Struct in a struct example

Imagine adding these struct definitions to our former example:

```
struct doll {
  char name[20];
  double price;
};

struct toys {
  struct block rubix;
  struct doll barbie;
};
```

Now, consider the following functions prototypes and main:

```
void setup_block(struct block *b);
void print_block(struct block b);
void setup_doll(struct doll *d);
void print_doll(struct doll d);

void init(struct toys *t);
void print_all(struct toys t);

int main() {
  struct toys mine;
  init(&mine);
  print_all(mine);
  return 0;
}
```

Here are the added functions:

```
// Post-condition: b will be initialized with information entered
// by the user.
void setup_doll(struct doll *d) {

  printf("Enter name,cost of the doll\n");
  scanf("%s %lf", &(d->name), &(d->price));

}

// Post-condition: b's components will be printed out.
void print_doll(struct doll d) {
  printf("Name: %s, Price: %lf\n",
                      d.name, d.price);
}

// Post-condition: Initializes all components of the struct t
//                 points to.
void init(struct toys *t) {
  setup_block(&(t->rubix));
  setup_doll(&(t->barbie));

}

// Post-condition: Prints out all information about t.
void print_all(struct toys t) {
  print_block(t.rubix);
  print_doll(t.barbie);
}
```

Notes about the Toy Example

Notice how each function that processes a doll struct doesn't need to "know" about a toy struct or a block struct. The syntax of these functions is COMPLETELY independent of either of those two other structs.

Now, when you declare a toy struct, in order to access and deal with its doll component, you must properly call functions that take in a doll struct. Consider the following call from the init function:

```
setup_doll(&(t->barbie));
```

It was important to pass the function a memory address, so we did. It was also important to pass the function a memory address to a doll. So, we had to access the doll component of the toy struct.

Also, the same care had to be taken for the block component o the toy struct. As this example shows, we can build arbitrarily complex structs, where one struct is part of another, that is part of another etc.

It is most important to keep track of which struct is which, and which components each struct is made up of. This way you can always properly access the component you want. Finally, it's important to differentiate between when a dot is needed and when an arrow is needed. Just remember, an arrow is just shorthand for a * preceding a variable with a dot following that variable. ***It is completely dependant on whether a variable is a struct itself or a pointer to a struct.***

An example with an array as part of a struct

```
#include <stdio.h>

#define SIZE 10

struct hotel {
  int rooms[SIZE];
  int available;
};

void init_hotel(struct hotel *h);
void get_room(struct hotel *h);
int find_room(struct hotel *h);

int main() {
  struct hotel plaza;
  int ans;
  init_hotel(&plaza);

  printf("Would you like to check out a"
        +"room?(y=1,n=0)\n");

  scanf("%d", &ans);
  while (ans == 1) {
      get_room(&plaza);
      printf("Would you like to check out a"
            +"room?(y=1,n=0)\n");

      scanf("%d", &ans);
   }
   return 0;
}
```

```
void init_hotel(struct hotel *h) {

    int i;
    for (i=0;i<SIZE;i++)
        h->rooms[i] = 1;
    h->available = SIZE;
}

void get_room(struct hotel *h) {
    int room_no;

    room_no = find_room(h);

    if (room_no == -1)
        printf("Sorry, we are full.\n");
    else {
        printf("You will stay in %d.\n",
                                          room_no);
        h->available--;
        h->rooms[room_no] = 0;
    }
}

int find_room(struct hotel *h) {

    int i;
    for (i=0; i<SIZE; i++) {

        if (h->rooms[i] == 1)
           return i;
    }
    return -1;
}
```

Explanation of the Hotel Room Example

The structure stores an array that keeps track of a room's availability and an integer that keeps track of how many more rooms are available.

Each time a room is requested, the find functions goes through the rooms, in order, looking for a vacant one. Once one is found, this room number is returned. A -1 return value indicates a full hotel. An easier way to check for availability than the code above would be to simply check the value of the `available` component of the hotel struct. As it is written the code simply "double checks" availability.

Each time a room is actually found, two updates have to be made:

1) Update the proper room to be "taken" (ie. put a 0 here.)
2) Reduce the `availabile` component by 1.

Natural extensions to this code:

1) Allowing customers to check out.
2) Allowing multiple hotels.
3) Keeping track of WHICH guest is in each room as opposed whether or not a guest is in the room.