

Chapter 19

Structs

This section is about making own own new variable types. It's not as exciting as it sounds, since all we can do is group together `ints`, `floats` and `strings`. But it's a nice trick, and it's the basis for classes and Object Oriented Programming.

For example, it takes three `floats` to make a color (red, green, blue.) A struct lets `Color` be an official type, which the computer knows is made of three `floats`.

Even better, the trick works for stuff we just make-up. Suppose we're writing a program about cows – each has a name, age and weight. It would be nice to build that into the program. Then we could declare `Cow c1`; and let the computer automatically make those variables for the parts.

The common name (and the one `C#` uses) for these is **struct**. There are some built-ins, and we can make our own. I think the best way to learn them is to make the `Cow` example, then a few more, then give the formal rules.

19.1 Cow struct example

Here's the top part of a program making a sample `struct` named `Cow`:

```
class TestA : MonoBehaviour {  
  
    struct Cow {  
        public string name;  
        public int age;  
        public float pounds;  
    }  
}
```

First, let's take a look at the overall form. The last five lines, `struct Cow { }`, count as one thing. The name `Cow` is in front. Then a big set of curly-braces mark what belongs to `Cow`. The inside looks like three variable declarations. They aren't, but it's the same rules.

Now onto what it does. This makes a new **type** named **Cow**, and says how to build it. It doesn't declare any variables, or really do anything. It's more like a recipe or a blueprint. It lets the computer know that if someone declares a **Cow** variable, it's legal, and says how to make one.

The names inside are called **fields**. It's the same use of the word as those web-based forms that say "required field." They describe the parts of a **Cow**.

Now we can declare some **Cows**:

```
int n; // old, boring declare just for reference

Cow c1;
Cow c2;
```

`Cow c1;` declares `c1` as a **Cow**. Just like you're probably thinking, the computer says "A Cow? How do you declare a Cow?" But then it looks near the top and sees `struct Cow`. That tells the computer it *can* declare a **Cow**, and the middle part tells it what the parts are – a string and int and a float.

`Cow c2;` declares another **Cow**. Nothing new there – I just want two **Cows** for later in the example.

Here's a picture of all 3 variables in the program:

```
n ---

c1          c2
-----
| name:    | | name:    |
| age:     | | age:     |
| pounds:  | | pounds:  |
-----
```

Again, `n` is nothing special – it's just there for comparison. The interesting thing is that `c1` is one variable, with those three parts inside of it. Then `c2` is another variable, with the same three parts. Inside of `c1`, `name` is a string. There's another string, `name`, inside of `c2`.

Here's a program using `c1` and `c2` (I'm including everything here, so only **Start** is new):

```
class TestA : MonoBehaviour {

    struct Cow {
        public string name;
        public int age;
        public float pounds;
    }

    int n;
```

```

Cow c1;
Cow c2;

void Start() {
    c1.name = "Bessy";
    c2.name = "Miss Cowy-cow";

    c1.age=6;
    c2.pounds=1200;

    print( c1.name + " is " + c1.age + " years old" ); // Bessy is 6 years old
}
}

```

This is showing us the last thing we need – how to use the parts of a **struct**. `c1.name="Bessy"`; shows the rule: put the variable name, then a dot, then the **field**. `c1.name` is the **name** slot in the first cow.

After you type "`c1.`" (`c1`, then a dot,) autocomplete will even show you the fields: **name**, **age** and **pounds**. It knows about them from the **struct** `Cow` we made at the top.

The first two lines are showing us that each `Cow` has it's own **name**. `c1.name` and `c2.name` are different. In other words, the picture is correct. The next two lines are using the **age** and **pounds** fields, just to show we can. `c2.pounds=1200`; goes to `c2`, then inside its **pounds** field, and sets that to 1200.

The **print** shows that the “varName-dot-field” rule is like a variable – you can assign to `c1.name` and also read from it.

After `Start` runs, here's the new picture:

```

n ---

c1                c2
-----          -----
| name: Bessy |  | name: Miss Cowy-cow
| age: 6      |  | age:
| pounds:     |  | pounds: 1200
-----          -----

```

The picture is really pretty boring. `Start` assigned four things, and the picture shows all four of them. The **pounds** in the lower-right is `c2.pounds`, the last assignment put 1200 in it, and there it is.

