Name: Jack Stawasz
Course: CSCI 312 Principles of Programming Languages
Assignment Deadline: April 2, 2025

Question 1

1. What is the bash command for generating a `ctags` file? **`ctags -R`**
2. What is the `vi` command for invoking generic keyword autocompletion? **Ctrl+n**
3. What is the `vi` command for invoking current buffer keyword autocompletion? **Ctrl+p**
4. What is the `vi` command for invoking whole line autocompletion? **Ctrl+x Ctrl+l**
5. What is the `vi` command for refining the word list as you type? **Ctrl+x Ctrl+n**

Question 2

1. What is a definition in C? **A definition assigns a name, type, and memory allocation to an object. This can only occur once for any object.**
2. What is a declaration in C? **A declaration informs a program of the name of an existing object, whether it be an `extern` variable or unimplemented function. There is no memory allocation in a declaration.**

Question 3

What are three differences between arrays and pointers?

1. **Pointers hold the address to data while arrays hold the data itself.**
2. **Data accessed from pointers must be loaded in before use while data accessed from an array can be done with subscripting (`arr[i]`) for direct access. Pointer subscripting retrieves data from a relative position from the pointer itself and is not the same as array subscripting.**
3. **Pointers are used for dynamic data holding while arrays can only hold data of a fixed memory size and type.**

Question 4 (Look at the Segments in an Executable)

Make a new directory called `Assignment3` in your ppl repo. Make a new directory called `Assignment3/Question4` that will contain your source code and executables for this question. Complete *Look at the Segments in an Executable* (Expert C Programming p. 142):

1. Implement 1 in a file cAalled `1.c` with an executable called `1.out`. Record your answer to 1 here:
   `size` command categorized sizes **`ls -l` total size**
   text data bss dec hex filename **15960**
   1376 600 8 1984 7c0 1.out

2. Implement 2 in a file called `2.c` with an executable called `2.out`. Record your answer to 2 here:
   `size` command categorized sizes **`ls -l` command total size**
   text data bss dec hex filename **15992**
   1376 600 4032 6008 1778 2.out
   Noted difference: while the file size reported by `ls -l` has barely changed, the total memory usage reported by `size` (mainly in bss, the uninitialized global variable size report) has increased drastically. This is because a large chunk of memory has been allocated to store the array of 1000 ints.

3. Implement 3 in a file called `3.c` with an executable called `3.out`. Record your answer to 3 here:
   `size` command categorized sizes **`ls -l` command total size**
   text data bss dec hex filename **20008**
   1376 4616 8 6000 1770 3.out
   Noted difference: compared to 2.out, much of the bss size category has moved to data size category because even initializing only a single element of an array will mark the entire array as data and therefore the size report will list its contents as data. In turn, this increases the file size reported by `ls -l` because the data of the entire 1000 element array is being stored directly in the file, despite only one element having been initialized.

4. Implement 4 in a file called `4.c` with an executable called `4.out`. Record your answer to 4 here:
   `size` command categorized sizes **`ls -l` command total size**
   text data bss dec hex filename **20096**
   1627 4624 8 6259 1873 3.out
   Noted difference: compared to 3.out, barely anything has changed other than the text size category (due to the function increasing the character count of the code). This means that local variables do not add size to the executable (because they are instead allocated at runtime on the stack).

5. Implement 5 in a file called `5.c` with an executable called `5d.out` for the debugging question and record your answer to the debugging question here:
   File size BEFORE debugging compilation: 15960
   File size AFTER debugging compilation: 17032

6. ...and an executable called `5o.out` for the optimization question and record your answer to the optimization question here:
   File size of optimized debugging compilation: 15960

As you implement new requirements in your assignments for the rest of the semester, continue to use branching to get more practice.

Question 5 (Stack Hack)

Make a new directory called `Assignment3/Question5` that will contain your source code and executables for this question. Complete *Stack Hack* (Expert C Programming p. 146):

1. Compile and run the small test program (to discover the approximate location of the stack on your system) in a file called `stack_hack_1.c` with an executable called `stack_hack_1.out`. Record your answer here:
   The stack top is near 0x7ffc8f8a7054

2. Discover the data and text segment, and the heap within the data segment, in a file called `stack_hack_2.c` with an executable called `stack_hack_2.out`. Record your answer here:
   The stack top is near 0x7ffe97e4030c
   The data segment is near 0x562c85aa7010
   The text segment is near 0x562c85aa4189
   The heap is near 0x562c86cef2a0

3. Make the stack grow in a file called `stack_hack_3.c` with an executable called `stack_hack_3.out`. What's the address of the top of the stack now? Record your answer here:
   BEFORE: The stack top is near 0x7ffde8444210
   AFTER: The stack top is near 0x7ffde8444214

Question 6 (The Stack Frame)

Make a new directory called `Assignment3/Question6` that will contain your source code and executables for this question. Complete *The Stack Frame* (Expert C Programming p. 151):

1. Manually trace the flow of control. Record your answer here:
   Control flow. Note that the printf() function is subjectively included in the control diagram despite its implementation not being directly relevant to this program. Return addresses are the line numbers where the program goes back (aka the line from where the program will continue once the call is returned).
   Call to main() - prev frame n/a, return address n/a
   Call to a(1) - prev frame main(), return address line 10
   Call to a(0) - prev frame a(1), return address line 3
   Call to printf() - prev frame a(0), return address line 5
   Call to a(0) - prev frame a(1), return address line 3
   Return to a(1) - prev frame main(), return address line 10
   Return to main() - prev frame n/a, return address n/a

2. Implement 2 in a file called `main.c` with an executable called `a.out`. Record your answer here: **Control flow stacks printed on next page. Comparison: the debugging**

uses register addresses to store each frame rather than line number references like my flow outline, but overall they are the same.

```
#0  0x000055555555518e in main ()

#0  0x0000555555555151 in a ()
#1  0x0000555555555198 in main ()

#0  0x0000555555555151 in a ()
#1  0x000055555555516c in a ()
#2  0x0000555555555198 in main ()

#0  printf (format=0x555555556004 ”i has reached zero”) at ./stdio-
#1  0x0000555555555182 in a ()
#2  0x000055555555516c in a ()
#3  0x0000555555555198 in main ()

#0  0x0000555555555151 in a ()
#1  0x000055555555516c in a ()
#2  0x0000555555555198 in main ()

#0  0x0000555555555151 in a ()
#1  0x000055555555516c in a ()

#0  0x0000555555555198 in main ()

i has reached zero
[Inferior 1 (process 2603487) exited normally]
```