have to explicitly specify `std::` such as `std::cout`, `std::endl`, and `std::string`.
- Similarly, defining a `stack_t` instance in `main()` needs to specify the namespace such as `ds::stack_t st`.

```
/* stack.h */

#ifndef __STACK_H__
#define __STACK_H__

#include "data.h"

namespace ds {

class stack_t {
public:
    stack_t(void);          // Constructor
    virtual ~stack_t(void); // Destructor

    // Get the number of elements in the stack.
    size_t size(void) const;
    // Return true if the stack has no elements.
    bool empty(void) const;
    // Get a reference of the top element.
    data_t& top(void) const;
    // Push back a new element to the stack.
    void push(const data_t &m_data);
    // Pop back the last element of stack.
    void pop(void);

protected:
    data_t *array;          // Pointer to an array used as a stack
    size_t array_size;      // Allocated array size
    size_t num_elements;    // Number of elements in the array
};

}; // End of namespace ds

#endif
```

- The next shows `stack.cc` that has the stack implementation.
- In the skeleton code, class constructor and destructor are already implemented. The constructor allocates an 1024-entry array, and this should be more than enough to use for this assignment. The destructor destructs all elements in the stack and deallocates the array space.
- A few simple functions such as `size()`, `empty()`, and `top()` are also provided.
- The two missing functions of `class stack_t` are `push()` and `pop()`. Fill in the functions to make a complete stack ADT.
- In `push()`, you do not have to worry about a situation that the array becomes full, since the constructor allocates a large enough space for the array. No error checking or array resizing is necessary in this assignment.

```
/* stack.cc */

#include <cstdlib>
#include <iostream>
#include <new>
#include "stack.h"

namespace ds {

#define STACK_SIZE 1024

// Constructor
stack_t::stack_t(void) :
    array(0),
    array_size(STACK_SIZE),
    num_elements(0) {
    // Reserve a memory space.
    array = (data_t*)malloc(sizeof(data_t) * array_size);
}

// Destructor
stack_t::~stack_t() {
    // Destruct all data, and deallocate the array.
    for(size_t i = 0; i < num_elements; i++) { array[i].~data_t(); }
    free(array);
}

// Get the number of elements in the stack.
size_t stack_t::size(void) const { return num_elements; }

// Return true if the stack has no elements.
bool stack_t::empty(void) const { return !num_elements; }

// Get a reference of the top element.
data_t& stack_t::top(void) const { return array[num_elements-1]; }



/*************************
 * EEE2020: Assignment 4 *
 *************************/

// Push back a new element to the stack.
void stack_t::push(const data_t &m_data) {
    /* Assignment */
}

// Pop back the last element of stack.
void stack_t::pop(void) {
    /* Assignment */
}

/*********************
 * End of Assignment *
 *********************/

}; // End of namespace ds
```

- The next code box shows the contents of `buffer.h` that defines `class buffer_t`.
- The buffer class is made based on the stack using the *class inheritance* feature of C++.
  - `class buffer_t : public stack_t` tells that `class buffer_t` is created based on `stack_t`. The buffer class inherits all the variables and functions of stack class.
  - As you notice, the class has only one line of code that defines `operator[]`. In short, `buffer_t` is the same as `stack_t` but with an added functionality of `operator[]` that allows accesses to any elements in the array by index.
  - Private members of `stack_t` are defined in the `protected` region, not `private`. The `protected` region allows a *derived class* (i.e., `buffer_t`) to access the `protected` members of *base class* (i.e., `stack_t`), but other unrelated classes or functions cannot access them.
  - The destructor of `stack_t` is defined with the `virtual` keyword. *Virtual functions* are used for overriding the functions of base class by those of a derived class. In this particular case of virtual destructor, the `virtual` keyword ensures the invocation of `buffer_t`'s destructor function. Although forgetting to call the destructor of derived class would be benign in this code, it is potentially subject to memory leaks.
  - Understanding the class inheritance of C++ requires far more explanations with diverse case examples. This assignment does not dive deeper into the related discussions but simply gives you a complete implementation of class inheritance as a part of the skeleton code. Once you complete the missing functions of `stack_t`, the buffer class will work as well.

```
/* buffer.h */

#ifndef __BUFFER_H__
#define __BUFFER_H__

#include "stack.h"

namespace ds {

// buffer_t is a derived class of stack_t.
class buffer_t : public stack_t {
public:
    // Get the reference of element at the specified index.
    data_t& operator[](const size_t m_index) const { return array[m_index]; }
};

}; // End of namespace ds

#endif
```

