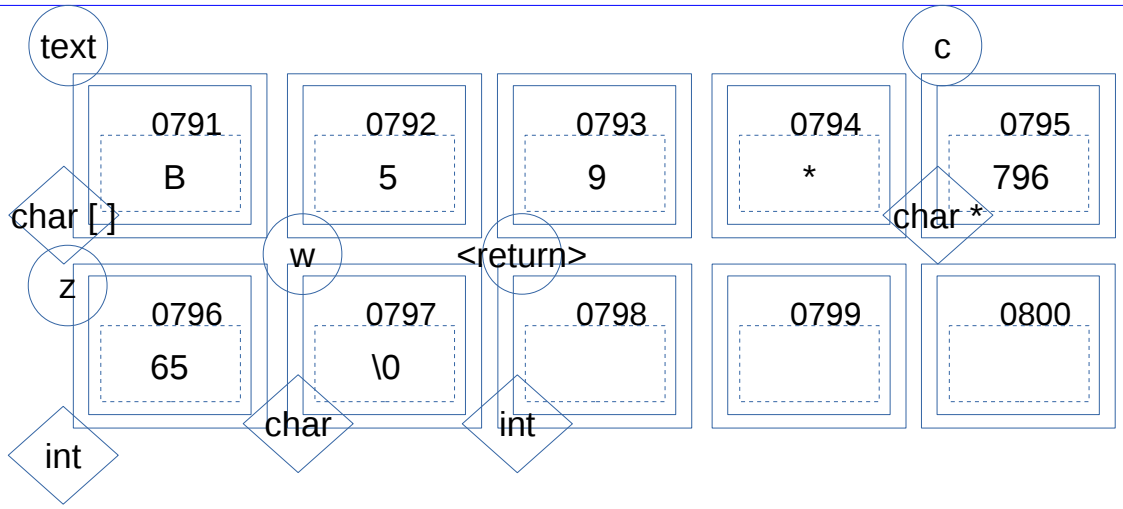
Unit 1&2 – Memory model and C programming

1.- [5 marks] Consider carefully the following program and show in the memory diagram below ***what variables are allocated*** and ***their final value (after the program is run)***. Assume variables are assigned to boxes ***in the order they are declared***.

```
int main()
{
      char text[4]={'A','4','8','\0'};
      char *c=NULL;
      int z=35;
      char w;

      c=&w;
      *(c)=text[3];
      c=&text[1];
      *(c-1)='B';
      c=c+1;
      *(c-1)='5';
      *(c)=*(c)+1;
      *(c+1)='*';                          // ← a) in question 2
      c=&z;                                // ← b) in question 2
      *(c)=65;                             // ← c) in question 2
      // d) – f) in question 2 would happen after this point
      // in the program... don't change the values in your boxes
      // because of d) – f) in question 2!
}
```



2.- [3 marks] Circle in the space below ***any instructions that will likely cause a segfault (program crash)*** given the code from 1). You get .5 for each correctly circled answer, and .5 for each correctly not-circled answer! Assume the instructions below are ***independent from each other*** (not carried out in sequence!).

a) (as shown above)

b) (as shown above)

c) (as shown above);

d) printf("%s\n",text);

e) free(c);

f) *(c-6)='\0';

Unit 3 – Organizing, Storing, and Accessing Data

For the ***Computer Graphics*** course ***CSCD18***, we keep a linked list of the objects for rendering. Each object contains a variety of data, so we use a CDT to keep track of everything. The structures below show a simple object description.

```
typedef struct obj_struct
{
     double Pos[3];               // Position [x,y,z] of this object in the scene
     double Col[3];               // RGB colour of the object
     int type;                    // Shape type, 0=circle, 1=triangle, 2=square
} Object;
```

Beside simple objects, we usually create compound objects made up of collections of simpler ones (for example, a snow-man can be constructed with a bunch of circles). The structure below can be used to define compound objects with up to 10 simple shapes – the actual shapes are ***dynamically allocated*** – i.e. we reserve memory only when we need to, and use pointers to figure out where they are stored. Just like we've been doing in class.

```
typedef struct comp_obj_struct
{
     int n_objects;          // How many shapes are in this object (n<=10)
     Object *objects[10];    // Pointers to simple objects that make up this shape
} CompObj;
```

