COMP9024 24T1

Week 4 Tutorial

Dynamic Data Structures

Data Structures and Algorithms

[Show with no answers] [Show with all answers]

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]]

[hide 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)

   a. Consider the following piece of code:

```
int a = 5;                  // line 1
int b = 123;                // line 2

int *pa = &a;               // line 4
int *pb = &b;               // line 5

*pa = 6;                    // line 7
*pb = 234;                  // line 8

int c = *pa;                // line 10
*pa = *pb;                  // line 11
*pb = c;                    // line 12

pa = pb;                    // line 14
*pa = 345;                  // line 15
```

   What is the state of memory after each line of code is executed?

   b. Consider the following piece of code:

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

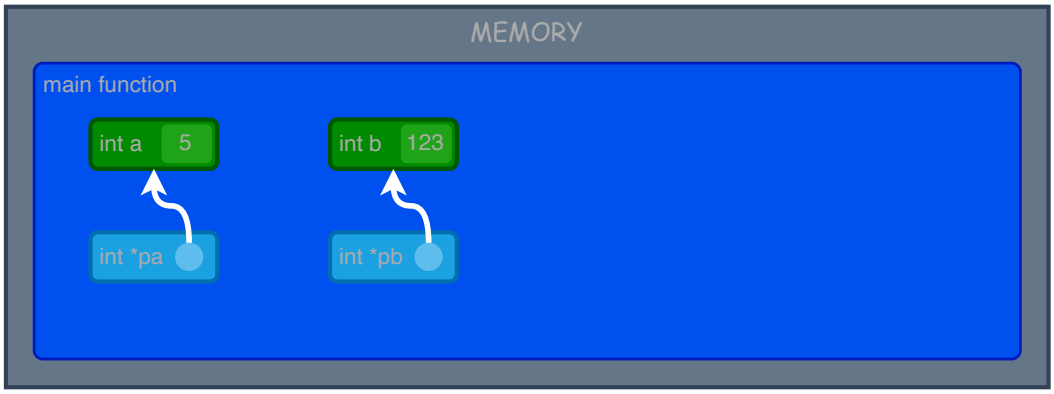
```
PersonT per1;
PersonT per2;
PersonT *ptr;

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?

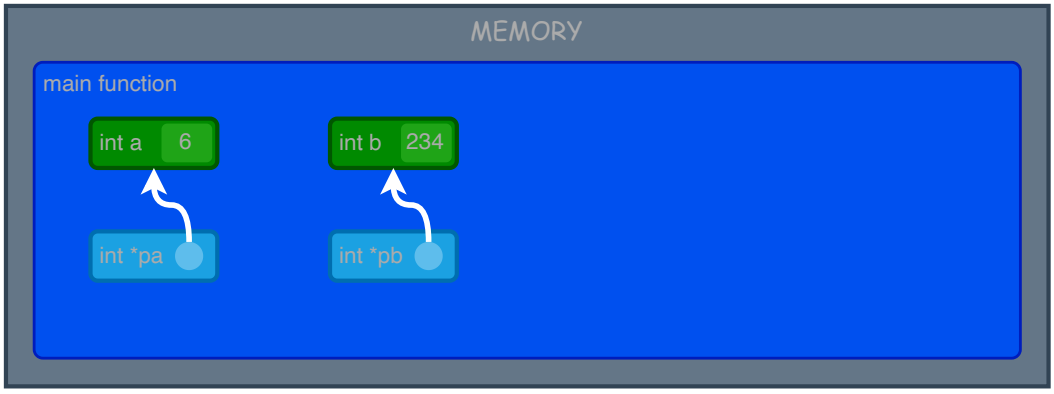
[hide answer]

a. The state of memory after line 2:

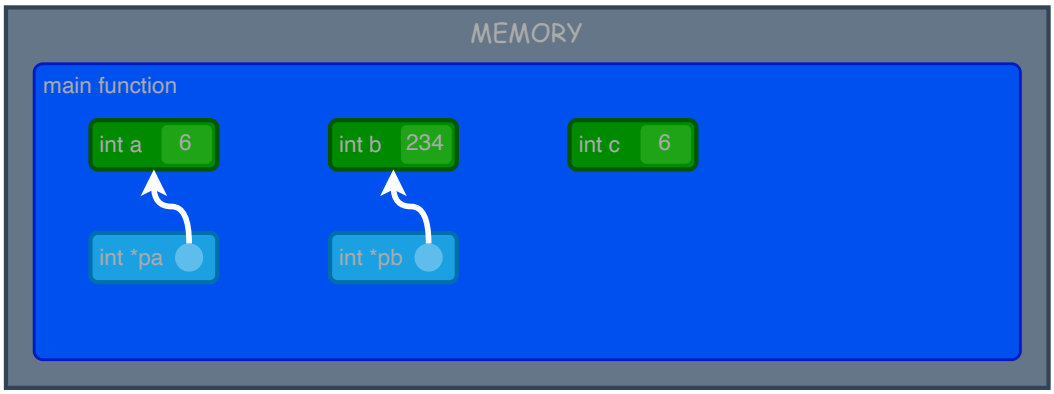
The state of memory after line 5:



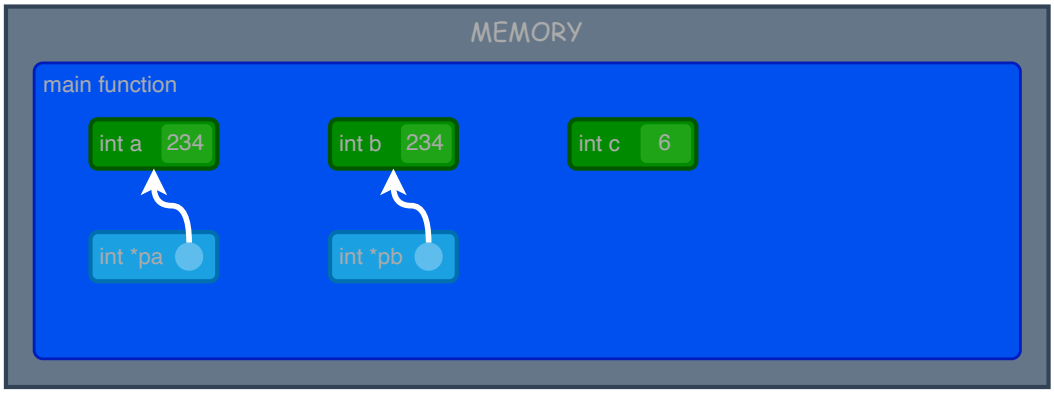
The state of memory after line 8:



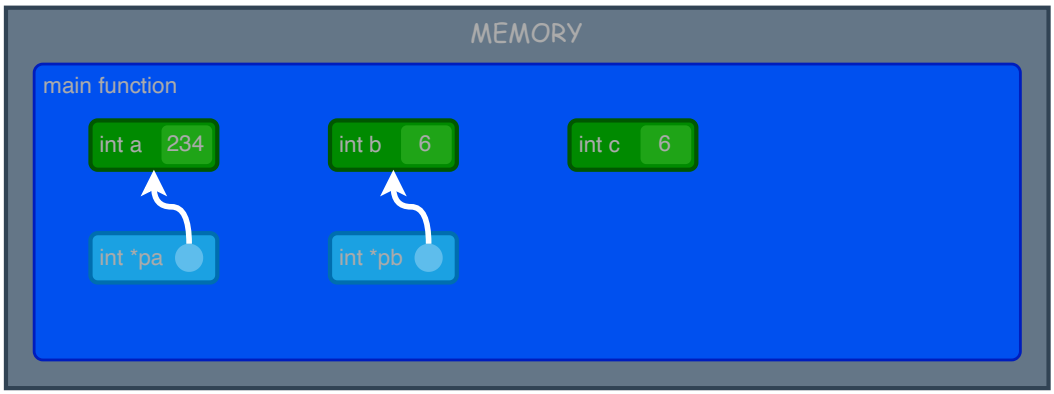
The state of memory after line 10:



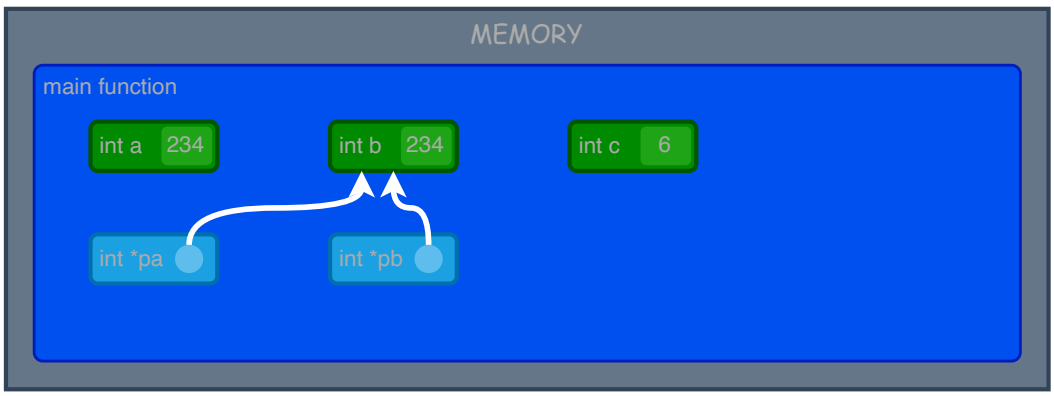
The state of memory after line 11:



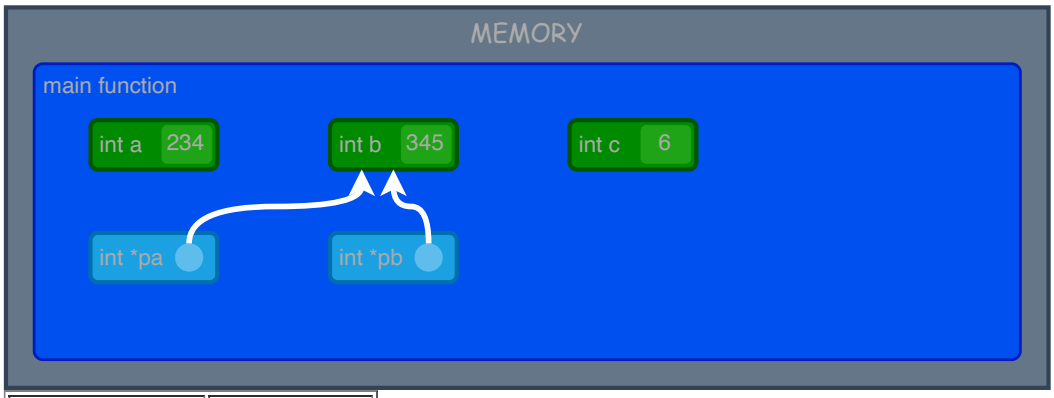
The state of memory after line 12:



The state of memory after line 14:



The state of memory after line 15:



b.

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)

a. Consider the following function:

