See handout for our due date

Program 3: morse

Fall 2010

Due Friday, Oct 22, 2010 @ 11:45pm

This programming assignment must be completed individually. Do not share your code with or receive code from any other student. Evidence of copying or other unauthorized collaboration will be investigated as a potential academic integrity violation. **The minimum penalty for cheating on a programming assignment is a grade of -100 on the assignment.** If you are tempted to copy because you're running late, or don't know what you're doing, you will be better off missing the assignment and taking a zero. Providing your code to someone is cheating, just as much as copying someone else's work.

DO NOT copy code from the Internet, or use programs found online or in textbooks as a "starting point" for your code. Your job is to design and write this program from scratch, on your own. Evidence of using external code from any source will be investigated as a potential academic integrity violation.

This program will translate between ASCII characters and Morse Code. Morse Code uses a series of lights, clicks, or tones to represent letters, numbers, and punctuation marks. It was developed for telegraph operators, though the current code is different from the original. This program will translate both ways: from ASCII to Morse, and from Morse to ASCII.

The learning objectives of the program are:

- Using strings and characters.
- Using file I/O – reading from and writing to files.
- Using command-line arguments.

Background

International Morse Code uses a combination of "dots" (short-duration tones) and "dashes" (long-duration tones) to represent letters, numbers, and some punctuation marks. We will use a textual representation of Morse Code, where a dot is represented by a period ('.', ASCII 46) and a dash is represented by a hyphen ('-', ASCII 45). Each letter will be represented by a string of dots and dashes, as specified in Tables 1-3, with no spaces. For example, the letter 'L' is written as ".-.." (dot-dash-dot-dot).

Morse Code is case-insensitive; there is no distinction between upper-case 'A' and lower-case 'a'. When we translate ASCII to Morse, all lower-case letters will be converted to upper-case. When we are converting from Morse to ASCII, only upper-case letters will be generated.

In our textual representation of Morse Code, a linefeed is used to separate characters in a word, and a special string ("###" followed by a linefeed) is used to separate words. A "word" is simply a contiguous collection of letters, numbers, or symbols without any spaces. Punctuation marks will be considered part of a word, if there is no space between the letters and the punctuation marks. (E.g., "what?" is a single word with five characters.)

This program will require you to read and write text from files. We will also be using command-line arguments to specify file names and the type of translation. We will discuss all of these issues in class, but for more information, see the following chapters in the C Primer Plus book:

- Command-line arguments: Chapter 11
- Strings and string functions: Chapter 11
- File I/O: Chapter 13 – we will only use text files, and will only need **fopen**, **fclose**, **fscanf**, and **fprintf**

Char	Code	Char	Code	Char	Code
A	.-	J	.---	S	...
B	-...	K	-.-	T	-
C	-.-.	L	.-..	U	..-
D	-..	M	--	V	...-
E	.	N	-.	W	.--
F	..-.	O	---	X	-..-
G	--.	P	.--.	Y	-.--
H	....	Q	--.-	Z	--..
I	..	R	.-.		

Table 1. Morse Code: Letters.

Char	Code	Char	Code	Char	Code	Char	Code
0	-----	3	...--	6	-....	9	----.
1	.----	4	....-	7	--...		
2	..---	5	.....	8	---..		

Table 2. Morse Code: Numbers.

Char	Code	Char	Code	Char	Code	Char	Code
Period [.]	.-.-.-	Open Paren [(]	-.--.	Equal [=]	-...-	Dollar Sign [$]	...-..-
Question Mark [?]	..--..	Close Paren [)]	-.--.-	Plus [+]	.-.-.	At Sign [@]	.--.-.
Apostrophe [']	.----.	Ampersand [&]	.-...	Minus, Hyphen [-]	-....-	Comma [,]	--..--
Exclam. Point [!]	-.-.--	Colon [:]	---...	Underscore [_]	..--.-		
Slash [/]	-..-.	Semicolon [;]	-.-.-.	Quotation Mark ["]	.-..-.		

Table 3. Morse Code: Punctuation

You also need to make sure your files are stored in the project directory. This is the same directory where your C source files, e.g., morse.c, are stored.

Program Specification: User Interface

The program executable will be named **morse**. The user invokes the program with the following command:

```
morse direction input_file output_file
```

The *direction* argument is a string that tells the direction of the translation: "a2m" means ASCII to Morse, and "m2a" means Morse to ASCII. The *input_file* is the name of the file that contains the text to be translated. The *output_file* is the name of the file where the translated text will be written.

