Name: TODO
Course: CSCI 312 Principles of Programming Languages
Assignment Deadline: March 26, 2025

Question 1

List two ways a typedef differs from macro text replacement:

1. **Macro text-replaced typenames can be extended with additional specifiers (eg adding ”unsigned” to type int) whereas typedef typenames are already ”complete” and cannot be modified.**
2. **A typedef typename provides the type for every declarator in a declaration whereas macro text-replaced typenames are handled by the pre-processor and will not apply to numerous declarators.**

Question 2

Make a new directory called `Assignment2` in your ppl repo. Reimplement *The Piece of Code that Understandeth All Parsing* (Expert C Programming p. 88) in a file called `Assignment2/cdecl.c`:

- You must create, merge, and delete a branch for each function in the program
- Each branch must be named `feature/<function name>`
- Each function must be implemented using no less than two commits
- Each commit message should adhere to the following guidelines:
  - Limit the subject line to 50 characters
  - Capitalize the subject line
  - Do not end the subject line with a period
  - Use the imperative mood in the subject line (*Fix bug* not *Fixed bug* nor *Fixes bug*)
- Write your reflog to a file called `Assignment2/reflog.txt` using the following command: `Assignment2$ git reflog > reflog.txt`
- Push your changes from your local repo to your upstream repo (on GitHub)

Question 3

Use the procedure in the Appendix[1] for deciphering C declarations to decipher the following C declarations. First, replace the TODO marker in each “Me” bullet for each C declaration with your

[1]Bournoutian

translation (using the procedure in the Appendix). Note that not all C declarations in this list are legal. Simply replace the TODO marker with ILLEGAL for illegal declarations. Second, use gcc to compile your `cdecl.c`, and replace the TODO marker in each "cdecl.c" bullet for each C declaration with the output of your program. Third, scan each and every C declaration, and for the ones where your translation differs from the cdecl output, explain the difference. *You will not have points taken off for an incorrect translation as long as you explain the difference.*

1. `int argc;`
    - Me: **argc is int**
    - `cdecl.c`: **argc is int**
    - Explanation (if necessary): **n/a**
2. `int *p;`
    - Me: **p is int**
    - `cdecl.c`: **p is pointer to int**
    - Explanation (if necessary): **I thought p was a variable and a new pointer was being created for this variable, but p is actually a new object that is just a null pointer.**
3. `int a[];`
    - Me: **a is array of int**
    - `cdecl.c`: **a is array of int**
    - Explanation (if necessary): **n/a**
4. `int f();`
    - Me: **f is function returning int**
    - `cdecl.c`: **f is function returning int**
    - Explanation (if necessary): **n/a**
5. `char **argv;`
    - Me: **argv is pointer to char**
    - `cdecl.c`: **argv is pointer to pointer to char**
- Explanation (if necessary): **After looking closely at the deal**$_{w}ith_{p}ointers, I realize the' *' getspoppedfromthestackAFTERits$ **`this.string`**$isprinted, meaningthatevery' *' ishandledby$ **`get`**
6. `int (*pa)[];`
    - Me: **pa is pointer to array of int**
    - `cdecl.c`: **pa is pointer to array of int**
    - Explanation (if necessary): **n/a**

7. `int (*pf)();`
    - Me: **pf is pointer to function returning int**
    - `cdecl.c`: **pf is pointer to function returning int**
    - Explanation (if necessary): **n/a**
8. `char *argv[];`
    - Me: **argv is pointer to array of char**
    - `cdecl.c`: **argv is array of pointer to char**
    - Explanation (if necessary): **I expected this to behave similar to problem 3.6, except I forgot that cdecl.c addresses how parentheses affect the order of type modifier application. Here, the '*' applies to the type of the array contents, not the array itself.**
9. `int aa[][];`
    - Me: **aa is array of array of int**
    - `cdecl.c`: **aa is array of array of int**
    - Explanation (if necessary): **2D array**
10. `int af[]();`
    - Me: **ILLEGAL**
    - `cdecl.c`: **af is array of int**
    - Explanation (if necessary): **DISREGARD cdecl.c OUTPUT: The code does not acknowledge illegal array/function combinations and will only handle the first set of delimeters ('()' or '[]').**
11. `int *fp();`
    - Me: **fp is function returning pointer to int**
    - `cdecl.c`: **fp is function returning pointer to int**
    - Explanation (if necessary): **n/a**
12. `int fa()[];`
    - Me: **ILLEGAL**
    - `cdecl.c`: **fa is function returning int**
    - Explanation (if necessary): **DISREGARD cdecl.c OUTPUT: The code does not acknowledge illegal array/function combinations and will only handle the first set of delimeters ('()' or '[]').**
13. `int ff()();`