3.- [5 marks] Complete the code below. The code declares a compound object, and we want the first shape in the compound object to be ***a circle,*** at ***location [3, 5, -2].*** Do not worry about the colour for this circle at this point.

```
int main()
{
     CompObj snow_man;

     // Write the code needed to initialize snow_man to contain 1 simple shape,
     // and set up that shape as described above. This should take no more than
     // 6 lines of code not counting curly braces.

     __ snow_man.objects[0]=(Object *)calloc(1,sizeof(Object)); _______

     __ if (snow_man.objects[0]!=NULL){

     __ snow_man.objects[0]->type=0;  __________

     __ snow_man.objects[0]->Pos[0]=3;   ______

     __ snow_man.objects[0]->Pos[1]=5;   _____

     __ snow_man.objects[0]->Pos[2]=-2; ________

     }
}
```

4.- [2 marks] What would you do to enable the C_Object CDT to support compound shapes with any number of simple shapes in them?

The compound objects should have a pointer to a linked list of shapes, or a dynamic array (as in dynamically re-sized). Either way, the number of objects in the compound entity can be anything we like.

Unit 3 – Linked Lists

Suppose our rendering software has a single linked list of ***simple objects*** that it needs to render on screen. That means that the components of any ***compound object*** have to be inserted into this linked list in order for compound objects to be drawn. The linked list uses the following node structure:

```
typedef struct obj_node_struct
{
      Object *obj;                  // Note that this is a pointer! (why?)
      struct object_node_struct *next;

} ObjectNode;
```

5.- [5 marks] Complete the code below. It implements the insert function for our linked list of objects. The order of objects in the linked list is ***NOT*** relevant.

```
ObjectNode  *insertObject(ObjectNode *head, Object *obj)
{
      ObjectNode *node=NULL;

      // Inserts the object whose location is given by the pointer 'obj' into
      // the linked list at 'head'. As usual, head==NULL if the list is empty.
      // This requires no more than 5 lines of code.

      node=(ObjectNode *)calloc(1,sizeof(ObjectNode);

      if (p==NULL) return head;

      node->obj=obj;

      node->next=head;        // Insert at head, old head becomes 2nd node

      return node;            // New node becomes the head
}
```

Unit 3 – Linked Lists (continued)

6.- [3 marks] Write a function below to insert a ***compound object*** into the linked list of objects using the insert function you completed above

```
ObjectNode *insertCompoundObject(ObjectNode *head, CompObj *comp_obj)
{
      // Add code here to implement this function. It should take no more than
      // 3 lines of code. This function is called for every compound object!

      __ for (int i=0; i<comp_obj->n_objects; i++) ____

      __    head=insertObject(head, comp_obj->objects[i]);

      __ return head; _______

}
```

7.- [5 marks] Of course, an important task when dealing with dynamic data structures is to clean up after we're done. Complete the function below to release ***ALL*** the memory assigned to the objects in the linked list (this means the objects themselves too).

```
void cleanUp(ObjectNode *head)
{
      ObjectNode *p, *q;

      // Add code here to implement this function. It should take no more than
      // 6 lines of code not counting curly braces.

      __ p = head; ____

      __ while(p!=NULL){  __________

      __   q=p->next;     __

      __   free(p->obj);  _________

      ___  free(p);  _____

      ___  p=q;  _____

      }
}
```

8.- [1 mark] After we call the cleanUp function above, should we call a separate function to release the memory allocated to compound objects? (answer ***Yes or No***, and ***give a brief explanation of your reasoning)***.