If you are using NetBeans, or some other IDE, you'll need to figure out how to pass arguments to your program. In NetBeans, open the "Project Properties" display and click on "Run". There should be an entry on the right labeled "Arguments". Put the arguments that would appear on the command line in this box (e.g., a2m sample.txt sample_morse.txt). For other IDE's, a Google search or (gasp!) reading the manual should tell you how to do this.

If the output file does not exist, it will be created. If the output file already exists, it will be overwritten with the translated text. NOTE: If a program error occurs, the content of the output file is undefined. (The output file will not be checked after an error.)

Example – to translate a file named "sample.txt" from ASCII to Morse Code, storing the result in a file named "morse.txt":

```
morse a2m sample.txt morse.txt
```

Error handling

If the input file does not exist or cannot be opened for reading, the program must print the following text and return the value EXIT_FAILURE:

```
Cannot open input file.
```

If the output file cannot be opened for writing, the program must print the following text and return the value EXIT_FAILURE:

```
Cannot open output file.
```

If the wrong number of arguments are entered, or if direction is not "a2m" or "m2a", the program must print the following text and return the value EXIT_FAILURE:

```
Usage: morse <direction> <input_file> <output_file>
```

If any character in the input file cannot be translated (details in the next section), the program must print the following text and return the value EXIT_FAILURE:

```
Cannot translate input file.
```

Program Specification: File Formats

The input file for ASCII-to-Morse translation is a simple text file. It may contain any of the characters listed in Tables 1-3, as well as lower-case letters and whitespace characters: space, linefeed, tab.

Whitespace characters simply serve as the separators for words – they are not translated. Lower-case letters must be converted to upper-case for translation. Any other characters cannot be translated, and will result in a translation error.

The output file for ASCII-to-Morse translation has the Morse Code string for one character on each line. A linefeed must follow the end of the code string. At the end of the word, the string "###" followed by a linefeed must be printed. There are no spaces or empty lines in a Morse Code file.

The input file for Morse-to-ASCII translation is the same format described in the previous paragraph.

The output file for Morse-to-ASCII translation has one word per line, all upper-case, and no spaces.

Examples of all three file formats are given in the Appendix.

Program Specification: Source Code Files

You'll need to add a variable name when you write your function.

Your source code must be split into two files.

The first file is named **translate.c**, and it is responsible for defining the following two functions:

```
const char *char2morse(char);
```

This function takes a character parameter and returns a pointer to the Morse Code encoding for that character. Note that the return pointer is **const** – this means that the calling function cannot change the content of the string after it is returned. If the translation cannot be performed (e.g, the character is not in one of the translation tables), then the return value must be zero.

NOTE: A pointer with value zero is known as a "null pointer." There is a special symbol `NULL` that can be used to represent the null pointer. `NULL` is defined in **stdlib.h**, so you must include that header file if you want to use the symbol. Don't confuse a null pointer (`NULL`) with the null character (`'\0'`).

```
char morse2char(const char *);
```

This function takes a pointer to a Morse code string, and returns the character that corresponds to that string. If the translation cannot be performed (e.g., the string is not found in one of the translation tables), then the return value must be zero (the null character).

Declarations for these functions are provided in **translate.h**.

The second source file must be named **morse.c**. It contains the `main()` function, and any other functions that you'd like to create to implement the program. You must call the `char2morse()` and `morse2char()` functions from this file. (You'll need to include **translate.h**, so that functions in your **morse.c** file know about the translation functions.)

You will also be provided with a header file named **tables.h**, which contains global variable declarations for three translation arrays, corresponding to Tables 1-3 in this document. *Use of this file is optional. If you use it, you may include it in only one C file.* (Otherwise, you'll see a compiler error because the global variables will be declared more than once. There is a better way to handle this in general, and we'll eventually discuss it in class. For this program, include this file in **translate.c** only, if you use it at all.)

Compiling

As with Program 2, we have two different source (**.c**) files to compile and link.

1. Compile **morse.c** file, creating **morse.o**:

```
gcc -c -g -Wall -pedantic morse.c
```

This was a little over 100 lines of C for me.

My translate.c is 41 lines of uncommented code.