`c1.pounds` and `c2.age` are empty, since the code didn't put anything in them. If I tried to print them, I'd get the usual “has not been initialized” error.

L-values

In the assignment chapter, I made a big deal about how the thing on the left had to be a variable, all by itself. You could never have a formula – `x+1 = 7;` is an error. But here, the left side isn't quite a variable. `c1.age` says to find variable `c1`, and then look inside for the `age` part. It's like a little formula, so `c1.age=6;` looks like it breaks the rule.

The real rule for assignment statements is that the left side needs to count as a variable. It needs to be a box that can be changed. The technical name is **L-value**. It stands for “value that can be on the left-hand of an assignment statement.” I know **L-value** sounds really fake, but put “lvalue” in a Search Engine – that's really what we call it.

Before this, only actual variables were **L-values**. Using a struct field is a new thing that also counts as a variable. In other words, in every way, `c1.age` works like an `int` variable. You can use `c1.age++;` or `c1.age *=2;` or `c1.name = "a" + c1.name + "b";`.

It takes a while to get used to seeing mini-formulas that count as variables, but you get used to it. Later, we'll see other things like this – they aren't simple variables, but they count that way.

19.1.1 More examples

This is a traditional very basic **struct** example, with just two fields of the same type. It makes a `FullName` **struct** containing a first and last name. This mini-program creates it, declares some `FullName` variables and uses them a little:

```
// create new struct type "FullName":
struct FullName {
    public string first, last;
}
```

As with `Cow`, this doesn't create any variables. It defines `FullName` as a new struct type.

The fields are both the same type – two strings. You can re-use any types you need for the fields. This also uses the comma-declare shortcut. The same as always it's just a shortcut and isn't any different than declaring them on two lines. A trick with choosing names is everyone knows we're inside of `FullName`, so just `first` and `last` are descriptive enough.

The start of the code using them:

```
void Start() {
    FullName mom1, mom2;

    mom1.first = "Olga";
```

```

mom1.last = "Pog";

string m = mom1.first + " " + mom1.last;
print( "Mother1's name: "+ m ); // Olga Pog

```

Structs can also be declared using the comma shortcut. `mom1` and `mom2` were declared inside of `Start` (the `Cow` example declared them outside.) Struct variables can go in both places, and have the same rules for local/global.

You can use struct variables in pretty much all the ways you'd use ints or strings.

The two lines assigning to `mom1` are pretty typical. In our minds we're assigning "Olga Pog" as her name, and it takes two steps. It common to assign to every field, all together like this.

In the next line I wanted to show the fields act like strings in all ways: `mom.first + " " + mom.last` is just three strings added together.

I'll use some other old tricks with `mom2`:

```

mom2.first= "blue" + "elf" + 451;
mom2.last = mom1.last;
mom2.last += "ie";

print( "Mother 2 : "+ mom2.first + " " + mom2.last ); // blueelf451 Pogie
}

```

The second line, `mom2.last = mom1.last;` is just copying a string to a string, even if it took those funny dots to get to them. In the next line, the `+=` shortcut works as normal, tacking `ie` to the end.

Here's a short code snippet using a `Cow` and a `FullName` together. Nothing really special, just to show it can be done:

```

Cow a1;
a1.name = "Lou-Lou";
a1.age=7;

FullName p1;
// person has same name as the cow:
p1.first = a1.name; // no problem -- just string = string
p1.last = "Smith";
print( p1.first + " " + p1.last ); // Lou-Lou Smith

// a1.last = "Cowstein"; // ERROR no last in a Cow
// p1.age = 85; // ERROR no age in a FullName

```

`p1.first = a1.name;` is one of those things that can confuse you later. First off, it's simple: `p1.first` is a string, and so is `a1.name;`, so it's no different than `w1=w2;`.

The possibly confusing part, if you start thinking about it, is how we're copying from the inside a `Cow` into the inside of a `FullName`. It sort of looks like the types are different. The trick is, every step of the way doesn't have to match – only the end. You can copy a string field from a `Cow` into a string field of a `FullName`.