- Me: **ILLEGAL**
- `cdecl.c`: **ff is function returning int**
- Explanation (if necessary): **DISREGARD cdecl.c OUTPUT: The code does not acknowledge illegal array/function combinations and will only handle the first set of delimeters ('()' or '[]').**

14. `int ***ppp;`
    - Me: **ppp is pointer to pointer to pointer to int**
    - `cdecl.c`: **ppp is pointer to pointer to pointer to int**
    - Explanation (if necessary): **n/a**
15. `int (**ppa)[];`
    - Me: **ppa is pointer to pointer to array of int**
    - `cdecl.c`: **ppa is pointer to pointer to array of int**
    - Explanation (if necessary): **n/a**
16. `int (**ppf)();`
    - Me: **ppf is pointer to pointer to function returning int**
    - `cdecl.c`: **ppf is pointer to pointer to function returning int**
    - Explanation (if necessary): **n/a**
17. `int *(*pap)[];`
    - Me: **pap is pointer to array of pointer to int**
    - `cdecl.c`: **pap is pointer to array of pointer to int**
    - Explanation (if necessary): **One '*' applies to `pap` (the identifier) and the other applies to `int` (the array contents' type).**
18. `int (*paa)[][];`
    - Me: **paa is pointer to array of array of int**
    - `cdecl.c`: **paa is pointer to array of array of int**
    - Explanation (if necessary): **n/a**
19. `int (*paf)[]();`
    - Me: **ILLEGAL**
    - `cdecl.c`: **paf is pointer to array of int**
    - Explanation (if necessary): **DISREGARD cdecl.c OUTPUT: The code does not acknowledge illegal array/function combinations and will only handle the first set of delimeters ('()' or '[]').**

20. `int *(*pfp)();`
    - Me: **pfp is pointer to function returning pointer to int**
    - `cdecl.c`: **pfp is pointer to function returning pointer to int**
    - Explanation (if necessary): **Similar to problem 3.17 but for a function now.**
21. `int (*pfa)()[];`
    - Me: **ILLEGAL**
    - `cdecl.c`: **pfa is pointer to function returning int**
    - Explanation (if necessary): **DISREGARD cdecl.c OUTPUT: The code does not acknowledge illegal array/function combinations and will only handle the first set of delimeters ('()' or '[]').**
22. `int (*pff)()();`
    - Me: **ILLEGAL**
    - `cdecl.c`: **pff is pointer to function returning int**
    - Explanation (if necessary): **DISREGARD cdecl.c OUTPUT: The code does not acknowledge illegal array/function combinations and will only handle the first set of delimeters ('()' or '[]').**
23. `int **app[];`
    - Me: **app is array of pointer to pointer to int**
    - `cdecl.c`: **app is array of pointer to pointer to int**
    - Explanation (if necessary): **n/a**
24. `int (*apa[])[];`
    - Me: **apa is array of pointer to array of int**
    - `cdecl.c`: **apa is array of pointer to array of int**
    - Explanation (if necessary): **n/a**
25. `int (*apf[])();`
    - Me: **apf is function returning pointer to array of int**
    - `cdecl.c`: **apf is array of pointer to function returning int**
    - Explanation (if necessary): **I should have read inside the declaration parentheses first, but instead I read outside the parentheses first, essentially making the output backwards. Another hint I should have gotten is how cdecl.c always reads/prints left to right.**
26. `int *aap[][];`

- Me: **aap is array of array of pointer to int**
- `cdecl.c`: **aap is array of array of pointer to int**
- Explanation (if necessary): **n/a**

27. `int aaa[][][];`

    - Me: **aaa is array of array of array of int**
    - `cdecl.c`: **aaa is array of array of array of int**
    - Explanation (if necessary): **3D array**

