System Programming in C – Homework Exercise 4

Publication date: *Thursday, May 16, 2024*
Due date: *Sunday, June 2, 2024* @ 21:00

The purpose of this assignment is to implement a new data type (which we will call *new word*) that represents short words using a single **`int`** (four bytes on our server). This data type uses an encoding for characters that is an alternative to ASCII, which we call the ASI encoding (short for ASCII :-)). We describe below this encoding and representation of words. In Problem 1 you are asked to implement several functions for this data type and in Problem 2 you are asked to implement a short program that uses these functions to parse text.

The ASI encoding

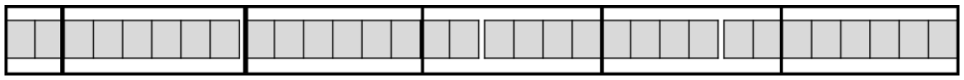
The *new word* data type is used to represent words that are made up of letters and digits. To this end, we define a special encoding for all letter and digit characters, as well as the period character `'.'`. Since this is a compact alternative to the standard ASCII encoding for characters, we call it the ASI encoding, and define it as follows:

- Character `'.'` - ASI value 1.
- Characters `'A'-'Z'` - ASI values 2-27 (in order).
- Characters `'a'-'z'` - ASI values 28-53 (in order).
- Characters `'0'-'9'` - ASI values 54-63 (in order).

We use ASI value 0 to encode an "empty" character, which is used to mark the end of the word. Since the maximum ASI value is 63, we can encode ASI values using six bits ($2^6 = 64$).

New words

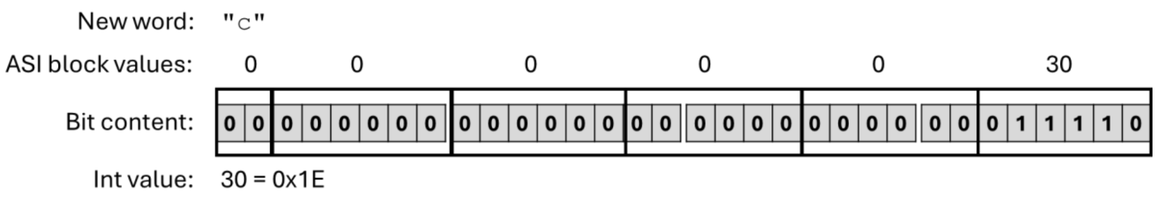
We use a single **`int`** value (four bytes on our server) to represent a sequence of letters and digits using ASI values. We partition the bits used by the **`int`** into consecutive ASI blocks, where each ASI block consists of six bits. On our server, **`int`** takes up four bytes (32 bits), so it can hold five full ASI blocks and a partial ASI block at the end with two bits. The figure below illustrates this.

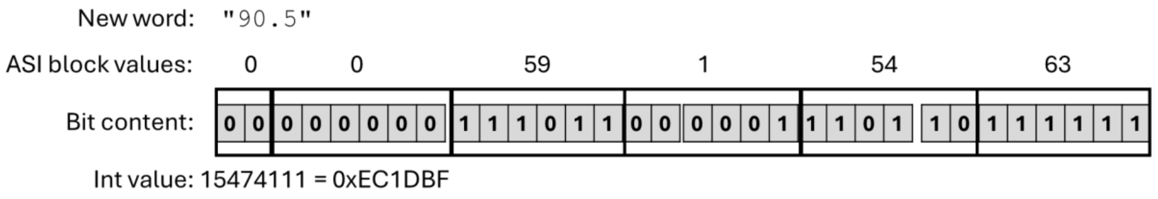


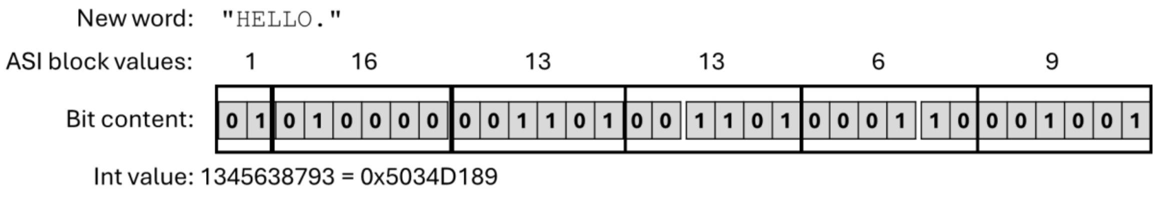
The right-most ASI block encodes the first character in the word, and the ASI block encoding the last character in the word is followed by an ASI block with 0 value. The partial ASI block at the end is not used to encode a letter or a digit, but it can hold ASI value 1 (encoding `'.'`) or ASI value 0.

