ECE26400 Programming Exercise #2

In this exercise, you will implement two functions:

1. To convert from a `char` to `int` (`char_to_int`).
2. To process a string and return a `long int` represented in the string (`str_to_long_int`).

The main learning goals are:

1. How to "iterate" over a string, which is an array of characters terminated with the NULL-character `'\0'`.
2. How to convert a string into a number in different bases.
3. How to set an error number when the base is invalid, or when there is an overflow or underflow.
4. How to test your own implementation.

This exercise is related to PA01.

1 Getting started

You should unzip on eceprog.ecn.purdue.edu the zip file `pe02_files.zip` using the following command:

```
unzip pe02_files.zip
```

The zip file `pe02_files.zip` contains a folder, named `PE02`, and three files within the folder:

1. `answer02.c`: this is the file that you would hand in after modification. It has the skeleton of the functions in it. This is the only file you have to include in the zip file to be submitted through Brightspace.
2. `answer02.h`: this is a "header" file and it contains a declaration and a brief description of the functions you will be writing for this exercise. **You should not make changes to** `answer02.h`**.**
3. `pe02.c`: You should use this file to write testing code that runs the functions in `answer02.c`, in order to ensure their correctness.

To get started, read this document in its entirety. You will be writing code in the `answer02.c` file.

You will also write code in the `pe02.c` file to test the functions written in the `answer02.c` file. `pe02.c` **is meant for your testing of** `answer02.c`**. It should not be submitted.**

2 Functions you have to write

For the purpose of this exercise, a string representing a number should be of the following form:

"possibly some white space" "optional +/- sign" "combination of 0..9, a..z, A..Z"

White space can be space (`' '`), horizontal tab (`'\t'`), carriage return (`'\r'`), form feed (`'\f'`), newline (`'\n'`), and vertical tab (`'\v'`). The function `isspace` from `ctype.h` may be useful here.

Here `'a'..'z'`, `'A'..'Z'` are considered legal symbol because we allow up to a base of 36. We will consider a base from 2 through 36. We do not consider a base of 1, because the only valid symbol will be `'0'`, which is quite meaningless. The functions `isalnum`, `isalpha`, `isdigit`, `islower`, and `isupper` from `ctype.h` may be useful here.

For a base of 2, the valid symbols are `'0'` and `'1'`. For a base of 3, the valid symbols are `'0'`, `'1'`, and `'2'`. In general, for a base of 2 through 10, the valid symbols are `'0'`, `'1'`, ... to the character corresponding to (base -1).

We also include the characters `'a'` through `'z'` when we consider a base from 11 through 36. `'a'` stands for 10 (in base 10), `'b'` for 11, ..., `'z'` for 35.

Again, for a base from 11 through 36, the valid symbols are `'0'`, `'1'`, ..., `'9'`, `'a'`, ... to the character corresponding to (base -1).

Note that your functions should accept both upper case and lower case letters.

The presence of a NULL-character `'\0'` means that you have reached the end of a string.

`ctype.h` has been included in `answer02.c`. Even though some functions from `ctype.h` are mentioned here, it is up to you whether you want to use any of them for this exercise.

2.1 `char_to_int` function

This function should receive a character and returns its numeric value. If the character is `'0'`, ..., or '9', it should return 0, ..., or 9. If the character is `'a'`, ..., or 'z', it should return 10, ..., or 35. If the character is `'A'`, ..., or 'Z', it should return 10, ..., or 35. For all other input characters, it should return the macro `INV_SYMBOL`, which is defined in `answer02.h`.

There is a straightforward way to write a function that returns one of the 37 values (`INV_SYMBOL`, 0, ..., 35) based on the input character using around thirty seven if-else statements or switch-case statements. You should think of a way that uses around three or four if-else statements. You may want to ask yourself what functions in `ctype.h` tell you that the input character is a valid symbol.

2.2 `str_to_long_int` function

You may assume that the input address supplied to the `str_to_long_int` function is not NULL (as indicated in `answer02.h` by `__attribute__((nonnull (1)))`. In our evaluation of your submission, we will not call this function with a NULL address. You may assume that location at which the address points contains a string.

Note that the type of address passed into the function is pointing at `char const`, which means that the function `str_to_long_int` is not allowed to modify the string. When we evaluate your function, we will check whether your implementation make modification to the string.

