Week 4 Problem Set
Dynamic Data Structures

1. (Pointers)

   Given the following definition:

   ```
   int data[12] = {5, 3, 6, 2, 7, 4, 9, 1, 8};
   ```

   and assuming that *&data[0] == 0x10000*, what are the values of the following expressions?

data + 4
*data + 4
*(data + 4)
data[4]
*(data + *(data + 3))
data[data[2]]

2. (Pointers)

   Consider the following piece of code:

   ```
   typedef struct {
      int   studentID;
      int   age;
      char  gender;
      float WAM;
   } PersonT;

   PersonT per1;
   PersonT per2;
   PersonT *ptr;

   ptr = &per1;
   per1.studentID = 3141592;
   ptr->gender = 'M';
   ptr = &per2;
   ptr->studentID = 2718281;
   ptr->gender = 'F';
   per1.age = 25;
   per2.age = 24;
   ptr = &per1;
   per2.WAM = 86.0;
   ptr->WAM = 72.625;
   ```

   What are the values of the fields in the *per1* and *per2* record after execution of the above statements?

3. (Memory management)

Consider the following function:

```
/* Makes an array of 10 integers and returns a pointer to it */

int *makeArrayOfInts() {
   int arr[10];
   int i;
   for (i=0; i<10; i++) {
      arr[i] = i;
   }
   return arr;
}
```

Explain what is wrong with this function. Rewrite the function so that it correctly achieves the intended result using `malloc()`.

4. (Memory management)

   Consider the following program:

```
#include <stdio.h>
#include <stdlib.h>
#include <assert.h>

void func(int *a) {
   a = malloc(sizeof(int));
   assert(a != NULL);
}

int main(void) {
   int *p;
   func(p);
   *p = 6;
   printf("%d\n",*p);
   free(p);
   return 0;
}
```

   Explain what is wrong with this program.

5. (Dynamic arrays)

   Write a C-program that

   - takes 1 command line argument, a positive integer *n*
   - creates a dynamic array of *n* `unsigned long long int` numbers (8 bytes, only positive numbers)
   - uses the array to compute the *n*'th Fibonacci number.

   For example, **`./fib 60`** should result in 1548008755920.

   Hint: The placeholder `%llu` (instead of `%d`) can be used to print an `unsigned long long int`. Recall that the Fibonacci numbers are defined as Fib(1) = 1, Fib(2) = 1 and Fib(*n*) = Fib(*n*-1)+Fib(*n*-2) for *n*≥3.

6. (Dynamic linked lists)

   Write a C-program called **llbuild.c** that builds a linked list of integers from user input. Your program should use the following functions:

- *NodeT *makeNode(int value)*: taken from the lecture
- *void freeLL(NodeT *list)*: taken from the lecture
- *void showLL(NodeT *list)*: taken from the lecture **but needs modification**
- *NodeT *joinLL(NodeT *list1, NodeT *list2)*: appends linked *list2* to *list1*. **Needs to be implemented.**

The program:

- starts with an empty linked list called *all* (say), initialised to `NULL`
- prompts the user with the message "Enter a number: "
- makes a linked list node called *new* from user's response
- appends *new* to *all*
- asks for more user input and repeats the cycle
- the cycle is terminated when the user enters any non-numeric character
- on termination, the program generates the message "Done. List is " followed by the contents of the linked list in the format shown below.

A sample interaction is:

```
prompt$ ./llbuild
Enter an integer: 12
Enter an integer: 34
Enter an integer: 56
Enter an integer: quit
Done. List is 12->34->56
```

Note that any non-numeric data 'finishes' the interaction. If the user provides no data, then no list should be output:

```
prompt$ ./llbuild
Enter an integer: #
Done.
```

We have created a script that can automatically test your program. To run this test you can execute the `dryrun` *program that corresponds to this exercise. It expects to find a program named* `llbuild.c` *in the current directory. You can use autotest as follows:*

```
prompt$ 9024 dryrun llbuild
```

Note: Please ensure that your output follows exactly the format shown above.

7. (Dynamic linked lists)

   Extend the C-program from the previous exercise to split the linked list in two halves and output the result. If the list has an odd number of elements, then the first list should contain one more element than the second.

   Note that:
   - your algorithm should be 'in-place' (so you are not permitted to create a second linked list or use some other data structure such as an array);
   - you should not traverse the list more than once (e.g. to count the number of elements and then restart from the beginning).

   An example of the program executing could be

```
prompt$ ./llsplit
Enter an integer: 421
Enter an integer: 456732
Enter an integer: 321
Enter an integer: 4
Enter an integer: 86
Enter an integer: 89342
Enter an integer: 9
Enter an integer: #
Done. List is 421->456732->321->4->86->89342->9
First part is 421->456732->321->4
Second part is 86->89342->9
```

To test your program you can execute the `dryrun` *program that corresponds to this exercise. It expects to find a program named* `llsplit.c` *in the current directory. You can use autotest as follows:*

```
prompt$ 9024 dryrun llsplit
```

Note: Please ensure that your output follows exactly the format shown above.

8. **Challenge Exercise**

   Write a C-program that takes 1 command line argument and prints all its *prefixes* in decreasing order of length.

   - You are not permitted to use any library functions other than `printf()`.
   - You are not permitted to use any array other than `argv[]`.

   An example of the program executing could be

```
prompt$ ./prefixes Programming
Programming
Programmin
Programmi
Programm
Program
Progra
Progr
Prog
Pro
Pr
P
```

Assessment

Group submissions will not be allowed. Your program must be entirely your own work. Plagiarism detection software will be used to compare all submissions pairwise.

Submit your solutions to **Exercise 6 and 7** using the following **give** command:

```
prompt$ give cs9024 week4 llbuild.c llsplit.c
```

Make sure you spell the filenames correctly. You can run **give** multiple times. Only your last submission will be marked.

The deadline for submission is **Monday, 14 October 11:00:00am**.

Each program is worth 1 mark. Total marks = 2. Auto-marking will be run by the lecturer several days after the submission deadline using different test cases than `dryrun` does.

Hints:

- Programs will not be manually marked.
- It is important that the output of your program follows exactly the format shown above, otherwise auto-marking will result in 0 marks.

- Ensure that your program compiles on a CSE machine with the standard options `-Wall -Werror -std=c11`. Programs that do not compile will receive 0 marks.
- `dryrun` and auto-marking also check the correct usage of dynamic memory allocation and pointers as well as the absence of memory leaks.
- Do your own testing in addition to running `dryrun`.