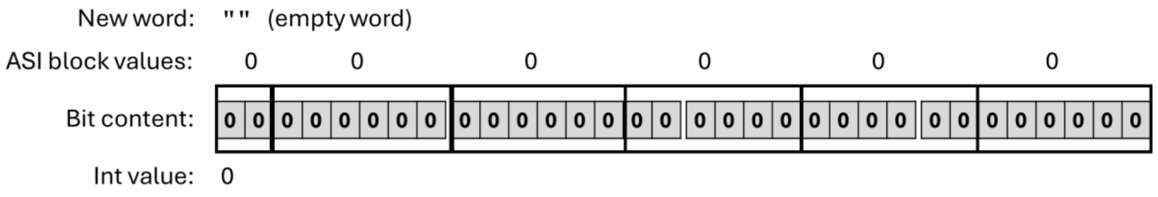
We provide several examples of encodings for different words. For each word, we specify the following:

- The word as a string of characters
- The sequence of ASI values for its characters (from right to left)
- The content of the (32) bits of the **`int`**
- The value of the int in decimal form (base 10) and hexadecimal form (base 16)


New word: "c"
ASI block values: 0 0 0 0 0 30
Bit content: 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 1 1 1 0
Int value: 30 = 0x1E


New word: "90.5"
ASI block values: 0 0 59 1 54 63
Bit content: 0 0 0 0 0 0 0 0 1 1 1 0 1 1 0 0 0 0 0 1 1 1 0 1 1 0 1 1 1 1 1 1
Int value: 15474111 = 0xEC1DBF


New word: "HELLO."
ASI block values: 1 16 13 13 6 9
Bit content: 0 1 0 1 0 0 0 0 0 0 1 1 0 1 0 0 1 1 0 1 0 0 0 1 1 0 0 0 1 0 0 1
Int value: 1345638793 = 0x5034D189


New word: "" (empty word)
ASI block values: 0 0 0 0 0 0
Bit content: 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
Int value: 0

Problem 1:

The purpose of this question is to implement functions for the *new word* data type based on the specification given on pages 1-2. Copy the source file `newWord.c` from the shared directory `/share/ex_data/ex4/` to your exercise directory `~/exercises/ex4/`. Review the code and documentation written in this file. You will notice that the file contains declarations and partial (faulty) implementations of seven functions, which you will be asked to complete here.

Implementation guidelines:

- Follow the specific guidelines for Problems 1.1-1.7 below.
- Modify the source file `newWord.c` only in the clearly designated places.
- Use short and simple implementations and write brief inline documentation, when appropriate. Your code is reviewed manually by the graders and 5 points will be allocated for coding style.
- Avoid explicitly assuming a specific size for data type **`int`**. In particular, we know that on our server, **`int`** takes up four bytes (32 bits), but your code should avoid assuming this explicitly. You may use the **`sizeof`** operator when appropriate. Recall that this operator returns the number of bytes that a certain type (or variable) takes.
- Note that some of the requirements may not be covered by our testing scripts and may be examined manually by the graders.

Compilation and testing: Testing your solutions requires you to compile your code in `newWord.c` with the test code file `/share/ex_data/ex4/test_ex4.c`. This file contains a main function, which runs tests for every function you implement. **You are encouraged to examine this code when you implement each function**. You "turn on" a specific test by compiling your source file together with `test_ex4.c` and using the `-D TEST_1_<SECTION>` option, where `<SECTION>` is the section number (1-7). For example, to test Problem 1.2, you set `<SECTION>` to `2` and use the following compilation command:

```
==> gcc /share/ex_data/ex4/test_ex4.c newWord.c \
          -Wall -D TEST_1_2 \
          -o test_ex4_1_2
```

[We discuss compilation of multi-file programs in detail in Lecture #11]

Compile your code using the flags mentioned above (including `-Wall`) and make sure you do not get any error or warning messages. After you successfully compile your code, run the resulting executable program (`test_ex4_1_2` in the example above) and compare the output with the expected output provided in file `/share/ex_data/ex4/test_ex4_1_<SECTION>.out` , as follows:

```
==> ./test_ex4_1_2 | diff - /share/ex_data/ex4/test_ex4_1_2.out
```

Run this test after you complete each of the tasks in Problems 1.1-1.7 below by replacing the '2' above with the appropriate section number.

