COMP9024 20T0 Data Structures and Algorithms

Week 2a 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]]

   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

$n \geq 3$.

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. (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 dryrun as follows:*

```
prompt$ 9024 dryrun llbuild
```

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

Answer:

```
// llbuild.c: create a linked list from user input, and print
#include <stdio.h>
#include <stdlib.h>
#include <assert.h>

typedef struct node {
   int data;
   struct node *next;
} NodeT;

NodeT *joinLL(NodeT *list1, NodeT *list2) {
// either or both list1 and list2 may be NULL
   if (list1 == NULL) {
      list1 = list2;
   } else {
      NodeT *p = list1;
      while (p->next != NULL) {
         p = p->next;
      }
      p->next = list2; // this does nothing if list2 == NULL
   }
   return list1;
}

NodeT *makeNode(int v) {
   NodeT *new = malloc(sizeof(NodeT));
   assert(new != NULL);
   new->data = v;
   new->next = NULL;
   return new;
}

void showLL(NodeT *list) {
   NodeT *p;
   for (p = list; p != NULL; p = p->next) {
      printf("%d", p->data);
      if (p->next != NULL)
         printf("->");
   }
   putchar('\n');
}

void freeLL(NodeT *list) {
    NodeT *p = list;
    while (p != NULL) {
        NodeT *temp = p->next;
        free(p);
        p = temp;
    }
}

int main(void) {
   NodeT *all = NULL;
   int data;

   printf("Enter an integer: ");
```

```
    while (scanf("%d", &data) == 1) {
        NodeT *new = makeNode(data);
        all = joinLL(all, new);
        printf("Enter an integer: ");
    }
    if (all != NULL) {
        printf("Done. List is ");
        showLL(all);
        freeLL(all);
    } else {
        printf("Done.\n");
    }
    return 0;
}
```

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 dryrun as follows:*

```
prompt$ 9024 dryrun llsplit
```

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

Answer:

The following C-function implements the solution to Exercise 7, Week 3 Problem Set.

```
NodeT *splitList(NodeT *list) {  // returns pointer to second half
    assert(list != NULL);

    NodeT *slow, *fast;

    slow = list;
    fast = list->next;
    while (fast != NULL && fast->next != NULL) {
        slow = slow->next;
        fast = fast->next->next;
    }
    fast = slow->next;            // this becomes head of second half
    slow->next = NULL;            // cut off at end of first half
```

```
    return fast;
}
```

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
```

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;
}
```