- The next shows the header file of postfix solver defined as `class solver_t`.
- The class has only one `public` function named `solve()` that takes an infix expression in string format (i.e., class of character array). Other functions and variables are all `private`.
- `std::string` is simply a vector of `char` data type with more operators and functions to easily manipulate the character array.
- Within the `solve()` function, it calls `parse()` to split the string expression into independent terms of numbers, operators, and parentheses. For instance, a string expression of `"1 + 2 * 3"` is split into five terms, `1`, `+`, `2`, `*`, and `3`. These terms are stored in the infix buffer (i.e., `infix_buffer`). This part of implementation is already included in the skeleton code, so you do not have to worry about handling the strings.
- The infix buffer as a `buffer_t` type is made out of `stack_t`, and the stack class takes `data_t` as array elements. Since `data_t` is defined as `std::string`, the infix buffer is thus a string buffer. It means that expression terms such as `1` and `+` are all stored as strings. Even the numbers are stored as strings.

- The `solver_t` class has two buffers and two stacks, infix and postfix buffers, operator and number stacks. Their usages are well described in the lecture slides of `4_stack_queue.pdf`.
- The `convert_infix_to_postfix()` function reorders the expression terms stored in the infix buffer using the operator stack (i.e., `operator_stack`), and the reordered terms are placed in the postfix buffer (i.e., `postfix_buffer`). This implementation is going to be your homework.
- The expression terms in the postfix buffer are processed through `calculate_postfix()`. The number stack (i.e., `number_stack`) at the end of calculation will have only one number left in the stack, the final result.
- `solve()` returns the final result as `return number_stack.top()`. Since everything is stored as string in buffers and stacks, the final result is also returned as a string.
- Above the class definition of `solver_t`, there are a number of `#define ...` lines. The first three lines define acceptable characters of numbers, operators, and parentheses.
- The rest are *function macros* that facilitate the use of string terms. For instance, `is_number()`, `is_operator()`, or `is_paren()` tells if a string is a number, operator, or parenthesis.
- Using `string_to_float()` and `float_to_string()`, you can change a string to float, and vice versa.

```
/* solver.h */

#ifndef __SOLVER_H__
#define __SOLVER_H__

#include <string>
#include "buffer.h"
#include "stack.h"

namespace ds {

#define numbers   std::string(".0123456789")
#define operators std::string("+-*/")
#define parens    std::string("()")

#define is_positive_number(s) \
    (s.find_first_not_of(numbers) == std::string::npos)
#define is_negative_number(s) \
    ((s.size() > 1) && (s[0] == '-') && \
    (s.find_first_not_of(numbers, 1) == std::string::npos))
#define is_number(s) \
    (is_positive_number(s) || is_negative_number(s))
#define is_float(s) \
    (s.find_first_not_of(".") != std::string::npos)
#define is_operator(s) \
    (operators.find(s) != std::string::npos)
#define is_paren(s) \
    (parens.find(s) != std::string::npos)
#define string_to_float(s) \
    std::stof(s)
#define float_to_string(f) \
    std::to_string(f)

class solver_t {
public:
    // Solve the expression.
    std::string solve(const std::string &expression);

private:
    // Convert the infix expression to postfix.
    void convert_infix_to_postfix(void);
    // Calculate the postfix expression.
    void calculate_postfix(void);
    // Parse the string expression.
    void parse(const std::string &expression);

    buffer_t infix_buffer;      // Infix buffer
    buffer_t postfix_buffer;    // Postfix buffer
    stack_t  operator_stack;    // Operator stack
    stack_t  number_stack;      // Number stack
};

}; // End of namespace ds

#endif
```

- The next code box shows the contents of `solver.cc`.
- The two missing functions of `class solver_t` that you have to complete are `convert_infix_to_postfix()` and `calculate_postfix()`.
- The `solve()` and `parse()` functions are already implemented and given to you.
- In the `solve()` function, it calls `parse()` to split the string expression and prints the contents of infix buffer to validate if the parsing has been done well. Once you complete the `push()` and `pop()` functions of `class stack_t`, the infix buffer will be filled well.
- Then, `solve()` calls `convert_infix_to_postfix()` and prints the contents of postfix buffer to valid if expression terms have been reordered correctly.
- Lastly, it calls `calculate_postfix()` and returns the final result located at the top of number stack.

```
/* solver.cc */

#include <cstring>
#include <iostream>
#include "solver.h"

namespace ds {

/************************
 * EEE2020: Assignment 4 *
 ************************/

// Convert infix expression to postfix.
void solver_t::convert_infix_to_postfix(void) {
    /* Assignment */
}

// Calculate the postfix expression.
void solver_t::calculate_postfix(void) {
    /* Assignment */
}

/********************
 * End of Assignment *
 ********************/