```
/* Makes an array of 1,000 integers and returns a pointer to it */

#define SIZE 1000

int *makeArrayOfInts() {
   int arr[SIZE];
   int i;
   for (i=0; i<SIZE; 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()`.

b. 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.

c. Modify the code below so that it allocates the struct on the heap, instead of the stack.

```
struct node {
   int value;
   struct node *next;
};

int main(void) {
   struct node n;
   n.value = 42;
   n.next = NULL;
}
```

[hide answer]

a. 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:

```
#define SIZE 1000

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

b. 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.

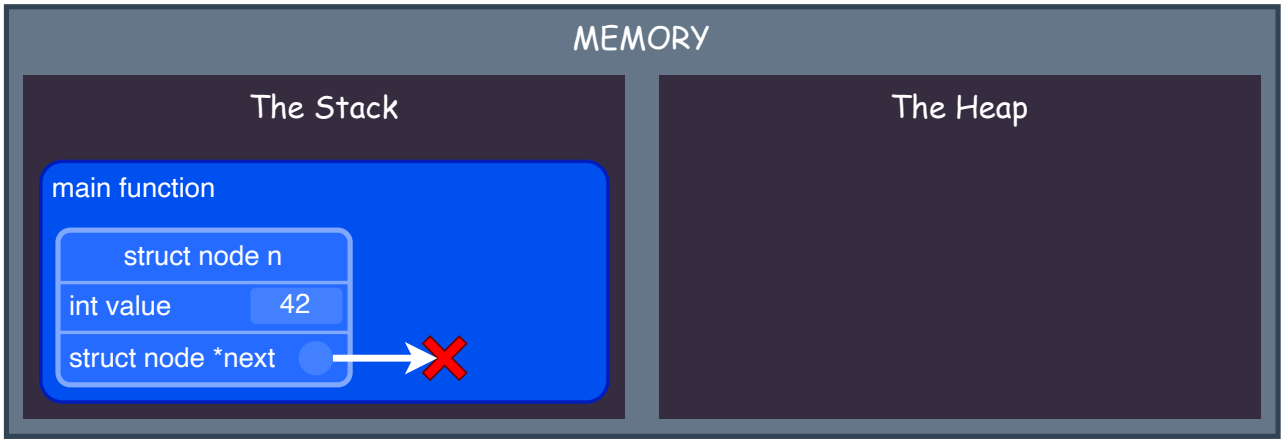
If you want to use a function to add memory to a pointer, then you need to pass the *address* of the pointer (i.e. a pointer to a pointer, or "double pointer"):