As there could be leading white space characters in the given string, you have to skip over characters that are considered to be white space before the conversion. The function `isspace` from `ctype.h` may be useful. You may find other functions from `ctype.h` useful.

The number represented in the string may start with a negative sign `'-'` or a positive sign `'+'`.

Immediately following the optional `+/-` sign should be the legal symbols for a particular base.

Your conversion of a string to a long integer should stop whenever you encounter an invalid symbol for that base. For example, for the string `"  +243adg"`, the following sub-strings are considered valid until the next character in the string is encountered for the respective bases:

1. For base 2, the substring `"  +"` is valid; the next character, `'2'`, in the given string is not a valid symbol.
2. For base 4, the substring `"  +2"` is valid; the next character, `'4'`, in the given string is not a valid symbol.
3. For base 11, the substring `"  +243a"` is valid; the next character, `'d'`, is not a valid symbol.

For base 2, the function should return 0.

For base 4, the function should return 2.

For base 11, $2 \times 11^3 + 4 \times 11^2 + 3 \times 11 + 10 = 3189$. Therefore, the function should return 3189.

2.2.1 Conversion

Given a string of valid symbols, how should we convert the string to a number? Let's assume that the number will be stored in the variable `ret_value`, and you will return that value.

First, initialize the `ret_value` to 0. Assuming that are converting `"243a"` for base 11, we encounter `'2'` first. First, you have to convert a character `'2'` to a number 2. We will update `ret_value` to `ret_value * base + 2`, which gives us $0 \times 11 + 2 = 2$.

When we encounter the next symbol `'4'`, we convert `'4'` to a number 4, and update `ret_value` to `ret_value * base + 4`, which gives us $2 \times 11 + 4 = 26$.

Next, convert `'3'` to number 3, update `ret_value` to `ret_value * base + 3`, which gives us $26 \times 11 + 3 = 289$. Last, convert `'a'` to number 10, and update `ret_value` to `ret_value * base + 10`, which gives us $289 \times 11 + 10 = 3189$.

The above steps assume that the sign is positive. You have to make minor changes to the steps when the sign is negative.

Seeing that you are repeating the steps of converting a character to a number, and updating the `ret_value` using the same equation, you should write these steps as an iteration (while-loop, for-loop, for example). When should the iteration stop? Perhaps when you encounter `'\0'` or when you encounter an invalid symbol for that base.

As `ret_value` is initialized to 0 and the function returns `ret_value` when we encounter an invalid symbol, 0 should be returned if the base is invalid or the string to be converted is invalid.

2.3 Handling `errno` in `str_to_long_int`

If the base is invalid, the variable `errno` should be set to `EINVAL` (invalid argument) before you return from the `str_to_long_int` function. The variable `errno` is defined in `errno.h`, which has been included in `answer02.c`.

It is possible that you are given a string that is too big or too small to be stored in a `long int`. The largest long integer is defined by `LONG_MAX` and smallest long integer is defined by `LONG_MIN`, both of which are defined in `limits.h`. `limits.h` has been included in `answer02.c`.

If you encounter such a string, you have to set the variable `errno` to `ERANGE` before you return from `str_to_long_int`. The value returned by `str_to_long_int` should be `LONG_MAX` or `LONG_MIN`, accordingly.

2.3.1 Determining overflow or underflow

What happens when you add two very large (positive) numbers together or when you add two very small (negative) numbers together? For example you can try to add `LONG_MAX` and `LONG_MAX` or `LONG_MIN` and `LONG_MIN`. In the former, you should get a negative sum and in the latter, you should get a non-negative sum. In the former, an overflow occurs because the sum is too big to be represented using a `long`, and in the latter, an underflow occurs because the sum is too small to be represented using a `long`.

To determine whether you have a number greater than `LONG_MAX` or smaller than `LONG_MIN`, let us examine the operations that may cause an overflow/underflow to occur. Assume that we have `ret_value` that is valid, and the current character is converted to a valid numeric value `curr_value`. We have to update the `ret_value` to `ret_value * base + curr_value`.