```

```

// Solve the expression.
std::string solver_t::solve(const std::string &expression) {
    // Parse the string expression.
    parse(expression);

    // Show the infix expression.
    std::cout << "infix expression:";
    for(size_t i = 0; i < infix_buffer.size(); i++) {
        std::cout << " " << infix_buffer[i];
    }   std::cout << std::endl;

    // Convert the infix expression to postfix.
    convert_infix_to_postfix();

    // Show the postfix expression.
    std::cout << "postfix expression:";
    for(size_t i = 0; i < postfix_buffer.size(); i++) {
        std::cout << " " << postfix_buffer[i];
    }   std::cout << std::endl;

    // Calculate the postfix expression.
    calculate_postfix();

    // Return the final result left in the operator stack.
    return number_stack.top();
}

// Parse the string expression.
void solver_t::parse(const std::string &expression) {
    try {
        // Check if the expression has illegal characters.
        size_t e = expression.find_first_not_of(numbers+operators+parens+" ");
        if(e != std::string::npos) { throw "invalid characters"; }

    ...
}

}; // End of namespace ds
```

- The following gives you some hints regarding the implementation of `convert_infix_to_postfix()` and `calculate_postfix()` functions.
- The `convert_infix_to_postfix()` function may look like the following.

```
// Convert infix expression to postfix.
void solver_t::convert_infix_to_postfix(void) {
    for(size_t i = 0; i < infix_buffer.size(); i++) {
        std::string &term = infix_buffer[i];
        if(is_number(term)) { postfix_buffer.push(term); }
        else if(term == "(") { operator_stack.push(term); }
        else if ...

    }

    ...
}
```

- And the `calculate_postfix()` function can be implemented as follows.

```
// Calculate the postfix expression.
void solver_t::calculate_postfix(void) {
    for(size_t i = 0; i < postfix_buffer.size(); i++) {
        std::string &term = postfix_buffer[i];
        if(is_number(term)) { number_stack.push(term); }
        else {
            float operand2 = string_to_float(number_stack.top());
            number_stack.pop();
            float operand1 = string_to_float(number_stack.top());
            number_stack.pop();
            float result;

            if (term == "+") { result = operand1 + operand2; }

            ...

            number_stack.push(float_to_string(result));
        }
    }
}
```

- The following shows the `main()` function.
- A comment block in the middle of function shows you an example to validate the operations of `stack_t`. Use and modify this example as necessary to test if your stack implementation is correctly written before you proceed to the postfix calculator.

```
/* main.cc */

#include <iostream>
#include "solver.h"

int main(int argc, char **argv) {
    // Run command message
    if(argc != 2) {
        std::cerr << "Usage: " << argv[0] << " 'expression'" << std::endl;
        exit(1);
    }


    /*
    // Test code to validate stack implementation
    ds::stack_t st;
    st.push("1");
    st.push("+");
    st.push("2");
    std::cout << st.top()  << std::endl;
    std::cout << st.size() << std::endl;
    st.pop();
    std::cout << st.top()  << std::endl;
    std::cout << st.size() << std::endl;
    st.pop();
    std::cout << st.top()  << std::endl;
    std::cout << st.size() << std::endl;
    */


    // Solve the expression.
    ds::solver_t solver;
    std::string result = solver.solve(argv[1]);
    std::cout << "result = " << std::stof(result) << std::endl;

    return 0;
}
```

- If `stack_t` and `solver_t` are all done, you can test the program as follows.

```
$ ./postfix '1 + 2 * 3'
infix expression: 1 + 2 * 3
postfix expression: 1 2 3 * +
result = 7

$ ./postfix '(1 + 2) * 3'
infix expression: ( 1 + 2 ) * 3
postfix expression: 1 2 + 3 *
result = 9
```