1. Function **`charToASI`** receives a single **`char`** parameter, **`ch`**, and it returns the ASI value of the character in **`ch`**, as defined on page 1. If **`ch`** is not a letter, digit or `'.'`, the function returns -1. Add your code between the two comment lines in function **`charToASI`** and avoid making changes elsewhere in the code.

   Execution examples:
   - **`charToASI('.')`** returns 1
   - **`charToASI('B')`** returns 3
   - **`charToASI('W')`** returns 24
   - **`charToASI('e')`** returns 32
   - **`charToASI('r')`** returns 45
   - **`charToASI('0')`** returns 54
   - **`charToASI('9')`** returns 63
   - **`charToASI('-')`** returns -1
   - **`charToASI(' ')`** returns -1

2. Function **`ASItoChar`** receives a single **`int`** parameter, **`ASIval`**, and it returns the character encoded by this ASI value, as defined on page 1. If **`ASIval`** holds a value outside the range of acceptable ASI values (1-63), the function returns the character `'!'`. This function is partially implemented in `newWord.c` using a **`switch`** statement, but the implementation is incorrect (for information on **`switch`** statements, see guide to C control flow on Piazza). Modify the implementation by replacing the values 121-126 with the appropriate expressions so that the function returns the expected character. Do not add any code outside of these expressions.

   Execution examples:
   - **`ASItoChar(1)`** returns `'.'`
   - **`ASItoChar(3)`** returns `'B'`
   - **`ASItoChar(24)`** returns `'W'`
   - **`ASItoChar(32)`** returns `'e'`
   - **`ASItoChar(45)`** returns `'r'`
   - **`ASItoChar(54)`** returns `'0'`
   - **`ASItoChar(63)`** returns `'9'`
   - **`ASItoChar(64)`** returns `'!'`
   - **`ASItoChar(0)`** returns `'!'`
   - **`ASItoChar(-2)`** returns `'!'`

3. Function **`getASIblock`** receives two **`int`** parameters, **`newWord`** and **`ind`**, and it returns the ASI value held by the **`ind`**th ASI block of the new word represented by **`newWord`**. The right-most ASI block has index 0, the one to the left of it has index 1, etc. The function is implemented using a single **`return`** statement, which currently returns the value 131. Replace this value with an expression so that the function returns the appropriate value. Follow these guidelines:
   - You may assume that **`ind`** is within its allowed range, meaning that it is non-negative and not above its maximum value (5 in our case). Your expression does not have to check this.
   - If the **`ind`**th ASI block is the partial block in the end of the new word, then the ASI value is the one represented by these bits. In our case of a 32-bit **`int`**, this block is associated with **`ind`**=5 and it consists of two bits, and so its ASI value is in the range 0-3.
   - Your expression should only use **bitwise** and **arithmetic** operators.
   - In your expression, use the **symbolic constant `BITS_PER_ASI_BLOCK`** to refer to the size of an ASI block (6). Do not use any constant literals (like 6) that assume this specific value. See more information on symbolic constants below.

   Execution examples: (see examples of *new words* specified on page 2)

• **`getASIblock(30,0)`**	returns **`30`**	[word is "c"]
• **`getASIblock(30,1)`**	returns **`0`**	
• **`getASIblock(30,5)`**	returns **`0`**	
• **`getASIblock(15474111,0)`**	returns **`63`**	[word is "90.5"]
• **`getASIblock(15474111,3)`**	returns **`59`**	
• **`getASIblock(15474111,4)`**	returns **`0`**	
• **`getASIblock(1345638793,1)`**	returns **`6`**	[word is "HELLO."]
• **`getASIblock(1345638793,4)`**	returns **`16`**	
• **`getASIblock(1345638793,5)`**	returns **`1`**	

Symbolic constants: Symbolic constants are values defined in a C source file using a **`#define`** directive. In `newWord.c` we make use of **`BITS_PER_ASI_BLOCK`** to represent the number of bits per ASI block (6). A **`#define`** directive instructs the compiler to replace every occurrence of the symbolic constant (**`BITS_PER_ASI_BLOCK`**) with its value (6). Symbolic constants are not variables in the program, but they enable more modular code. In our case, they will enable us to change the size of the ASI block simply by changing the value (6) associated with **`BITS_PER_ASI_BLOCK`**. We discuss symbolic constants in Lecture #8.

