struct and typedef | 10.3 Structures | UT.6.0... https://courses.edx.org/courses/UTAustinX/UT...
(https://www.edx.org)
UTAustinX: UT.6.01x Embedded Systems - Shape the World
KarenWest (/dashboard) ▼
Courseware (/courses/UTAustinX/UT.6.01x/1T2014/courseware) Course Info (/courses/UTAustinX/UT.6.01x/1T2014/info)
Discussion (/courses/UTAustinX/UT.6.01x/1T2014/discussion/forum) Progress (/courses/UTAustinX/UT.6.01x/1T2014/progress)
Questions (/courses/UTAustinX/UT.6.01x/1T2014/a3da417940af4ec49a9c02b3eae3460b/)
Syllabus (/courses/UTAustinX/UT.6.01x/1T2014/a827a8b3cc204927b6efaa49580170d1/)

Help

A **structure** has elements with different types and/or precisions. In C, we use **`struct`** to define a structure. The **`const`** modifier causes the structure to be allocated in ROM. Without the **`const`**, the C compiler will place the structure in RAM, allowing it to be dynamically changed. If the structure were to contain an ASCII string of variable length, then we must allocate space to handle its maximum size. In this first example, the structure will be allocated in RAM so no **`const`** is included. The following code defines a structure with three elements. We give separate names to each element. In this example the elements are **`Xpos Ypos Score`**. The **`typedef`** command creates a new data type based on the structure, but no memory is allocated.

```
struct player{
  unsigned char Xpos;     // first element
  unsigned char Ypos;     // second element
  unsigned short Score;   // third element
};
typedef struct player playerType;
```

We can allocate a variable called **`Sprite`** of this type, which will occupy four bytes in RAM:

```
playerType Sprite;
```

We can access the individual elements of this variable using the syntax **`name.element`**. After these three lines are executed we have the data structure as shown in Figure 10.4 (assuming the variable occupies the four bytes starting at 0x2000.0250.

```
   Sprite.Xpos = 10;
   Sprite.Ypos = 20;
   Sprite.Score = 12000;
```

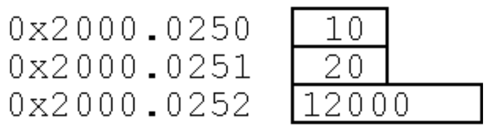

Figure 10.4. A structure collects elements of different sizes and/or types into one object.

We can also have an array of structures. We define structure array in a similar way as other arrays

```
playerType Ships[10];
unsigned long i;
```

While we are accessing an array, we must make sure the index is valid. In this case, the variable `i` must be between 0 and 9. We can read and write the individual fields using the syntax combining array access and structure access.

```
    Ships[i].Xpos = 10;
    Ships[i].Ypos = 20;
    Ships[i].Score = 12000;
```

The C function in Program 10.3 takes a player, moves it to location 50,50 and adds one point. For example, we execute `MoveCenter(6);` to move the 7th ship to location 50,50 and increase its score.

```
// move to center and add to score

void MoveCenter(unsigned long i){
  Ships[i].Xpos = 50;
  Ships[i].Ypos = 50;
  if(Ships[i].Score < 65535){
    Ships[i].Score++;
  }
}
```

Program 10.3. A function that accesses a structure.

Observation: Most C compilers will align 16-bit elements within structures to an even address and will align 32-bit elements to a word-aligned address.

Without the **const**, the C compiler will place the structure in RAM, allowing it to be dynamically changed. If the structure resides in RAM, then the system will have to initialize the data structure explicitly by executing software. If the structure is in ROM, we must initialize it at compile time. The next section shows examples of ROM-based structures.

Help

Help