There are two mathematical operations that could cause an overflow/underflow: multiplication and addition. First, consider two `long int` variables `a` and `b`, both of which are non-negative (when you are converting a string into a positive number, both `long int` variables involved are non-negative). If `a + b` $>$ `LONG_MAX`, you get an overflow. However, you cannot check for the condition "`a + b` $>$ `LONG_MAX`" because when that condition holds, overflow has already occurred, and the result of (`a+b`) will give you a number smaller than `LONG_MAX`. You should also write simple code to find out what happens when you add a positive number to `LONG_MAX` to understand the challenge here.

Therefore, to check for the condition that "`a + b` $>$ `LONG_MAX`", we typically check for the condition "`LONG_MAX - b` $<$ `a`" or "`LONG_MAX - a` $<$ `b`".

How about the multiplication of two non-negative `long int` variables `a` and `b`? Overflow occurs when `a * b` $>$ `LONG_MAX`. Again, you cannot perform multiplication and then check for overflow. You can use a similar idea as in the case of addition to determine whether overflow occurs. Note that overflow may occur only if both `a` and `b` are not 0.

You also have to determine whether underflow occurs during the conversion process when you are dealing with string that gives you a negative number. Here, we have to add or multiply two non-positive negative `long int` variables `a` and `b` together? You have to determine underflow in a similar fashion.

2.4 Printing, helper functions and macros

All debugging, error, or log statements in `answer02.c` should be printed to `stderr`. We do not expect any messages to be printed to `stdout`.

You may define your own helper functions in `pe02.c` and `answer02.c`. You may view those test functions provided in `pe02.c` as helper functions. All these functions are declared and defined as static functions. In other words, the scope of these functions are only within the files that they are found. Declaring and defining these functions as static eliminate the conflicts in function names. You should not modify any of the `.h` files.

A static helper function could skip over all white spaces when given a string.

~~\texttt{static char const *skip_white_spaces(char const *str)~~

```
static char const *skip_white_spaces(char const *str)
{
```

```
    ...
}
```

Here, the function returns the address of the non-white-space character.

A static helper function could determine whether the conversion should return a non-negative number ($+1$) or a negative number (-1).

~~\texttt{static char const *determine_sign(char const *str, int *sign_ptr)~~

```
static char const *determine_sign(char const *str, int *sign_ptr)
{
    ...
}
```

Note that, the returned address should point to a location that contains the most significant symbol, if it is valid, to be converted. The caller function should have an `int` variable `sign`, and the address of `sign` is passed to `determine_sign` as `sign_ptr`. In `determine_sign`, `*sign_ptr` should store $+1$ or -1 accordingly.

You may have other appropriate static helper functions.

You also should not use macros that start with `T_` (Upper case T and underscore) in their names. We will use such macros in the evaluation of your submissions. If you use such macros, we may not be able to evaluate your submission properly.

3 Compiling your program

We use the same flags introduced in PE01 to compile the program for debugging purpose.

```
gcc -std=c99 -Wall -Wshadow -Wvla -Werror -g -pedantic -fstack-protector-strong \
    --param ssp-buffer-size=1 pe02.c answer02.c -o pe02
```

When we evaluate your program, we also use the optimization flag `-O3` (uppercase letter `O` and number three) instead of the `-g` flag as follows:

```
gcc -std=c99 -Wall -Wshadow -Wvla -Werror -O3 -pedantic -fstack-protector-strong \
    --param ssp-buffer-size=1 pe02.c answer02.c -o pe02
```

You are recommended to copy the `Makefile` from PE01 and modify it appropriately for PE02.

4 Running and testing your program

To run your program, it depends on what you put in `pe02.c`. You should decide how you want to write your test functions and what input arguments your main function is expecting. Therefore, we cannot tell you what exactly you should type into the terminal. In its current form, you do not have to type in any input arguments. In general, it would look like this:

```
./pe02 "your input arguments"
```

Note that this should print something like:

```
Welcome to ECE264, we are working on PE02.

...
```

Some rudimentary test functions have been put in `pe02.c`. However, you should put in more tests to thoroughly test your program.

It is your responsibility to test the functions implemented in `answer02.c` and ensure that they work. How should you test your implementation?

C has a function that would do exactly what you have been asked to do in this exercise for the `str_to_long_int` function. You may have seen this function in the lecture notes, and that function is `strtol`.