Here's the picture, after it runs:

```

a1                                p1
-----                            -----
| name: Lou-Lou | | first: Lou-Lou
| age: 7        | | last: Smith
| pounds:       | -----
-----

```

The two errors at the end are showing how each `struct` type has its own personal fields – the ones listed in its `struct { }`, and nothing else. `Cows` don't have last names, because we didn't write `public string last` inside `Cow`.

This next example is about making a semi-useful `struct` to improve my old move-and-wrap code, and using some Unity-magic to see it.

To make something increase and wrap-around, I need 4 variables: the value, the speed, the maximum and the minimum (the last two are for the wrap-around part.) In my mind, those four variables make one variable-mover. So I'll make a `struct` for it:

```

struct Mover {
    public float val; // the main thing
    public float spd; // amount to change val, each update
    public float min, max; // the limits for val
}

```

I want to use them in a program that changes the red and green parts of my color. Each will use its own `Mover` variable:

```
Mover red, green;
```

That creates the 4 floats I need for red, and the 4 for green. Here's a picture:

```

red                                green
-----                            -----
| val    | | val    |
| spd    | | spd    |
| min    | | min    |
| max    | | max    |
-----                            -----

```

We need to add some special Unity tricks to see them in the Inspector. Adding `public` in front of a global normally makes Unity show it, but `structs` need two extra things. You have to also write `public` before the struct definition, and `[System.Serializable]` on the line before. That second thing is a real C# thing, but it's pretty far-out and Unity is abusing it anyway, so I'll just say it's magic.

The final result looks like this:

```
[System.Serializable]
public struct Mover {
    public float val;
    public float spd;
    public float min, max;
}
```

```
public Mover red, green; // now they're visible in the inspector
```

The really cool part is, if you put just that in a script (before `Start`.) on your Cube, the Inspector will show `red` and `green`. If you pop-open the little triangles, you can see that each one has `val`, `spd`, `min` and `max`. It knows about `struct Mover` and makes a picture for us!

Here's some code using them. It's the same color change code from way back, except using the struct:

```
void Update() {
    red.val += red.spd;
    if(red.val > red.max) red.val=red.min;

    green.val += green.spd;
    if(green.val > green.max) green.val=green.min;

    GetComponent<Renderer>().material.color = new Color( red.val, green.val, 0);
}
```

I like how a simple `struct` helps organize this. The program really just has 8 variables, and before we would have hand-declared all 8, like `float redMax`;. Using the `struct` automatically makes `red` and `green` have the same 4 subparts; and typing `red.dot` and using the pop-up is an easy way to write the program. As a bonus, using the struct lets the Unity Inspector give us those nice toggles for `red` and `green`.

For running it, the variables start as all 0, so this won't do anything yet. Start could give them values, but it's easier just to enter values in the Inspector. Try making them move at different speeds, maybe something around 0.01. Try changing green min/max to go from 0.5 to 1, or 0 to 0.5. With a little work, they can cycle through a bunch of different shades before repeating.

19.2 Common errors

Seeing some common struct errors is a good excuse to talk about the rules, and explain more about how they work:

- `Cow.name = "steve";` is an error, because `Cow` isn't a variable. In the sample program, there are two cows: `c1` and `c2`, so there are only two names – `c1.name` and `c2.name`. `Cow` is a **type**, so `Cow.name="Steve";` is like `int = 6;`. The thing in front of the dot has to be an actual declared variable.
- `age=6;` won't change the age of any cows, and is probably an error. There has to be a cow in front – `c1.age` or `c2.age`. Even if you have only one cow variable, `age=6;` won't assume you mean that cow.
- `c1 = "Elly";` is an error. Struct variables always have to use a dot and say which field you want (there are a few exceptions.) Even though `name` is the only place it could go, you still have to write `c1.name="Elly";` The computer will never “guess” a **field** for you. Also, as a double-check, the types don't even match: `c1` is a `cow` and `"Elly"` is a string.
- `FullName p; p="Mark";` is an error. I thought that since both parts of a `FullName` are strings, maybe the computer would change them both to "Mark". But it still won't guess fields for me. I have use `p.first = "Mark";`. Or I could use the old same-assign shortcut: `p.first=p.last="Mark";`.
- `print(c1);` isn't a red-dot error, but won't work. You have to print out each part yourself: `print(c1.name+", "+c1.age+ ...)`. Most commands with **structs** are that way. For example, you can't use `f1+f2` on two `FullNames`. You have to add `f1.first+f2.first` yourself.
- `int age;` is *not* an error! The field names only live inside that struct. If you declare variable `age` like that, and use `age=7;` the computer knows you're talking about the regular `age` variable. If you use `c1.age=7;`, it's definitely the `age` field of `c1`.