No – the objects themselves are already freed by the cleanup function

Unit 4 – Complexity of list operations

For each of the questions below: ***Give a complexity estimate in terms of big O, and briefly justify your answer***

9.- [2 marks] The rendering process requires shapes to be drawn from back to front (i.e. shapes more distant from the observer are drawn first, the closest ones are drawn last). So the list has to be sorted in order of distance. If you write a function to ***sort the linked list of objects,*** and once they are sorted, they have to be drawn on screen. You decide the best way to do this is to go through the list, and insert each object in the list into a ***new list that is kept in order***. That means finding where the object goes. ***What is the Big O worst-case complexity of inserting a single object into this sorted list?***

In the worst case, we have to go through a list of length N-1 to find the right spot, so
O(N)

10.- [2 marks] Given your answer for 9), what is the ***worst-case Big O complexity of sorting the whole list?***

Because we have to do an O(N) traversal N times, we get
$O(N^2)$

11.- [4 marks] To make things more ***interesting*** we have to handle animation – things move around, and their distance from the camera can change! After the objects move, the list is no longer sorted. Assume that at most ***k*** objects have moved (with ***k << N***). What is the ***worst-case Big O complexity for re-sorting the list***? Here you should think about what the best way to handle this situation would be.
Describe your proposed method for re-sorting the list, and explain ***briefly*** your complexity estimate.

For each of the K objects that moved:
    Find object in current list and unlink (remove it) O(N)
    Insert it into a temporary linked list at head O(k)

This is O(k*N) → O(N)

For each entry in the temporary list
    Insert the object in sorted order into the sorted list with the remaining objects O(N)

This is also O(k*N) → O(N)

So overall the process is only O(N) – much better than re-sorting!

Applying what we've learned

12.- [3 marks] For A1, you implemented a movie database that allowed you to keep track of the cast for each of the movies stored in it. You also implemented several common queries that directly involved looking at data within the movie's compound data type. However, typical databases have to answer fairly complex queries that may require looking at information stored in multiple different locations.

Suppose we want to provide a function to ***return the box office total for all movies whose cast includes a specific cast member.*** Given the structure of the movie database you built for A1, assuming there are ***N*** movies, and that each movie's cast list can have up to ***M*** cast members, ***what is the worst-case complexity of answering this query?*** (Consider here only the number of times cast-member entries have to be searched looking for the query's requested cast member, do not worry about the operations required to sum up the box office total!) ***Give your worst-case estimate in terms of BigO notation, and make sure to briefly explain how you arrived at this estimate.***

We have to look at each movie – That's N movies
For each movie, we have to potentially look at the entire cast list with M entries
So this is O(M*N)

13.- [5 marks] The current structure of our database does not help much in terms of answering the query described above. This is a common problem, and the solution to it is to build an ***index***. This is just an auxiliary data structure that makes it faster for us to answer a specific query. In the case above, it looks like we should have an ***index on cast-member name*** (and for now let's assume those are unique!).

Given what you have learned up to this point: Design and implement a data structure (you choose which one) to store information that will allow us to answer the same queries as in (12), but much more efficiently. Assume each cast member only participates in a small number ***k*** of movies, where ***k<<N***.
DO NOT ADD CODE HERE: Explain and illustrate your design, state clearly what type of data structure you're using, what information is stored in each entry of your data structure (e.g. if it' a CDT, list the fields), and the complexity of answering the queries using your design. Your data structure must be ***dynamic*** – no pre-set size.

Build a linked list by cast-member name.

Inside each node, store a linked list of movies the cast-member has acted in

This means that in order to answer a query we now have to traverse a list of length M,
Once we found the right cast-member, we have the list of k movies they played in.

O(M+k) since M<N and k<<N, this is much better than the original one.

Alternately: Use an array of cast-members (sorted by cast member name), each entry has a ***pointer*** to the linked list of movies they played in.

It takes O(LogM) time to find the right cast member by binary search, so the process becomes O(k Log M).