2. Compile **translate.c** file, creating **translate.o**:

```
gcc -c -g -Wall -pedantic translate.c
```

3. Link **morse.o** and **translate.o** to create an executable named **morse**:

```
gcc -o morse -g morse.o translate.o
```

Progam Design: Suggestions

We've already made one decision for you, by specifying two functions that translate a single character to Morse Code, and vice versa. In this section, we'll talk more about those two functions, as well as the flow of the main function. (You can create additional functions to split the program into modules; this is highly recommended. However, we will talk about everything other than the translate functions as if it were done in the main function.)

ASCII-to-Morse

In the **tables.h** file, we have provided four arrays (tables) to help in translating from an ASCII character to the corresponding Morse Code string. Consider the array named `letterTable`, declared as follows (with some parts left out):

```
char *letterTable[26] = { ".-",   /* A */
                          "-...", /* B */
                          "-.-.", /* C */
                          /* etc.... */
                        };
```

Comparing this data to Table 1, you can see that the array contains each Morse Code string corresponding to each letter, in alphabetical order. Therefore, to lookup the string for a particular character (after converting to upper-case), we only need to index into the table at the correct location.

The index for A is 0, for B is 1, for C is 2, etc. See the pattern? The index is the "distance" between the desired letter and 'A'. Since the ASCII code puts all the letters together in one contiguous sequence, we can simply subtract 'A' from the letter, and we have the distance.

The same is true for the `numberTable` array, except that we subtract the character '0' (zero) to get the index into the table.

It's a little more complicated for the punctuation marks. They are not contiguous in the ASCII table, and there are some ASCII punctuation marks for which we don't have a Morse Code string in Table 3. For this translation, we use two arrays. First, we find the character in the array named `punctMarks`. (Note that this array is initialized as a string; it's an array of characters, so why not? The notation is more convenient that way.) If there's a match, then the index at which the character appears in punctMarks is the same as the index of the corresponding Morse Code string in the array `punctTable`.

To summarize the ASCII-to-Morse translation process: (1) Determine whether the character is a letter, number, or punctuation mark. (There are standard library functions for this purpose – see page 229 of C Primer Plus.) (2) Convert the character into an index into the proper table. (3) Return the pointer stored in the table at that index.

You'll need
#include <ctype.h>
#include <string.h>

Morse-to-ASCII

For translating Morse Code to an ASCII character, we need to match the string to one of the entries in the table. We don't know which table to look in, so we have to (potentially) look at all three. (Remember to use `strcmp()` instead of == to compare the input string to the string in the table.)

If we find the string, we use the index of the matching element to figure out the corresponding character. If it's a letter, add the index to 'A'. If it's a number, add the index to '0'. If it's a punctuation mark, use the index to look up the character in the `punctMarks` array.

Main Function – Start

Use `argc` (see Chapter 11) to see whether the correct number of arguments have been provided. Check the first argument (`argv[1]`) to make sure it's either "a2m" or "m2a". Open the file specified by the input filename (`argv[2]`) for read-only access. (See Chapter 13.) Open the file specified by the output filename (`argv[3]`) for write-only access. If an error occurs at any of these steps, print the proper error string and return.

Translating an ASCII File

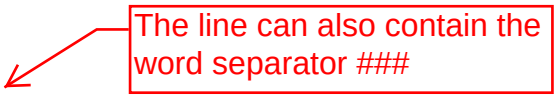
If the first argument is "a2m", then we are translating from an ASCII file to a Morse Code file. We recommend that you read one character at a time (`%c`), because we don't know anything about the potential length of words in the input file. (This lack of knowledge makes it difficult to safely use `%s` for reading a string at a time. Plus, you'd just have to access each character of the string anyway.)

When you read a character from the file, there are three possible outcomes:

1. The character read is a non-whitespace character. (See the `isspace()` function on page 229.) Translate the character and print the Morse Code string, followed by a linefeed, to the output file. (If the character could not be translated, it's an error.)
2. The character read is a whitespace character. This indicates the end of a word. Print the word separator ("###" and a linefeed) to the output file.
3. The return value from `fscanf` is `EOF` (end of file). This means there are no more characters to be read. The progam can exit. (See the "Main Function – End" section below.)

Keep reading characters from the input file until `EOF` is returned.

Translating a Morse Code File



In the Morse Code file, there is one code string per line. In this case, we know how big the largest Morse Code string is (see Tables 1-3), so we can safely allocate an array large enough to hold the largest string. So we recommend using `%s` to read the entire string at once.

(NOTE: The string will not contain the linefeed character at the end of the line. When you read the next string, `fscanf` will skip over the linefeed if you use the `%s` format code.)

Pass the string to the translate function, and print the returned character to the output file. (If the string cannot be translated, it's an error.)

