Week 3 Problem Set
Dynamic Data Structures

1. (Memory)

   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]]

   Answer:

data + 4	== 0x10000 + 4 * 4 bytes == 0x10010
*data + 4	== data[0] + 4 == 5 + 4 == 9
*(data + 4)	== data[4] == 7
data[4]	== 7
*(data + *(data + 3))	== *(data + data[3]) == *(data + 2) == data[2] == 6
data[data[2]]	== data[6] == 9

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?

   Answer:

per1.studentID	== 3141592
per1.age	== 25
per1.gender	== 'M'
per1.WAM	== 72.625
per2.studentID	== 2718281
per2.age	== 24
per2.gender	== 'F'
per2.WAM	== 86.0

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()`.

Answer:

The function is erroneous because the array `arr` will cease to exist after the line `return arr`, since `arr` is local to this function and gets destroyed once the function returns. So the caller will get a pointer to something that doesn't exist anymore, and you will start to see garbage, segmentation faults, and other errors.

Arrays created with `malloc()` are stored in a separate place in memory, the heap, which ensures they live on indefinitely until you free them yourself.

The correctly implemented function is as follows:

```
int *makeArrayOfInts() {
   int *arr = malloc(sizeof(int) * 10);
   assert(arr != NULL);  // always check that memory allocation was successful
   int i;
   for (i=0; i<10; i++) {
      arr[i] = i;
   }
   return arr;           // this is fine because the array itself will live on
}
```

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.

Answer:

The program is not valid because `func()` makes a *copy* of the pointer `p`. So when `malloc()` is called, the result is assigned to the copied pointer rather than to `p`. Pointer `p` itself is pointing to random memory (e.g., `0x0000`) before and after the function call. Hence, when you dereference it, the program will (likely) crash.

If you want to use a function to add memory to a pointer, then you need to pass the *address* of the pointer (i.e. a pointer to a pointer, or "double pointer"):

```
#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;
}
```

Note also that you should always ensure that `malloc()` did not return `NULL` before you proceed.

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.

An example of the program executing could be

```
prompt$ ./fib 60
1548008755920
```

If the commad line argument is missing, then the output to `stderr` should be

```
prompt$ ./fib
Usage: ./fib number
```

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* `fib.c` *in the current directory. You can use dryrun as follows:*

```
prompt$ 9024 dryrun fib
```

Answer:

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

int main(int argc, char *argv[]) {
   if (argc != 2) {
      fprintf(stderr, "Usage: %s number\n", argv[0]);
      return 1;
   }

   int n = atoi(argv[1]);
   if (n > 0) {
      unsigned long long int *arr = malloc(n * sizeof(unsigned long long int));
      assert(arr != NULL);
      arr[0] = 1;
      arr[1] = 1;
      int i;
      for (i = 2; i < n; i++) {
         arr[i] = arr[i-1] + arr[i-2];
      }
      printf("%llu\n", arr[n-1]);
      free(arr);                    // don't forget to free the array
   }
   return 0;
}
```

6. **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
```

Answer:

```
#include <stdio.h>

int main(int argc, char *argv[]) {
   char *start, *end;

   if (argc == 2) {
      start = argv[1];
      end = argv[1];
      while (*end != '\0') {    // find address of terminating '\0'
         end++;
      }
      while (start != end) {
         printf("%s\n", start); // print string from start to '\0'
         end--;                 // move end pointer up
         *end = '\0';           // overwrite last char by '\0'
      }
   }
   return 0;
}
```