However, you are not allowed to call this function in your function, as we do not allow you to include `stdlib.h` in `answer02.c`. If that function shows up in `answer02.c`, even if it is commented, you will receive ZERO for this exercise. Similarly, if `stdlib.h` shows up in `answer02.c`, even if it is commented, you will receive ZERO for this exercise.

However, you are allowed to use `strtol` in your test functions in `pe02.c` to verify that your functions work correctly. In the test function we provided in `pe02.c`, we use `strtol` as follows:

```
char *endptr;  // this is for strtol
int base;

...

// call strtol to cross check the results
// note that the results should not match when base == 16.
// before you call the function strtol
// set errno to zero

errno = 0;
test = strtol("     +0X044", &endptr, base);
printf("%ld :%s %d\n", test, endptr, errno);

...
```

Note that we set `errno` to 0 before calling `strtol`. We call the `strtol` function with the address to a string (passing `"     +0X044"` is equivalent to passing the address of the string `"     +0X044"`), the address of `endptr`, and the base. `strtol` returns the converted long integer. It also returns via the second parameter the address of the character that causes the conversion to stop. After that, we print the converted long integer, the remaining string that has not been converted (starting from the character pointed to by `endptr`), and `errno`. In a more useful program, you would check for the `errno` when the returned value is `LONG_MAX` or `LONG_MIN` to see whether there was an overflow or underflow.

In `pe02.c`, we also set `errno` to 0 before calling the `str_to_long_int` function.

However, note that if you try to convert `"0x10"` using `strtol` for base 16 or base 0, you will get 16. However, the `str_to_long_int` function should return 0 in both cases. Moreover, `errno` should be assigned `EINVAL` for base 0. To find out more, please do a "`man strtol`" at your terminal.

Moreover, you should check for memory errors/issues of your program using `valgrind`.

5 Submission

You must submit a zip file called `PE02.zip`, which contains one file:

1. `answer02.c`

Assuming that you are in the folder that contains `answer02.c`, use the following command to zip your files:

```
zip PE02.zip answer02.c
```

Make sure that you name the zip file as `PE02.zip`. Moreover, the zip file should not contain any folders. Submit `PE02.zip` through Brightspace. You may submit as many versions as you like. Brightspace will keep only the most recent submission, and we will grade only the final submission.

6 Grading

Your `answer02.c` should not include `stdlib.h` and it should not call `strtol`. If your submission has any occurrence of `stdlib.h` or `strtol`, your submission will receive a 0 grade, even if such an occurrence appears in a a comment.

All debugging messages should be printed to `stderr`. We do not expect any output to be printed to `stdout`. It is important that if the instructor has a working version of `pe02.c`, it should be compilable with your `answer02.c` to produce an executable. For evaluation purpose, we will use different combinations of your submitted `.c` files and our `.c` files to generate different executables. If a particular combination does not allow an executable to be generated, you do not get any credit for the function(s) that the executable is supposed to evaluate. We use both `-g` and `-O3` flags (separately) to compile your submission. Your submission is evaluated only when compilation using each of the flags is successful.

Be aware that we set a time-limit for each test case based on the size of the test case. If your program does not complete its execution before the time limit for a test case, it is deemed to have failed the test case. We will not announce the time-limits that we will use. They will be very reasonable.

The `char_to_int` function accounts for 20%, the `str_to_long_int` function accounts for 80%. For the `str_to_long_int` function, the converted long integer accounts for 80%, the `errno` accounts for 20%.

The occurrence of any memory issues (memory errors or memory leaks flagged in a `valgrind` report) will result in 50-point penalty.

7 A few points to remember

Your `answer02.c` should not include `stdlib.h` and it should not call `strtol`. If your submission has any occurrence of `stdlib.h` or `strtol`, your submission will receive a 0 grade, even if such an occurrence appears in a comment.

All debugging messages should be printed to `stderr`. We do not expect any output to be printed to `stdout`.

You are not submitting `answer02.h`. Therefore, you should not make changes to `answer02.h`.

You can declare and define additional static functions that you have to use in `pe02.c` and `answer02.c`.

You should not use macros that start with `T_`.

Grading of programming exercises and assignments is performed on machines with similar setup as eceprog.ecn.purdue.edu. You should perform testing of your work on eceprog.ecn.purdue.edu before submission. Correct output on your computer does not translate into correct output on eceprog.ecn.purdue.edu.