1)

*int * x = 1;*

The above statement creates a pointer x and tells it to point to memory address 1 (right?).
Any decent compiler will tell you this is invalid: **cannot initialize a variable of type char *' with an rvalue of type 'int'**

But if I do:

*char * x = "Hello!";*

x doesn't point to an address named "Hello!", instead to points to the first element in the character array storing the string "Hello!". why does this happen?

Magic. The C compiler takes any strings, (sequences of chars surrounded by “” quotes) and stores them in a special part of the application binary. So when the application loads, x has a pointer to this string in the special part of the compiled binary (which has been loaded into memory).

We can decompile some binaries to see how this works. (see q1.c)

2) Malloc returns an address belonging to a block of memory in the heap right?
It sure does!

3)Why use malloc for dynamic strings if using an array of characters also allows me to set aside memory for a string which I can both read and write to (like if I used malloc instead).
Good question. See q3.c

4)
Is this:
char x = 'a';
the same as:
char x = "a"
and:
char x = a;
If not, how are they different?
Nope, not at all.

char x = 'a' ;

This sets x to the literal char constant for a (integer 97).

char x = "a";

This actually won't compile. You will get an incompatible pointer error. Since "a" means a string of length 2 -> ['a', 0]. Remember the terminator counts. See the decompilation example in q1 which explains why this pointer magic happens.

char x = a;

Assuming you have the variable **a** this will set x to be whatever **a** has stored in it.

5)An ADT "interface" is simply a list of functions a person can use without knowing anything about the code for these functions right? Is this like stdio.h and stdlib.h?

Similar, but not quite (well not the functions you guys are using at least – you will see FILE pointers next semester).

The idea of an ADT that it abstracts the inner workings of your own data structure. People on the outside world need to use the provided interface functions to interact with it. They don't know how it works on the inside. This means you can change how you implement your struct at any time without affecting other peoples code.

The header file for your ADT should only provide a list of functions that the ADT provides and the ADT pointer typedef.

6)Salil used the term "constant memory" when talking about how strings are stored. What is this?
See in question 1 where we decompiled the binary. This is where strings that are quoted in your program, such as "Hello!" are stored. char * str[20] is stored in the frame, and char * str = malloc(20); is stored in the heap.

7)If I define a character array like this:
int arr[];

What is happening? I haven't specified a size for the array but the compiler is fine with it.

It shouldn't compile: "definition of variable with array type needs an explicit size or an initializer".

8)Say I did this:

char aFunction(char x[2]);

what exactly is **aFunction** expecting for it's input?

Confusing! I actually could not work out an answer to this, but in general practise – this is wrong (I think, we should get confirmation of this from a C overlord). Running this yields expected results, it takes a pointer to a char array, and I can index it like normal, accessing elements from 0-n (no limitations here as the '2' would make you think).

See the sample code in q8.c

9)To do this:
x->y->z;
Do both x and y have to be pointers since we are dereferencing them? Does y and z both have to be elements of a struct since arrow notation only works when the pointer is pointing to a struct?

Yes, x must be a pointer and so must y. Even still, if either point to NULL you will get a segfault. Yes **x** would be a struct which has a pointer variable **y. Y** would also be a struct since it has a property **z. Z** doesn't have to be a struct, **z** could just be an int.