```
#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;
}
```

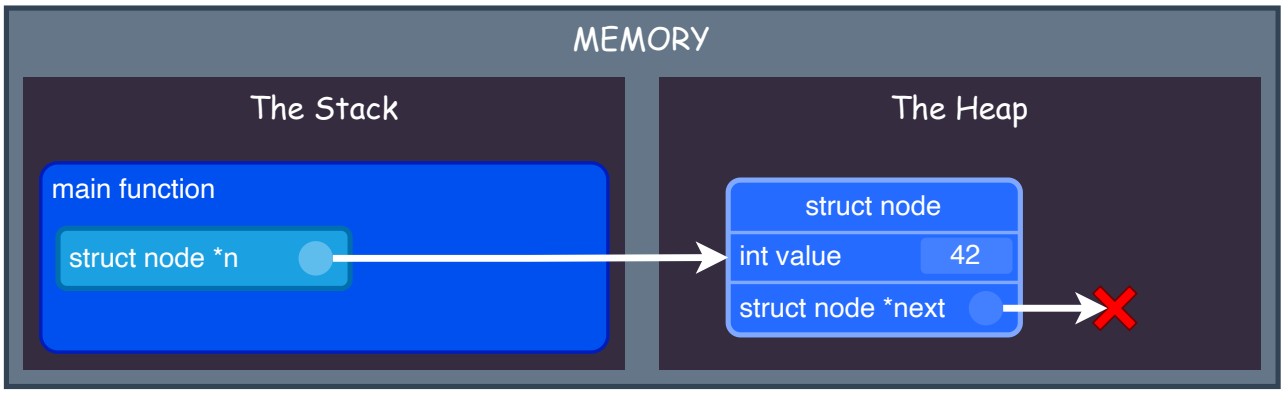
c. Remember that to access struct fields through a pointer, you need to use the arrow operator (–>).

```
struct node {
   int value;
   struct node *next;
};

int main(void) {
   struct node *n = malloc(sizeof(struct node));
   n->value = 42;
   n->next = NULL;
}
```

This is what allocating the node struct on the stack looks like:



This is what allocating the node struct on the heap looks like:



4. (Linked lists)

a. Consider the following two linked list representations:

```
// Representation 1

struct node {
   int value;
   struct node *next;
};



int listLength(struct node *list);
```

```
// Representation 2

struct node {
   int value;
   struct node *next;
};

struct list {
   struct node *head;
};

int listLength(struct list *list);
```

i. Compare the two representations diagramatically.
ii. How is an empty list represented in each case?
iii. What are the advantages of having a separate list struct as in Representation 2?

b. Consider the following simple linked list representation:

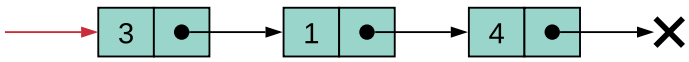
```
struct node {
   int value;
   struct node *next;
};
```

Write a function to sum the values in the list. Implement it first using `while` and then using `for`.

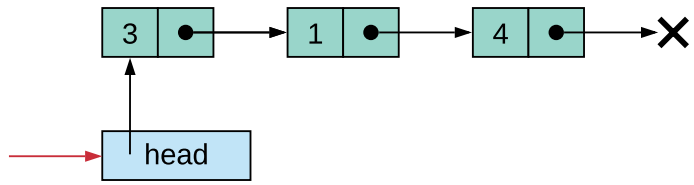
[hide answer]

a.

i. Representation 1:

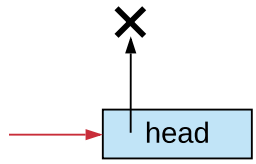


Representation 2:



ii. Representation 1: An empty list is simply represented by a NULL pointer.

Representation 2: An empty list is represented by a pointer to a `struct list` that contains a NULL pointer in its `head` field.



iii. Some advantages:

- Functions that modify the list don't need to return the updated list.
- The list struct can be used to store additional information about the list (metadata), which can improve performance. For example, if the list struct contained a field for the length of the list, a function that gets the length of the list could simply return this value instead of iterating through the entire list. If the list struct contained a pointer to the last node of the list (usually called `last` or `tail`), then this pointer can be used to easily insert items at the end of the list. The tradeoff is that these fields need to be constantly updated to be consistent with the list.

b. Version using `while`:

```
int sumList(struct node *list) {
   struct node *curr = list;
   int sum = 0;
   while (curr != NULL) {
      sum += curr->value;
      curr = curr->next;
   }
   return sum;
}
```

Version using `for`:

```
int sumList(struct node *list) {
   int sum = 0;
   for (struct node *curr = list; curr != NULL; curr = curr->next) {
      sum += curr->value;
   }
   return sum;
}
```

Challenge Exercises

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 command 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
```

[hide 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 > 2) {
      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
   } else if (n > 0) {
      printf("1\n");
   }
   return 0;
}
```

6. (Command-line arguments)

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
```

[hide 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;
}
```