```
#include <stdio.h>
struct car
{
int size;
char *name;
};
int main()
{
struct car car1;

car1.name = "bmw";
car1.size = 10;

struct car car2;

car2.name = "ferrari";
car2.size = 20;

printf(" car1 %d, %s\n",car1.size,car1.name);
printf(" car2 %d, %s\n",car2.size,car2.name);
return 0;
}
```

in this code is struct car does not loading in memory until you make a variable of type struct car car1
I meen now var store in memory have the name size or name only car1.size and car1.name

please explain what happen when I init struct and make variable of his type in memory lyout

In C, when you define a struct, you are essentially creating a new data type. However, at the point of definition, no memory is allocated yet. The `struct` only serves as a template for the compiler. Memory is allocated for the struct when you declare a variable of that struct type.

Here's what happens in memory with your `struct car` and its instances:

1. **Defining the struct:**
   When you define the struct `car`:

   ```
   struct car
   {
       int size;
       char *name;
   };
   ```

   You're telling the compiler that `struct car` will have an `int` called `size` and a `char*` called `name`. This doesn't allocate memory; it only informs the compiler of the layout of the struct.

2. **Declaring a struct variable:**
   When you declare a variable of type `struct car`, such as:

   ```
   struct car car1;
   ```

   At this point, the compiler allocates memory for `car1`. The memory size will be at least the sum of the size of an `int` (for the `size` field) and the size of a `char*` (for the `name` field), plus any padding that may be required for alignment purposes.

3. **Initializing the struct variable:**
   When you initialize the fields of `car1`:

   ```
   car1.name = "bmw";
   car1.size = 10;
   ```

   You're storing the value `10` in the memory location set aside for `car1.size`. For `car1.name`, you're storing the address of the string literal `"bmw"` in the memory location for `car1.name`. The string literal `"bmw"` is typically stored in a read-only section of the memory (like the text segment), separate from the memory allocated for `car1`.

4. **Memory layout:**
   The memory layout for `car1` and `car2` would look something like this (not to scale and the actual layout can vary based on architecture, compiler, and padding/alignment):

   ```
   car1 memory layout:
   +--------+-------------------+
   | size   | name              |
   +--------+-------------------+
   | 10     | address of "bmw"  |
   ```

```
+--------+-------------------+

car2 memory layout:
+--------+----------------------+
| size   | name                 |
+--------+----------------------+
| 20     | address of "ferrari" |
+--------+----------------------+
```

Here `size` would be however many bytes an `int` takes on your system (commonly 4 bytes), and `name` would be the size of a pointer on your system (commonly 4 bytes on a 32-bit system and 8 bytes on a 64-bit system).

5. **Accessing struct members:**
   When you access `car1.size` or `car1.name`, the compiler knows to access the memory region associated with `car1` and offset it by the appropriate amount to reach the `size` or `name` field.

So, in summary, when you initialize a struct and create variables of its type, the compiler sets aside memory for each instance of the struct. The names `size` and `name` within the struct's context only inform the compiler of the structure of that memory; they are not variable names in the traditional sense that are directly stored in memory.

From ChatGPT, designed with FancyGPT