Here's a picture of cow `c1`, `c2`; float `age`;

```

c1                c2
-----
| name:          | | name:
| age:           | | age:
| pounds:       | | pounds:
-----
```

```
age:
```

The three ages are `c1.age`, `c2.age` and `age`. The regular `age` could even be a float or a string. That `age` and the Cows' `age`'s are totally unrelated.

19.3 rules

We've seen most of the rules in examples, but it's still nice to have them all in one place. For most, just skim them to be sure you've gotten them right. But there should be a few new things in here:

- A new **struct** type is defined using `struct identifier { list-of-fields }`. It goes in the global area. **struct** is short for *structure*, as in a small building.
- The name of a struct is any legal identifier: `a`, `d.2`, `bunchOfRocks` We try to use style rules so you don't confuse struct names with variable names. One rule is to capitalize the first letter. Another is to add underscore-t (for example `cow_t`).
- Each field looks like a variable declaration with the word **public** in front: any type, a space, and an identifier. Ex: `public float f;`. They aren't really variable declarations, but it seemed right to borrow the syntax. If you forget the word **public**, it isn't a "red dot" error, but it means you can't use that variable (this is a somewhat silly, tricky rule. If you make a struct, but you get errors when you try to use a part, check you put **public** in front of that one.)
- Can use the multi-shortcut for fields: `struct FullName { public string first, last; }`
- Fields can be different types, the same type, or any combination. There's nothing special about a struct where all fields happen to be the same type, or all different.
- It's legal to have only one field, or even zero. That's usually pointless, but sometimes people have a reason to do it. Ex:
`struct Number { public int val; }`. If you do that, you still have to use the fields: `Number x; x.val=5;`
- The order you create fields doesn't matter. In other words, the definition of `Cow` could have `age` first, and it would work the same. Since you look them up using words, the computer finds the correct field no matter where it is.
- The field names can be reused outside of that **struct**. Here's a legal example where `Book` uses `title` as a field name, and so does `Song`, and we also declare it as a global:

```
struct Book { public string title; public int pages; }
string Song { public string artist, title; public float length; }
int title;
```

The three `titles` have nothing in common with each other. It's just a coincidence they have the same name.

- There isn't any short-cut for using field names by themselves. Even if you only have 1 `Cow` variable, `age=6;` won't work on it. You always have to have `variable-dot-fieldName`.
- Structs have a special `=` shortcut to copy one entire struct to another. For two cows, `c1=c2;` means to use `=` on each part. It's a shortcut for `c1.name=c2.name; c1.age=c2.age; c1.pounds=c2.pounds;`.
There's a new error here as well: every part must have been assigned. `FullName f1, f2; f1.first="A"; f2=f1;` is an error. You have to at least write `f1.last=""`; before you can copy it.
- Different structs always count as different, even if they have the exact same fields. For example, if you made `struct Sheep` with the exact same contents as `struct Cow`, and had `Cow c1; Sheep s1;`, then `s1=c1;` would be an error.

This is on purpose. If you made one `struct` for sheep and another for cow, it's because you want the computer to tell you when you mix them up.

Of course, `c1.name = s1.name;` is always legal – it's just a string to a string.

19.3.1 namespace vs. struct dots

We already know `C#` uses the dot for namespaces. In that chapter I warned you we were going to double-use the dot for something else. This is the second use. An example of both uses:

```
float n = Mathf.PI; // namespace
int a = c1.age; // inside struct
```

They both look inside the thing before the dot, but besides that, they're very different. `Mathf` isn't a variable, and there's one `Mathf.PI`. It's really a built-in global.