As above, when `EOF` is returned from `fscanf`, it means you've reached the end of the file.

Main Function – End

Before returning, close the input and output files. Then return either EXIT_SUCCESS or EXIT_FAILURE, depending on whether any errors occurred during the translation process.

Testing the Program

You can write a simple test program to test your translation functions before you write **morse.c**. This is a very good idea. Make sure to test letters, numbers, and punctuation marks. And test your functions with characters/strings that cannot be translated.

NOTE: We are likely to use this approach to test your translation functions.

Once you have translation working, then write **morse.c**. Test under various conditions, with different input files. For example:

- Test your argument processing by leaving off one or more arguments, or put something bogus in the direction argument.
- Try to translate an input file that doesn't exist.
- Create a read-only file and try to use it as the output file.

```
% touch out.txt
% chmod a-w out.txt
% ./morse a2m sample.txt out.txt
```

If you've tested the translation functions thoroughly, then you don't need much variety in the characters in your sample input files. But you do want to test the word separator, since that's not covered in the translation functions. Also try an input file with a character that cannot be translated, to make sure you handle that case properly.

Hints and Suggestions

- Don’t overcomplicate the program. Do not use anything that we have not covered in class.
- Read the Testing section (above) carefully.
- Write the translation functions first. You can do this before we cover file I/O in class. Then you'll be ready to write the file-handling code for **morse.c**.
- For compiler errors, look at the source code statement mentioned in the error. Try to figure out what the error is telling you. Try to fix the first error first, and then recompile. Sometimes, fixing the first error will make all the other errors go away. (Because the compiler got confused after the first error.)
- Use a source-level debugger, like *kdbg*, to step through the program if the program behavior is not correct. If you are using NetBeans on your own computer, the debugger is integrated with the editor and compiler, so there’s no excuse for not using it.
- For general questions or clarifications, use the Message Board, so that other students can see your question and the answer. For code-specific questions, email your code (as an attachment) to the support list: ece209-001-sup@wolfware.ncsu.edu.
- Read the book. Use the debugger. Ask questions.

Can be downloaded from course web site.

Administrative Info

Updates or clarifications on the Message Board:

Any corrections or clarifications to this program spec will be posted on the Message Board. It is important that you read these postings, so that your program will match the updated specification.

Where to find files:

The **translate.h** and **tables.h** files can be downloaded from the Programs web site, or can be copied from:

/afs/eos/lockers/workspace/ece/ece209/001/common/prog3

What to turn in:

- Source files – they must be named **morse.c** and **translate.c**. You don't need to submit *translate.h* or *tables.h*. Submit via Wolfware to the Program 3 assignment in your problem session section (201, 202, ..., 206). Your grade will be posted on the 209 gradebook, and you will be able to get detailed grading information using the "Retrieve Assignment" option on Wolfware.

Submit via moodle

Grading criteria:

15 points: File submitted with the proper name.

15 points: Compiles with no warnings and no errors (using –Wall and –pedantic). (If the program is not complete, then only partial credit is available here. In other words, you won't get 15 points for compiling a few trivial lines of code.)

10 points: Proper coding style, comments, and headers. No unnecessary global variables. No goto. (See the Programs web page for more information.)

15 points: Function char2morse functions correctly.

15 points: Function morse2char functions correctly.

5 points: Command-line arguments are processed correctly.

10 points: File translated from ASCII to Morse (a2m) correctly.

10 points: File translated from Morse to ASCII (m2a) correctly.

5 points: Errors are handled correctly. (Error messages must exactly match the spec. There must be a linefeed printed after the error message.)

Appendix: Sample Files

The input ASCII file will have exactly one whitespace character (typically a space or linefeed) between each pair of words. Both upper-case and lower-case letters can be used.

The file **sample.txt** is a typical input file. It contains:

```
See Spot run.  Run, Spot, run!
```

When this file is translated to Morse Code (`morse a2m sample.txt sample_morse.txt`), the output file (**sample_morse.txt**) looks like this:

```
...
.
.
###
...
.--.
---
-
###
.-.
..-
-.
.-.-.-
###
.-.
..-
-.
..--..
###
...
.--.
---
-
..--..
###
.-.
..-
-.
-.-.--
###
```

When that file is translated back to ASCII (`morse m2a sample_morse.txt sample_ascii.txt`), the output file (**sample_ascii.txt**) has the following content:

```
SEE
SPOT
RUN.
RUN,
SPOT,
RUN!
```