Homework #5

Released: 02-07-2017
Due: 02-14-2017 11:59pm

This is the first homework in the homework series of our project to build a tiny network simulator. In this homework, you are required to implement three utility functions and write comprehensive unit tests. The three functions are: `parse_int` and `tokenize` in `parsing.cpp`, and `parse_IP` in `datagram.cpp`. We will talk about the specification of these functions in Section 2. Hints on how to implement these functions are given in Section 5. Unit tests will be written in `networksim_test.cpp`

In addition to what we have seen in the class, we will also be introducing two more C++ language features (or standard library classes) in this homework: scoped enum "`enum class`" and fixed-sized arrays "`std::array`".

Scoped Enum, `enum class`

Scoped `enum` gives one the ability to defined groups of uninterpreted symbols in the code. There are two groups of `enum class`es in the provided code, the first one models the commands of our simulator and the second one models the error codes in our system.

```
// interface.h
enum class cmd_code
{
    halt,
    system_status,
    // ...
    undefined,
};
```

```
// errors.h
enum class err_code
{
    cmd_undefined,
    syntax_error,
    bad_ip_address,
    // ...
};
```

The syntax for referring to symbols in an `enum class` group is "*GroupName*::*Symbol*". For example, we can refer to `cmd_codes` by `cmd_code::halt`, `cmd_code::system_status`, `cmd_code::undefined` and refer to error codes by `err_code::cmd_undefined`, `err_code::syntax_error` and `err_code::bad_ip_address`.

These symbols are normal values in a C++ code. We can store them in variables, pass them to functions or even use them to throw exceptions as in the following example. While these symbols can be easily compared for equality with the symbols in the same group, we need additional work even just to make them printable. Throughout the project, we will only compare the symbols for equality.

The rationale behind this language construct is to provide a way to model enumerated items. With `enum class`es, we can avoid using plain integers to represent enumerated items and avoid name conflicts.

```
void err(err_code e)
{
    if (e == err_code::cmd_undefined)
        cout << "undefined command\n";
}

err_code e = err_code::syntax_error;
err(e);
throw err_code::bad_ip_address;
```

Fixed-Size Array, `std::array`

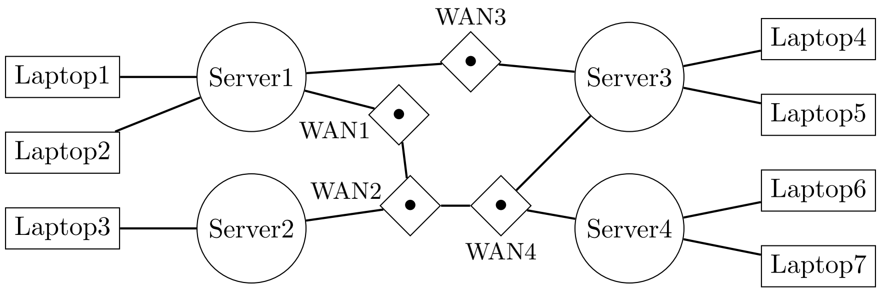
In `datagram.h`, `IP_address` is temporarily defined as an array of four `int`s. `std::array` is a C++ standard class defined in the `array` header that let one create a fixed-size array on the stack.

```
// datagram.h
#include <array>
using IP_address = std::array<int, 4>;
```

An `std::array<T, N>` is a type of an array storing `N` objects of type `T`. Take `std::array<string, 5>` as an example, it is the type for an array of 5 strings. Similar to `std::vector`, we can use the `_[_]` operator to access the elements of an array and use the `_.size()` member function to obtain its length. Since the length of an array is fixed, there is no `_.push_back(_)` member function.

```
std::array<int, 4> ip;
ip[0] = 192; ip[1] = 168; ip[2] = 0; ip[3] = 1;
cout << ip[1] - ip[0] << ' ' << ip.size() << '\n';
```

1 Project Introduction: Homework 5-8



In this project, we are going to build a tiny network simulator modeling a small system that has laptops, servers and WAN (Wide Area Network) nodes. We will also model datagram transmission between them. A laptop must first be connected to a server. A server can connect multiple laptops, building a LAN (Local Area Network) between them. A server can also be connected to multiple WANs, in which case it will be able to transfer datagrams indirectly to other servers and finally to other laptops outside LAN. A WAN node can connect not only to arbitrary servers, but also to other WAN nodes.

Starting from homework 6, we will implement one class for each of the constructs in this system: a `System` class for the entire network system, a `Datagram` class for datagrams and machine classes `Laptop`, `Server`, `WAN_node` for laptops, servers, and WAN nodes respectively. The `System` class will have member functions corresponding to network operations. These include: sending and receiving a datagram on a `Laptop`, adding and removing machines from the network, and a time ticking function for servers and WAN nodes to route datagrams one step toward their destination.