On the other hand, there's one `age` for each `Cow` we declare; and `Cow.age` is an error.

There really should have been a different symbol for each, and it is confusing at first, but you get used to it. When you see `Jark.groat` you just have to check whether `Jark` is a namespace, in which case `groat` is a global there; or whether `Jark` is a declared variable, in which case `groat` is a field.

Examples:

```
Cow c1; c1.age=Random.Range(0,8); // 1st dot is field, 2nd is namespace

int n = Mathf.Max(c1.age, c2.age); // 1st dot is namespace, rest are fields
```

19.4 Builtin structs: Color, Vector3

We know Unity3D's position and size are a combined x,y&z. Now that we've seen structs, it should be obvious that's how Unity is doing that. `Vector3` is just three floats, named x, y and z. It's already in the computer, so you can't redefine it, but here's what it looks like:

```
struct Vector3 {
    public float x, y, z;
}
```

The name is from math: vector is a math term for list of numbers, and the 3 at the end is a reminder it has 3 parts.

The old position line, `transform.position = new Vector3(x,0,0);`, was using a shortcut. `new Vector3(x,0,0)` is a way to create an instant struct. Creating a normal `Vector3` variable can be nicer. Here's the old and the new way:

```
void Start() {
    transform.position = new Vector3(0,0,0); // old way

    // new way:
    Vector3 upperLeft;
    upperLeft.x=-6.5f; // just pick some numbers for x,y,z
    upperLeft.z=4.0f;
    upperLeft.y=0;

    transform.position = upperLeft;
}
```

The last line is assigning a struct to a struct – it copies the x,y,z from `upperLeft` into `position` (the old line also did that, but I didn't make a big deal about it.)

We'll see how this improves movement code, later.

For fun, write `public Vector3 p;` as a global variable in any script and look in the Inspector. You should see the x, y and z fields going across. It lists them that way instead of with the triangle-toggle, since it's built-in – they tried to make it look nicer.

This sort-of fun script will slide to how-ever you change any part of `p` (use the right-click slide trick to slide the inspector numbers):

```

public Vector3 p;

void Update() { // using Update so it moves as you change p
    transform.position = p;
}

```

Scale is also a `Vector3`. That might seem odd, since `Vector3` is for position, but it makes perfect sense. After all, we use floats for anything that wants a decimal – x location, x size, how red to be, how much a cow weighs. Anytime we want an x,y,z together, we should use `Vector3`.

This make us be wide and skinny (I'm leaving out the old way):

```

void Start() {
    Vector3 sz;
    sz.x=3; // wide
    sz.y=sz.z=0.4f; // standard double-assign trick
    transform.localScale = sz;
}

```

To to make the point both use a `Vector3`, we could re-use the same one for position and size:

```

void Start() {
    Vector3 v; v.x=6; v.y=-2; v.z=0; // just some position
    transform.position = v;

    v.x=0.4f; v.y=2; v.z=1; // redo v to use as scale
    transform.localScale = v;
}

```

19.4.1 Color

We've been setting colors using r, g and b – another struct. It looks like this (Unity already has it defined):

```

struct Color {
    public float r, g, b, a;
}

```

The very short field names are fine, since they're inside of a color – r clearly stands for red. The last one, a, is “alpha” for transparency. It won't do anything (setting up the ability to be transparent takes a few steps.)

An interesting thing is we already have `Vector4`, which is four floats, just like `Color`. Making two structs with the same contents is legal. And it's nice having the descriptive name `Color` and the nicer field names (having to use x for red would be confusing.)

If you declare `public Color c1;` as a global, the Inspector will make a fancy color-picker. In the mini-window you can change r, g, b and a; or you can directly select the color. That's what cool GUIs are for, but the end result no matter what you slide, is setting those 4 floats.

This will turn us orange, using a `Color` variable. Even though alpha is ignored, we have to set it to something – the rule where you can't assign a struct unless every field is set:

```
void Start() {
    Color cc;
    cc.r=0.9f;
    cc.g=0.6f;
    cc.b=0.1f;
    cc.a=1; // ignored, but must be set to something

    // same old line, but assigning our Color variable:
    GetComponent<Renderer>().material.color = cc;
}
```

19.5 Constructors, struct literals

My old way of setting position was `transform.position = new Vector3(x,y,z);`. The part after the `=` is a **constructor**. The real rules for them are in the section on **classes**, but I want to introduce them here, since I've been using them.