4. Function **`isWordFull`** receives a single **`int`** parameter, **`newWord`**, and it returns 1 if the <u>last complete</u> ASI block in the new word represented by **`newWord`** is used (i.e., has non-zero value), and 0 otherwise. In our case of a 32-bit **`int`**, the last complete ASI block is associated with **`ind`** = 4. However, your expression should not assume a specific size for **`int`**. You may use the **`sizeof`** operator, if needed. Moreover, as in (3) above, you should use the symbolic constant **`BITS_PER_ASI_BLOCK`** if you need to make assumptions about the size of an ASI block. The function is implemented using a single **`return`** statement, which currently returns the value 141. Replace this value with an expression so that the function returns the appropriate value. Your expression may contain a call to function **`getASIblock`**.

   Execution examples:

• **`isWordFull(0x1E)`**	returns 0	[word is "c"]
• **`isWordFull(0xEC1DBF)`**	returns 0	[word is "90.5"]
• **`isWordFull(0x4EC1DBF)`**	returns 1	[word is "90.5C"]
• **`isWordFull(0x5034D189)`**	returns 1	[word is "HELLO."]
• **`isWordFull(0x1034D189)`**	returns 1	[word is "HELLO"]
• **`isWordFull(0x134D189)`**	returns 1	[word is "HELL."]
• **`isWordFull(0x34D189)`**	returns 0	[word is "HELL"]

5. Function **`newWordType`** receives a single **`int`** parameter, **`newWord`**, and it returns a number in 0-3 indicating the type of new word that **`newWord`** represents according to the following specification:
   - If the new word is effectively empty, meaning that it consists only of period characters and no letter or digit characters, its type is 0.
   - If the new word is alphabetic, meaning that it has at least one letter and no digit characters, its type is 1.
   - If the new word is a number, meaning that it has at least one digit and no letter characters, its type is 2.
   - If the new word is "mixed", meaning that it has letters and digits, its type is 3.

   The function is implemented using a **`for`** loop, which is currently empty. Add statements to the **`for`** loop so that variable **`type`** will hold the appropriate value. <u>Do not define additional variables</u> (other than the three **`int`** variables already defined), and you may use calls to functions implemented in (1)-(4) above.

Execution examples: (see examples of *new words* specified on page 2)

`newWordType(0)`	returns 0	[word is ""]
`newWordType(1)`	returns 0	[word is "."]
`newWordType(0x1E)`	returns 1	[word is "c"]
`newWordType(0xEC1DBF)`	returns 2	[word is "90.5"]
`newWordType(0x4EC1DBF)`	returns 3	[word is "90.5C"]
`newWordType(0x5034D189)`	returns 1	[word is "HELLO."]
`newWordType(0x1034D189)`	returns 1	[word is "HELLO"]
`newWordType(0x3F34D189)`	returns 3	[word is "HELL9"]

6. Function **`appendWordChar`** receives an **`int`** parameter, **`newWord`**, and a **`char`** parameter, **`ch`**, and it returns the **`int`** representing the new word obtained by appending **`ch`** to the new word represented by **`newWord`**. The end of the new word is defined as the first ASI block that has zero value. In the following cases, the character cannot be appended, and the original new word is returned:
    - When the character is not a letter, digit, or a period (`'.'`).
    - When all ASI blocks have non-zero values.
    - When the only ASI block with zero value is the partial block in the end and **`ch`** is not the period character (`'.'`).

   The function is currently implemented with a single **`return`** statement. Replace this statement with a series of statements that will correctly implement the function as specified above. You may use calls to functions implemented in (1)-(5) above. You may assume here that all ASI blocks after the first ASI block with zero value also have zero value.

   Execution examples: (we gradually create the *new word* "90.5A.")