The simulator, aside from the `System` class modeling the entire network, also contains a command line interface to interact with the user. The user can enter commands to control the system and view the status of the network system. In this homework, we are going to implement three utility parsing functions that help the command line interface convert input strings into commands and accompanying data in order to invoke the corresponding member functions of the `System` class.

In provided the code, `main.cpp` and `interface.cpp` implement the command line interface. In `main.cpp`, the `main` function repeatedly reads a line from the user, parses the input into tokens by the `tokenize` function in Section 2, and calls `execute_command` to perform the corresponding operations. If an error is thrown, it catches the error code `err_code` and prints an error message.

In `interface.cpp`, the `execute_command` function first identifies the input command by searching through the `command_syntaxes` list, match the command string and obtain the `cmd_code` for the input command. `execute_command` then parses the accompanying data (some by `parse_IP` in this homework) and invokes the member function of `System`. However, in this homework, the `execute_command` function merely calls made up functions that print a message. The actual `System` definition, `execute_command`, as well as user command explanation, will be given in later homework.

2 Parsing Strings

2.1 Tokenizing Strings

Implement the function `std::vector<std::string> tokenize(const std::string& line);` that tokenizes the input string while recognizing double quotes “`"`”. In our case, tokenizing a string `line` means sequentially grouping characters other than spaces and double quotes in `line` as multiple substrings. Recognizing double quotes means treating the characters between two enclosing ‘`"`’s as being in the same substring. Here are some examples[1]:

```
// The tokenization of ‘   de@f.com  "ghi j-k " w == "z"’ is, in C++ syntax,
// tokens == {"de@f.com", "ghi j-k ", "w", "==", "z"}
vector<string> tokens = tokenize("   de@f.com  \"ghi j-k \" w == \"z\"");

// The tokenization of ‘hel"l"o world "" "eecs   211"’ is, in C++ syntax,
// tokens == {"hel", "l", "o", "world", "", "eecs   211"}
vector<string> tokens = tokenize("hel\"l\"o world \"\" \"eecs   211\"");

// The tokenization of ‘"abc\""def"’ is, in C++ syntax,
// tokens == {"abc\\", "def"}
vector<string> tokens = tokenize("\"abc\\\"\"def\"");
```

In the first example, the first group of consecutive characters that are neither spaces nor double quotes are “`de@f.com`”. The second group are the characters “`ghi j-k `”, enclosed by two double quotes. Continuing this mannar, the three following groups are “`w`”, “`==`” and “`z`”. Similar to the second group, “`z`” is enclosed by two double quotes.

In the second example, the first group of characters are “`hel`”, because the double quote following the first `l` ends the group. But the double quote also starts the second group, which is the second `l`, enclosed by two double quotes. The third group “`o`” then follows and stops after `o` as there is a space. The fifth group is an empty string enclosed by a pair of double quotes.

In the third example, two pairs of matching double quotes “`"abc\"`” and “`"def"`” enclose two groups of consecutive characters that are not double quotes. Thus the first group of characters are “`abc\`” and the second group of characters are “`def`”. Note that there could be no spaces between these two pairs of double quotes. Even though the character sequence `\"` is treated specially in C++ string literals, it does not have any special meaning in our `tokenize` function.

2.2 Parsing Integers

Implement the function `int parse_int(const std::string& s);` that parses the entire string `s` into an `int` without using any library functions like string streams, `atoi`, `stoi`, etc. For example, `parse_int("29") == 29` and `parse_int("00456000") == 456000`.

A valid `s` which represents a number should be non-empty and consists only of characters from `'0'` to `'9'`. There can be no spaces, no new lines, no letters and no symbols. In particular, this also implies that our `parse_int` function will not deal with negative numbers.

[1] “`\"`” and “`\\`” are how we insert double quotes and back slashes in C++ string literals.

2.3 Parsing IP Addresses

Implement the function `IP_address parse_IP(std::string s);` that parses the entire string `s` into an `IP_address`. As seen in the introduction, the `IP_address` type is temporarily defined to be the synonym of four-element `int` arrays `std::array<int, 4>`. In addition, the values of all four `int`s in a valid IP address must be between 0 and 255 (inclusive).

A valid IP address, when represented as a string `s`, should be four numbers separated by exactly three dots, "$d_0.d_1.d_2.d_3$". The four numbers $d_0, \ldots, d_3$ must represent non-negative numbers in the sense of Section 2.2. Also, there can be no spaces, no new lines, no letters and no symbols except those three dots.