However, this requires an O(N) operation to insert new cast members, and if needed resize the array

Multiple Choice Questions

Consider all options carefully, ***and don't forget to record your answers on the bubble sheet – otherwise you will receive a zero. Only record your final answer for each question.***

1.- Which of the following ***for loop declarations*** will ***not compile*** (because it's not a valid thing to do!)
Assume 'string' has been declared as *char string[1024];*

a) *for (int x=100; 2*x>2; x=x-1);*

b) *for (char *p=&string[0]; *(p)!='\0'; p++);*

c) *for (double y=2*3.141592; y>=0; y=y-.01);*

d) *for (string="A"; string<"B"; string++);*

e) *for (string[0]='A'; string[0]<'Z'; string[0]++);*

2.- All of the things ***except for one*** are things we can do with variables in C. Select the one that is the exception to the rest.

a) We can pass them to functions

b) We can change their data type using ***casting***

c) We can change their value using pointers

d) We can printf() them with a different ***data type*** than declared

e) We can use them in computations involving different compatible data types

---

3.- Which of the following statements regarding ***static arrays*** in C is not correct?

a) We can declare an array of any data type including CDTs.

b) Array indexing starts at 0 and goes up to array_size-1

c) We can not return a ***static array*** from a function that defined it

d) When we pass an array to a function, the function gets a copy just like with any other variable

e) Array entries are ***always*** contiguous in memory

4.- From the ***Unit 2*** notes and our class discussions, we know that ***data types exist because*** (select the option that ***does not apply***)

a) Because data in memory is in binary

b) Because the CPU has different circuits for each data type

c) To make us think carefully about what data we need to handle, and how it will be used

d) So the compiler can do automatic type conversion

e) To allow for checking the consistency of data manipulation in our code

5.- At the end of the ***Unit 2 Notes,*** there is an exercise that consisted of starting a string about a blue rhinoceros named Randolph. The code there has a major problem, and this was brought up on Piazza, so you should have looked into it. The reason the exercise code would not work is:

a) The function returned a pointer to its local string array

b) The function did not use the library function *strcpy()* correctly

c) It declared a string that was too small to hold the story

d) The function could not be called twice

e) The function was trying to access a string from main() that it did not have a pointer to

---

6.- In Assignment 1, you worked with a compound data type for movies, then you built a different compound data type for a linked list of movie reviews which contained a movie review CDT and a pointer. Below are several reasons why we choose to have the separate CDT for the linked list instead of just adding a pointer to the movie CDT – ***however one of these reasons does not apply – which one?*** (think about this, it was not explicitly stated in class or in the notes but if you think carefully about it, one of them doesn't really make much sense).

a) Having a separate CDT for the linked list makes it easier to re-purpose our linked-list code (i.e. it's easier to adapt the list to work with a different data type or CDT)

b) The movie CDT may be used in other data structures, not just a linked list

c) It's a good idea to make the definition of the ***data*** separate from the definition of its ***container***

d) Movie CDT variables and arrays take up less space without the pointer

e) It makes accessing information within the list easier by using pointers

---

7.- From the point of view of the amount of time it takes to access information in a linked list, which of the following applications ***would not be efficiently solved by using a linked list?*** (in particular, consider the order in which information is likely to be accessed!)

a) Storing the words in a large book we are editing

b) Storing the list of transactions for a credit card, so we can compute the monthly balance

c) Keeping a music player's current playlist

d) Keeping a record of the minute-by-minute price of a stock to make a graph for the year.

e) Keep track of the queue of airplanes departing Pearson airport

---

8.- From the ***Unit 3 notes*** we read that ***hacking at code before you have gone through the process of solving the problem*** will cause any/all of the issues below ***except for one*** (which one?).

a) Make it harder to come up with a robust and efficient solution

b) Make your code less readable

c) Yield code that is less organized and harder to maintain

d) Lead to code that is fragile and easy to break

e) Produce a solution that does not do what was expected

9.- Which of the complexity classes below corresponds to the ***hardest problems*** we can find in computer science?

a) O(N!) b) $O(N^2)$ c) O(N Log(N)) d) $O(2^N)$ e) O(N)

10.- We care about the ***Big O*** complexity of an algorithm because ***(one of the statements is not correct, which one?)***

a) The results are valid independently of computing power available

b) It let's us compare different algorithms independently of their implementation

c) It gives us information about what algorithms will be faster ***for every possible input***

d) It applies to the large problems, the type of problem we will need to solve if we handle large collections of data (e.g. Google, Amazon, etc.)

e) It provides a mathematically rigorous way to analyze and compare algorithmic efficiency

---