28. `int aaf[][]();`

    - Me: **ILLEGAL**
    - `cdecl.c`: **aaf is array of array of int**
    - Explanation (if necessary): **DISREGARD cdecl.c OUTPUT: The code does not acknowledge illegal array/function combinations and will only handle the first set of delimeters ('()' or '[]').**

29. `int *afp[]();`

    - Me: **ILLEGAL**
    - `cdecl.c`: **afp is array of pointer to int**
    - Explanation (if necessary): **DISREGARD cdecl.c OUTPUT: The code does not acknowledge illegal array/function combinations and will only handle the first set of delimeters ('()' or '[]').**

30. `int afa[]()[];`

    - Me: **ILLEGAL**
    - `cdecl.c`: **afa is array of int**
    - Explanation (if necessary): **DISREGARD cdecl.c OUTPUT: The code does not acknowledge illegal array/function combinations and will only handle the first set of delimeters ('()' or '[]').**

31. `int aff[]()();`

    - Me: **ILLEGAL**
    - `cdecl.c`: **aff is array of int**
    - Explanation (if necessary): **DISREGARD cdecl.c OUTPUT: The code does not acknowledge illegal array/function combinations and will only handle the first set of delimeters ('()' or '[]').**

32. `int **fpp();`

    - Me: **fpp is function returning pointer to pointer to int**

- `cdecl.c`: **fpp is function returning pointer to pointer to int**
- Explanation (if necessary): **n/a**

33. `int (*fpa())[];`
    - Me: **fpp is function returning pointer to array of int**
    - `cdecl.c`: **fpa is function returning pointer to array of int**
    - Explanation (if necessary): **Functions may return a pointer to an array but not the array itself.**
34. `int (*fpf())();`
    - Me: **fpf is function returning pointer to function returning int**
    - `cdecl.c`: **fpf is function returning pointer to function returning int**
    - Explanation (if necessary): **n/a**
35. `int *fap()[];`
    - Me: **ILLEGAL**
    - `cdecl.c`: **fap is function returning pointer to int**
    - Explanation (if necessary): **DISREGARD cdecl.c OUTPUT: The code does not acknowledge illegal array/function combinations and will only handle the first set of delimeters ('()' or '[]').**
36. `int faa()[][];`
    - Me: **ILLEGAL**
    - `cdecl.c`: **faa is function returning int**
    - Explanation (if necessary): **DISREGARD cdecl.c OUTPUT: The code does not acknowledge illegal array/function combinations and will only handle the first set of delimeters ('()' or '[]').**
37. `int faf()[]();`
    - Me: **ILLEGAL**
    - `cdecl.c`: **faf is function returning int**
    - Explanation (if necessary): **DISREGARD cdecl.c OUTPUT: The code does not acknowledge illegal array/function combinations and will only handle the first set of delimeters ('()' or '[]').**
38. `int *ffp()();`
    - Me: **ILLEGAL**
    - `cdecl.c`: **ffp is function returning pointer to int**
    - Explanation (if necessary): **DISREGARD cdecl.c OUTPUT: The code does not acknowledge illegal array/function combinations and will only handle the first set of delimeters ('()' or '[]').**

Appendix

NOTE1. Read `*` as “pointer to” (always on the left side), `[ ]` as “array of” (always on the right side), and `( )` as “function returning” (always on the right side) as you encounter them in the declaration.

NOTE2. Illegal combinations include `[ ]( )` (cannot have an array of functions), `( )( )` (cannot have a function that returns a function), and `( )[ ]` (cannot have a function that returns an array).

Step 1. Find the identifier. This is your starting point. Then say to yourself, “<identifier> is.” You’ve started your declaration.

Step 2. Look at the symbols on the right of the identifier. If, say, you find `( )` there, then you know that this is the declaration for a function. So you would then have “<identifier> is function returning”. Or if you found a `[ ]` there, you would say “<identifier> is array of”. Continue right until you run out of symbols or hit a *right* parenthesis. (If you hit a left parenthesis, that’s the beginning of a `( )` symbol, even if there is stuff in between the parentheses. More on that below.)

Step 3. Look at the symbols to the left of the identifier. If it is not one of our symbols above (say, something like “int”), just say it. Otherwise, translate it into English using **NOTE1** above. Keep going left until you run out of symbols or hit a *left* parenthesis.

Now repeat steps 2 and 3 until you’ve formed your declaration.