Question 1. [8 marks]

Complete the code below according to the comments.

```
// This struct represents information about names in a database.

struct database
    char **names;


int main()
    // Declare a new struct database variable (stack-allocated).
    struct database d;


    // Allocate space for an array of three char * values on the heap. Store the address
    // of the first value into the 'names' member of the variable you just declared.
    d.names = malloc(sizeof(char *) * 3);



    // Set the first char * value to refer to the string literal "David",
    // and the second char * value to refer to the *heap-allocated* string "Michelle".
    // (Don't change the third char * value.)
    d.names[0] = "David";
    d.names[1] = malloc(strlen("Michelle") + 1);
    strcpy(d.names[1], "Michelle");
```

note strcpy copies null teriminator

```
    // Free all the dynamically-allocated memory you used in the previous parts.
    free(d.names[1]);
    free(d.names);

    return 0;
```

Question 2. [3 MARKS]

Assume you have a terminal open, and the current working directory contains a C program file called `blurb.c`.

Part (a) [1 MARK]

Write a command to compile `blurb.c` into an executable called `blurb`, including debugging symbols and using the c99 standard.

```
gcc -Wall -o blurb -g -std=c99 blurb.c
```

Part (b) [2 MARKS]

Run a command to get the names of all files in the current directory, and pipe the results as input to `blurb`. Store the output of `blurb` in a file called `output.txt`. Use `ls` without any command line arguments.

```
ls | blurb > output.txt
OR
ls | ./blurb > output.txt
```

Question 3. [3 MARKS]

For the program below, each time a variable is declared or memory is otherwise allocated, write the amount of memory that is allocated, where it is allocated, and when the memory is de-allocated. For stack memory, specify which stack frame the memory belongs to. Note: some programs allocate more than one block of memory.

Code Fragment	Amount of memory	Where?	De-allocated when?
`int fun(int *ptr) {`	sizeof (int *)	stack-fun	end of fun
`  int s = ptr[0] + ptr[1];`	sizeof (int)	stack - fun	end of fun
`  ptr = malloc(sizeof(int));`	sizeof (int)	heap	end of main/program
`  return s + 2;`			
`}`			
`int main() {`			
`  int a[3] = {2, 4, 10};`	3 * sizeof(int)	stack - main	end of main/program
`  a[2] = fun(a);`			
`  return 0;`			
`}`			

note an array is not a pointer to an array of int
a[] itself represents the array, defaults to pointing to first element in array

Question 4. [4 MARKS]

For each of the code fragments below, there is missing code. At the very least, the line (or lines) that declare and possibly initialize the variable `x` are missing. If the code will not compile no matter what you put for the missing code, check `COMPILE ERROR` and explain why. If the code will compile, but is not guaranteed to run without an error, check `RUN-TIME ERROR` and explain why. Otherwise, check `NO ERROR` and give the correct declaration for x. You don't need to show any other missing code. The first two are done for you.

Code Fragment	ERROR	Declaration for `x` or explanation

`int y;`
`// missing code`
`x = y;`
	☑`NO ERROR`
	□ `COMPILE ERROR`
	□ `RUN-TIME ERROR`
		`int x;`

`int z;`
`// missing code`
`x = *z;`
	□ `NO ERROR`
	☑`COMPILE ERROR`
	□ `RUN-TIME ERROR`
		code will not compile –
		you can't dereference an int.

`int *totals[3];`
`// missing code`
`x = *totals[0];`
	☑`NO ERROR`
	□ `COMPILE ERROR`
	□ `RUN-TIME ERROR`
		`int x;`

`struct user **user_ptr_add;`
`// missing code`
`x = *user_ptr_add;`
	☑`NO ERROR`
	□ `COMPILE ERROR`
	□ `RUN-TIME ERROR`
		`struct user *x;`

`double width;`
`double *width_ptr;`
`*width_ptr = width;`
`// missing code`
`*x = *width_ptr;`
	□ `NO ERROR`
	□ `COMPILE ERROR`
	☑`RUN-TIME ERROR`
		segmentation fault – assigning to dereferenced width_ptr, but it has no memory allocated

`char *s = "hello";`
`// missing code`
`x = *(s+3);`
	☑`NO ERROR`
	□ `COMPILE ERROR`
	□ `RUN-TIME ERROR`
		`char x;`

Question 5. [7 marks]

We define a prefix of string `s` as a substring of `s` that begins at `s[0]`. For example, "ZAP" has prefixes "", "Z", "ZA", and "ZAP". Complete the function on the next page according to its documentation given here by using `strstr` to search for longer and longer prefixes of `inner` within `outer`. Notice that the parameters are `const` - do not mutate them. See the API for an excerpt from the strstr man page.

```
/* Precondition: piece and s1 are both null-terminated strings.
 * Return the length of the longest prefix of inner that occurs anywhere in outer
 *    "ABCxCAABCDxx", "ABCDE"  should return 4 since "ABCD" is found
 *    "_123_01", "01234" should return 2 since "01" is found
 */

int longest_prefix_length(const char *outer, const char *inner) {

    int result = 0;
    // allocate space for the longest possible prefix
    char prefix[strlen(inner)+1];

    for (int i = 1; i < strlen(inner); i++) {

        // build the prefix for this try
        strcpy(prefix, inner);
        prefix[i + 1] = '\0';

        // now search for it in outer
        char *loc = strstr(outer, prefix);
        if (loc != NULL) {
            result += 1;     /* or result = i; */
        }
    }
    return result;
}
```