`appendWordChar(0x0, '9')`	returns `0x3F`	[word is "9"]
`appendWordChar(0x3F, '-')`	returns `0x3F`	[invalid character]
`appendWordChar(0x3F, '0')`	returns `0xDBF`	[word is "90"]
`appendWordChar(0xDBF, '.')`	returns `0x1DBF`	[word is "90."]
`appendWordChar(0x1DBF, '5')`	returns `0xEC1DBF`	[word is "90.5"]
`appendWordChar(0x1DBF, '@')`	returns `0xEC1DBF`	[invalid character]
`appendWordChar(0xEC1DBF, 'A')`	returns `0x2EC1DBF`	[word is "90.5A"]
`appendWordChar(0x2EC1DBF, 'A')`	returns `0x2EC1DBF`	[word is full]
`appendWordChar(0x2EC1DBF, 'x')`	returns `0x2EC1DBF`	[word is full]
`appendWordChar(0x2EC1DBF, '3')`	returns `0x2EC1DBF`	[word is full]
`appendWordChar(0x2EC1DBF, '.')`	returns `0x42EC1DBF`	[word is "90.5A."]
`appendWordChar(0x42EC1DBF, '.')`	returns `0x42EC1DBF`	[word is full]
`appendWordChar(0x42EC1DBF, 'x')`	returns `0x42EC1DBF`	[word is full]

7. Function **`printNewWord`** receives a single **`int`** parameter, **`newWord`**, and it prints to the standard output the new word represented by **`newWord`**. The function returns no value. If the last character in the new word is a period (`'.'`), the function prints a new line (`'\n'`) after the word, otherwise it prints a space character (`' '`). The function is currently implemented with a single empty **`return`** statement. Add before this statement a series of statements that will correctly implement the function as specified above. You may use calls to functions implemented in (1)-(6) above. You may assume here that all ASI blocks after the first ASI block with zero value also have zero value.

   Execution examples: (see examples in (3)-(6) above)

• **`printNewWord(0x1E)`**	prints **`c`**	(followed by a space)
• **`printNewWord(0x3F)`**	prints **`9`**	(followed by a space)
• **`printNewWord(0xDBF)`**	prints **`90`**	(followed by a space)
• **`printNewWord(0xEC1DBF)`**	prints **`90.5`**	(followed by a space)
• **`printNewWord(0x4EC1DBF)`**	prints **`90.5C`**	(followed by a space)
• **`printNewWord(0x1DBF)`**	prints **`90.`**	(followed by a newline)
• **`printNewWord(0x5034D189)`**	prints **`Hello.`**	(followed by a newline)
• **`printNewWord(0x1034D189)`**	prints **`Hello`**	(followed by a space)
• **`printNewWord(0x3F34D189)`**	prints **`Hell9`**	(followed by a space)

Final testing for Problem 1: After you finished implementing all seven functions, and you individually tested every function using the guidelines given on page 3, you should ensure that your code provides the expected output for all functions by executing the test script `/share/ex_data/ex4/test_ex4.1` from the directory containing your `newWord.c` source file. This script produces a detailed error report to help you debug your code.

Problem 2:

The purpose of this question is to implement a program that uses the new word data type to process text. The program reads a stream of text from the standard input, detects words in this stream of text, and prints them in an orderly fashion to the standard output. At the end, it prints statistics about the printed word.

Program description:

- A word in this context is a consecutive sequence of characters that can be encoded using the ASI encoding defined on page 1 (letters, digits, and the period character `'.'`).
- A word in the input stream ends either when followed by a character that cannot be encoded using the ASI encoding, or when it is of maximum length that can be represented by a new word (on our server this is five characters and an optional period at the end). Thus, a word ends when appending the next character in the stream to it using **`appendWordChar`** fails (see Problem 1.6).
- Once a word ends, it is printed to the standard output using function **`printNewWord`** (see Problem 1.7). Thus, if it ends with a period (`'.'`), a new line (`'\n'`) is printed after it, and otherwise a space (`' '`) is printed.
- When a dollar character `'$'` is read in the input stream, the program halts.
- Before halting, the program prints the number of words it printed in each of the types defined by function **`newWordType`** (see Problem 1.5). The message should have the following format:
```
Number of words is <NUM_TYPE1>
Number of numbers is <NUM_ TYPE2>
Number of mixed words is <NUM_ TYPE3>
```
- See the execution examples on page 11 for specific demonstrations of the expected behavior of the program.

Implementation guidelines: The program should be implemented in a source file named `newWordParser.c`. Copy the source file `newWordParser.c` from the shared directory `/share/ex_data/ex4/` to your exercise directory `~/exercises/ex4/`. The initial version of `newWordParser.c` contains an `#include <stdio.h>` directive (that will enable you to call **`printf`** and **`scanf`**), declarations of the seven functions that you implemented in Problem 1, and an empty **`main`** function. You should write a definition for the main function that implements the program described above and follow these guidelines:

- Your code may contain calls to functions that you implemented in Problem 1, as well as the **`printf`** and **`scanf`** functions from the `stdio` library.
- Read characters from the input stream one by one using a call to **`scanf`** with conversion specifier `%c`.