Here are some examples:

```
// In C++ syntax, ip == {192, 168, 0, 1}
IP_address ip = parse_IP("192.168.0.1");

// In C++ syntax, ip == {255, 255, 255, 0}
IP_address ip = parse_IP("255.255.255.0");

// In C++ syntax, ip == {0, 0, 0, 0}
IP_address ip = parse_IP("0.0.0.0");
```

3 Handling Errors

For error handling,

- `tokenize` should throw a `runtime_error` if the double quotes are not in pairs.
- `parse_int` should throw a `runtime_error` when the input string does not represent non-negative integers – those that do not fit into the string format specified in Section 2.2.
- `parse_IP` should throw an `err_code` exception with value `bad_ip_address` if the input is not a string that represents a valid IP address. In other words, when `parse_IP` is called with a string that does not fit into the string format specified in Section 2.3, it should "`throw err_code::bad_ip_address;`".[2]

4 Unit Testing

Implement comprehensive unit tests for the three functions specified in Section 2 in `networksim_test.cpp`. You have to figure out what cases there might be and implement a corresponding unit test to ensure that the required functions work properly. We will also grade on the completeness of unit test coverage in the form of self-evaluation as in Homework 4.

5 Hints

There are more than one possible solutions for all required functions in this homework. Below are some hints on how these functions can be implemented. Feel free to come up with other implementations so long as the behavior matches the specification.

The `_.substr(`*begin*`,` *length*`)` member function of the `string` class might be useful in some functions. When we call `s.substr(`a`,` l`)` on a `string s`, it will return a new string that equals

[2]Note that the string format includes both the numbers and dots, and the ranges of the numbers, which are limited.

to the content of `s` between a and $a + l - 1$. That is, in pseudo-syntax, `s.substr(`a`, `l`) == s[`$a..(a + l - 1)$`]`.

5.1 std::vector<std::string> tokenize(const std::string& line);

We can implement the `tokenize` function by a nested loop. The outside loop scans through `line`. Upon the loop seeing the begin of a group of characters (i.e. a non-space character or a double quote), the inner loop scans from the current position to find the end of the group.

If the beginning character of this group is a double quote, then the inner loop scans until it finds the ending double quote. Otherwise, the inner loop scans until it finds the first character outside this group, which can be either a space or a double quote. We then collect the group of characters found by the inner loop in a vector, and the outer loop continues from where the inner loop ends.

To sum up, the function looks like:

```
current_position = 0
while we have more to search
    if line[current_position] is a double quote
        the inner loop scans to find the ending double quote
        we have found a new group of characters
        set current_position to where the inner loop ends + 1
    else if line[current_position] is not a space
        the inner loop scans to find the ending position (a space or a double quote)
        we have found a new group of characters
        set current_position to where the inner loop ends
    else
        advance current_position by 1
```

5.2 int parse_int(const std::string& s);

Before implementing this function, please note that `'0'` and 0, `'1'` and 1, ..., `'9'` and 9 are all different values. The former ones are `char` literals and the latter ones are integer literals. Even though the former ones can be casted to integral types, it is still likely that they assume different values from the latter ones. For example, `'0'` assumes the value 48 and `'9'` assumes the value 57 on my machine (see: ASCII code).

To map the `char` literals to integer values, one simple way is to declare a `vector` storing the characters and look them up to obtain the integer value, like the following:

```
// '0' is stored in ords[0], ..., '9' is stored in ords[9]
const vector<char> ords{'0', '1', '2', '3', '4', '5', '6', '7', '8', '9'};
```

To come up with an conversion algorithm, let's first examine an example. To process the string `"8"`, we only need to look up the `ords vector`. To process the string `"214"`, we can first convert the string `"21"` to the number 21 and lookup the character `'4'` in `ords` to obtain its value, 4. Then, the numeric value of the entire string `"214"` will be $21 \times 10 + 4$.

Thus, implementing the following algorithm using recursion or a loop suffices: (1) If the string contains exactly one digit, then we simply lookup its value in `ords` (2) otherwise, given the string "$s_0 \dots s_{n-1}s_n$", we first convert "$s_0 \dots s_{n-1}$" into a number d_{n-1}. After looking up the number n of s_n in `ords`, we calculate the value of the entire string $d_n = 10d_{n-1} + n$.

5.3 IP_address parse_IP(std::string s);

We can use the `tokenize` function and the `parse_int` function that we have just written. Given a string `"192.168.0.1"`, we first write a loop to replace the dots by spaces to obtain `"192 168 0`

`1"`. Then `tokenize` gives back a vector of four strings `{"192", "168", "0", "1"}`. Now we can apply `parse_int` to obtain the final result `{192, 168, 0, 1}`.

Beware that this approach would have allowed additional invalid IP addresses to pass through the parser such as `" 192 .  168 0.\"1\""`. We need to check, then, that there are no characters other than digits and dots, before replacing the dots with spaces.