These are some general tasks or exercises that exploit different principles found in C programming language when used on Embedded Systems (or in general). Exercises are categorized. Here are a couple of pointers (pun intended):

- As solution provide:
  - Only .c and (if needed) .h files in a folder named: <category>_<exercise number>
  - For text answers in .txt file in a folder named <category>_<exercise number>
- Solutions should not include compiler specific macros or pragmas (except if really needed)
- Solution should be compilable with the latest GCC compiler.
- Be aware to cover edge cases, make your exercise solution more robust.
- Speed of solution making is not as important as correctness. So rather take some more time to verify that your code does what exercise demands of it.

Pointers

1. Create an **integer variable** named `myInt` and initialize it to value 25. Create a **pointer to an integer variable** named `pMyInt` and initialize it in a way it would point to the address of variable `myInt`. Change the value of myInt to 52, by using only the `pMyInt` variable. In the end verify correctness by printing out the console value of `myInt` and the value at the address stored in `pMyInt`. Both should be 52.

2. In the main function, declare an integer variable named `myInt` and initialize it to value 42. Create two additional functions:
   - Function `incrementInt` that returns nothing and takes **integer** as parameter, increments parameter by 2 and prints it out (after increment)
   - Function `incrementIntPtr` that returns nothing and takes **pointer to integer** as a parameter, increments parameter value by 2 and prints value out (after increment)

   Then execute main in such way (pseudocode):

   ```
   print("Value at beginning: " + myInt);
   incrementInt(myInt);
   print("Value after incrementInt: " + myInt);
   incrementIntPtr(pointer to myInt);
   print("Value after incrementIntPtr: " + myInt);
   ```

   (Bonus) Do you notice anything about the value of `myInt`, throughout your program? If so, can you explain why it happens?

3. (Bonus) In the C programming language different data types take up different sizes in memory to hold values. For example char and int, generally take up 1 and 4 bytes respectively in a 32-bit system.
   - Can the same be said for pointer variables (e.g. size of pointer to int vs pointer to char)?
   - How about pointer variable sizes in 32-bit or 64-bit systems, are those the same?

4. (Bonus) Does a pointer variable that points to a pointer variable that points to a data variable (double pointer) makes sense to use? If yes, name one use case.

Data types

1. Johnny has an embedded device with 4 kB of dedicated memory for storing data and variables. He is not so economic with variable types in his code and compiler says:

```
Error: 4kB of dedicated data memory is overflown by 8 bytes
```

Now he needs to optimise his code (or variable sizes), so he turns to you for help and immediately you see that he has following variables in code:

```
unsigned int dwMyCarSpeed; // my car maximum speed is 200 km/h
unsigned long qwMyCarRPM; // my car maximum RPM is 6000
```

How can you change variable types so that code will compile for Johnny (take up at least 8 bytes less of memory)?

2. Using standard C (no non-standard libraries), store the string “Today is a sunny day” in variable `myString` and print it out with `printf`.

(Bonus) How do C libraries or functions know when “string” ends? E.g. Why doesn’t `printf` just go further in memory and start printing out characters that are in memory after `myString`?

Bitwise operations

1. Create functions that will change state of single bit in parameter value:
   - `setBit`: takes two parameters: integer value and index of bit that should be set. Function returns modified value with specified bit set to 1.
   - `clearBit`: takes two parameters: integer value and index of bit that should be cleared. Function returns modified value with specified bit cleared to 0.
   - `flipBit`: takes two parameters: integer value and index of bit that should be flipped (0->1 or 1->0, based on original value). Function returns modified value with specified bit flipped.

Example of what functions should return:

```
newVal = setBit(0x100, 1); // newVal should be 0x102
newVal = clearBit(0x116, 1); // newVal should be 0x114
newVal = flipBit(0x316, 8); // newVal should be 0x216
newVal = flipBit(0x216, 8); // newVal should be 0x316
```

2. Create functions that will change state of multiple bits in parameter value that are set in mask parameter:
   - `setBits`: takes two parameters: integer value and mask where bits with value 1 should be set in passed value. Function returns modified value.
   - `clearBits`: takes two parameters: integer value and mask where bits with value 1 should be cleared in passed value. Function returns modified value.
   - `flipBits`: takes two parameters: integer value and mask where bits with value 1 should be flipped (0->1 or 1->0, based on original value). Function returns modified value.

Example of what functions should return:

```
newVal = setBits(0x100, 0x101); // newVal should be 0x101
newVal = clearBits(0x116, 0x022); // newVal should be 0x114
newVal = flipBits(0x316, 0x111); // newVal should be 0x207
```

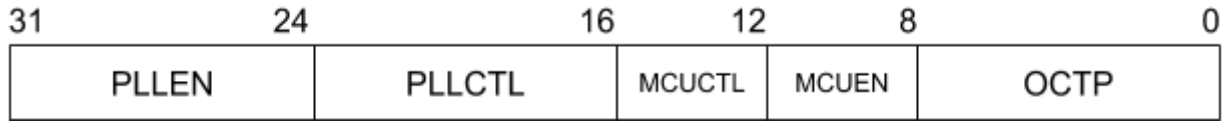
Structures

1. You and your friends Don and Glenn have an argument about which of your three cars is the best. To settle this, you decide to use your programming skills, so: create a struct(ure) named `carInstance`, which has following structure members:
   - `maxSpeed`: which is used to store car's maximum speed
   - `maxRPM`: which is used to store car's maximum RPM
   - `torque`: which is used to store car's torque
   - `horsepower`: which is used to store car's horsepower
   - `numberOfGears`: : which is used to store car's number of gears

   Then in code create three instances of carInstance, for you and each of your friends. Create then function(s) that would print to console, which of the three cars is best for a specific category (struct member).

   (Bonus) Create a solution that settles disputes between different numbers of friends, not just three (by only adding new structure instances in code).

2. (Bonus) We have an embedded system with 32-bit register SYS_CTRL, that has following structure:



   So, to change:
   - OCTP bits, we need to change bits 0-7 in SYS_CTRL register
   - MCUCTL bits, we need to change bits 8-13 in SYS_CTRL register
   - …

   How would you define a structure in C that would change exactly and only MCUCTL bits when one would write following code:

```
SYS_CTRL.MCUCTL = 0x1;
```

Project with multiple source files

1. Create two .c files:
   - One that contains function `printParameter` (with one integer parameter), that outputs passed integer parameter to console:

     ```
     I printParameter, was just called with parameter <parameter value>
     ```

   - One that contains the main function with calls to the `printParameter` function from the other file.