- Your code should not explicitly assume a specific size for any type. In particular, it should not assume that a new word can contain at most five characters (and an optional period at the end). This information should be conveyed through calls to functions that you implemented in Problem 1.
- Do not add any `#include` directives to the code and do not use any library functions other than **`printf`** and **`scanf`**.
- Write clear and readable code. Use appropriate indentation and try to follow the style of `newWord.c` as much as possible. You should also add brief documentation for the critical parts of your implementation. Keep in mind that your code is reviewed by the graders and 5 points will be allocated for code style.
- This program can be implemented using a relatively short **`main`**, so there is no need to implement additional functions.

Compilation and testing: Testing your solution requires you to compile the source files `newWord.c` and `newWordParser.c` together. This should be done using the following compilation command:

```
==> gcc -Wall newWordParser.c newWord.c -o newWordParser
```

[We discuss compilation of multi-file programs in detail in Lecture #11]

Compile your code using the flags mentioned above and make sure you do not get any error or warning messages. If compilation is successful, it creates an executable program called `newWordParser`, which you can run and test on basic execution examples provided on page 11, as well as additional examples.

To help you test your code, we provide a working executable in `/share/ex_data/ex4/newWordParser`. You should prepare a set of inputs and compare your output with the output of our program. You may use the execution examples from the next page as a base set of inputs.

Finally, when you are convinced that your program works well, execute the testing script `/share/ex_data/ex4/test_ex4.2` from the directory containing your `newWordParser.c` and `newWord.c` source files. The script produces a detailed error report that may help you find additional bugs in your code.

Execution examples:

```
==> echo "helloworld2024.1st place$ this part is not read" | ./newWordParser
```

```
hello world 2024.
1st place
Number of words is 3
Number of numbers is 1
Number of mixed words is 1
```

```
==> echo $?
0
```
(← exit status should always be 0 for this program)

Explanation: the text `helloworld2024.1st place$ this part is not read` is streamed into the program. The program reads until the first dollar character `'$'`. This part of the text contains five new words, which are printed to the standard output in two lines. The transition between the first four words (until `1st`) is defined by the maximum length of a new word. The word `2024.` Is the only one that ends with a period, so a newline is printed after it. The word `1st` ends because it is followed by the space character `' '`, which does not have an ASI encoding. After the five words are printed, the program prints the counts of each type: three words (`hello world place`), one number (`2024.`), and one mixed (`1st`).

```
==> echo "How is everyone doing?We hope you're 100%.$" | ./newWordParser
How is every one doing We hope you re 100 .

Number of words is 9
Number of numbers is 1
Number of mixed words is 0
```

```
==> echo "one 2 thr33. 4.4 f1ve six.7 eight nine.Ten$" | ./newWordParser
```

```
one 2 thr33.
4.4 f1ve six.7 eight nine.
Ten
Number of words is 4
Number of numbers is 2
Number of mixed words is 3
```

```
==> echo "Good.luck.everyone.Hope.you.enjoyed.your.1st.assignment.in.C$" \
         | ./newWordParser
Good.
luck.
every one.H ope.y ou.en joyed.
your.
1st.a ssign ment.
in.C
Number of words is 11
Number of numbers is 0
Number of mixed words is 1
```

Submission Instructions:

1. After you validated and tested your solution, make sure that your `~/exercises/ex4/` directory contains the following C source files, which includes your implementation:
   - `newWord.c`
   - `newWordParser.c`
2. your `~/exercises/ex4/` directory should also contain a `PARTNER` file with the user id of the non-submitting partner. The non-submitting partner should also add a `PARTNER` file containing the user id of the submitting partner.
3. Check your solution by running **`check_ex ex4`**. The script should be executed from the account of the submitting partner, and it may be run from any directory. Clean execution of this script guarantees you 80% of the assignment's grade.
4. Once you are satisfied with your solution, you may submit it by running **`submit_ex ex4`**. The script should be executed from the account of the submitting partner, and it may be run from any directory. You may modify your submission any time before the deadline (**2/6 @ 21:00**) by running **`submit_ex ex4 -o`** from any location.
5. For more information on the submission process, see the **Homework submission instructions** file on the course website.