- Your code will be graded for infix expressions defined in the `test.sh` script file. Executing the script, you should get the following output.

```
$ ./test.sh
Test #1:
infix expression: 51 - 3 * 8 + 20
postfix expression: 51 3 8 * - 20 +
result = 47

Test #2:
infix expression: 3 - 4.2 * 7 - 10 + 5 / -2 + 12.3 * 3
postfix expression: 3 4.2 7 * - 10 - 5 -2 / + 12.3 3 * +
result = -2

Test #3:
infix expression: -7 / ( -3 + 1 ) + -2 * 9 - ( 0.5 + 3 ) * 5
postfix expression: -7 -3 1 + / -2 9 * + 0.5 3 + 5 * -
result = -32

Test #4:
infix expression: 17 - 5 * ( -3 - 1.5 * 4 ) / ( 2 + 2.5 / 5 ) - 5 * 3
postfix expression: 17 5 -3 1.5 4 * - * 2 2.5 5 / + / - 5 3 * -
result = 20

Test #5:
infix expression: ( 15.7 - ( 3 - 2 ) * 5 / 2 ) + 5 * 2.2 / ( 5.1 - 2.6 ) * 2 - (
                  -5 * 3.1 )
postfix expression: 15.7 3 2 - 5 * 2 / - 5 2.2 * 5.1 2.6 - / 2 * + -5 3.1 * -
result = 37.5

Test #6:
infix expression: ( ( ( 11 / 2 ) - 5 ) * ( 2 + ( ( 4 - 1.5 ) / -2.5 * 5 ) * -1 )
                  ) - ( 5.25 - 2.5 * 2.5 )
postfix expression: 11 2 / 5 - 2 4 1.5 - -2.5 / 5 * -1 * + * 5.25 2.5 2.5 * - -
result = 4.5
```

3 Submission

- When the assignment is done, execute the `tar.sh` script in the `stack/` directory.
- It will compress the `stack/` directory into a `tar` file named after your student ID such as `2020140000.tar`.

```
$ ./tar.sh
rm -f main.o solver.o stack.o postfix
a stack
a stack/main.cc
a stack/data.h
a stack/Makefile
a stack/solver.h
a stack/stack.cc
a stack/buffer.h
a stack/stack.h
a stack/tar.sh
a stack/test.sh
a stack/solver.cc

$ ls
2020140000.tar  data.h   Makefile   solver.h  stack.h  test.sh
buffer.h        main.cc  solver.cc  stack.cc  tar.sh
```

- Upload the tar file (e.g., `2020140000.tar`) on LearnUs. Do not rename the `tar` file or C++ files included in it.

4 Grading Rules

- The following is the general guideline for grading. 30-point scale will be used for this assignment. The minimum score is zero, and negative scores will not be given. Grading rules are subject to change, and the grader may add a few extra rules for fair evaluation of students' efforts.

  -3 points: A submitted `tar` file is renamed and includes redundant tags such as `hw4`, student name, etc.

  -5 points: A program code does not have sufficient amount of comments. Comments in the skeleton code do not count. You must make an effort to clearly explain what each part of your code intends to do.

  -5 points each: There are total six sets of testing performed by `test.sh`. For each incorrect block, 5 points will be deducted.

  -25 points: The code does not compile or fails to run (e.g., no or completely wrong results), but it shows substantial efforts to complete the assignment.

  -30 points: No or late submission. The following cases will also be regarded as no submissions.

  $*$ Little to no efforts in the code (e.g., submitting nearly the same version of code as the skeleton code) will be regarded as no submission. Even if the code is incomplete, you must make substantial amount of efforts to earn partial credits.

  $*$ Fake codes will be regarded as cheating attempts and thus not graded. Examples of fake codes are i) hard-coding a program to print expected outputs to deceive the grader as if the program is correctly running, ii) copying and pasting random stuff found on the Internet to make the code look as if some efforts are made. More serious penalties may be considered if students abuse the grading rules.

  Final grade = F: The submitted code is copied from someone else. All students involved in the incident will be penalized and given F for the final grades irrespective of assignments, attendance, etc.

- Your teaching assistant (TA) will grade your assignments. If you think your assignment score is incorrect for any reasons, feel free to discuss your concerns with the TA. In case no agreement is made between you and the TA, elevate the case to the instructor to review your assignment. Refer to the course website for the contact information of TA and instructor: `https://icsl.yonsei.ac.kr/eee2020`