A **constructor** is a way to make a struct out of parts, without declaring one. `new Vector3(1,2,3);` counts as a `Vector3`. It's the same as `Vector3 a; a.x=1; a.y=2; a.z=3;`, without declaring `a`.

The syntax is always: **new**, then the name of the struct, then parens around the inputs (like a function call.) For example `new Color(1,1,1);` or `new Cow();`.

Constructors have to be defined inside the struct. This means the structs we write won't have them. Built-in structs usually do. It also means we often write 2 or more with different inputs.

If you type "`new Vector3`" the drop-down shows you three versions. You can give it x, y and z, like we've been doing, or give it no inputs (it makes all 0's) or just give it x and y (sets z to 0.) Here are some simple constructor examples with `Vector3` and `Color`:

```
Vector3 a1, b1, c1;
a1 = new Vector3(1,2,3); // old way
b1 = new Vector2(10,3); // makes (10, 3, 0)
c1 = new Vector3(); // all are 0's
```

```
Color c1; c1= new Color(1,1,0); // shortcut for c1.r=1; c1.g=1; c1.b=0;
```

But again, they're only shortcuts – we can always set each field by hand.

All structs, even ones you make, get a free constructor with no inputs. It sets everything to 0 or "". For example `FullName ff = new FullName();` is a shortcut to set `ff.first=""`; `ff.last=""`;

19.6 Nested structs

The fields in a `struct` can be any type, even another `struct`. We usually call that a nested struct, like those Russian dolls nested inside of each other. It's a useful thing to be able to do. There aren't any special rules for this – structs inside structs work the normal way – but they can look funny. This is just a lot of examples.

I want to change the `Cow` struct so each `Cow` also has two `Color`'s. So that's a struct in a struct. To simplify, I'm getting rid of `age` and `weight`:

```
struct Cow {
    public string name;
    public Color mainCol;
    public Color spotCol;
}
```

Here's some code using it. I set each color a different way. For `mainCol` I copy an entire `Color` variable into it. For `spotCol` I copy in `r,g,b,a` one at a time:

```
Cow c1;
c1.name = "Bessy"; // same as before

Color blk; blk.r=0; blk.g=0; blk.b=0; blk.a=1;
c1.mainCol = blk; // copy a Color to a Color

c1.spotCol.r=0.4f;
c1.spotCol.g=0.3f;
c1.spotCol.b=0.1f;
c1.spotCol.a=1;
```

The last part with `c1.spotCol.r=0.4f`; is the most interesting. Using more than one dot isn't a new rule – it's just the way the usual struct rule works out. `c1.spotCol` is a `Color`, so you can use `dot-r`, `dot-g` or `dot-b` after it.

The trick is to read left-to-right. You can think of it as `(c1.spotCol).r`; (but no one writes it that way.)

As usual, once you get to a box, it counts as a variable of that type. `c1.mainCol.g` is a float – you can write `c1.mainCol.g+=0.2f`;

`c1.r` is an error (plus it makes no sense.) We have to use `c1.mainCol` or `c1.spotCol`, then we can add the dot-r. It's really the old can't-jump-over-a-field error.

Here's another example, putting one of my own **structs**, inside another. A `Plumber` has a `FullName` and an hourly rate:

```
struct FullName { public string first, last; } // no change

struct Plumber {
    public FullName name;
    public float hourlyRate;
}

FullName customer; // for comparison
Plumber p1;
```

The first thing is that this doesn't hurt or change `FullName`. We can still use it like normal, to declare `FullName customer;`. In other words, now that `FullName` is a type, anyone can use it, including `Plumber`.

Here's a picture of `p1`:

```
p1
-----
|           |
| name | first:
|           | last:
|           |
| hourlyRate:
-----
```

`p1` has two parts. That's what the definition of `Plumber` says. The first part, `name`, also has two parts: `p1.name.first` and `p1.name.last`.

Here's some code using `customer` and `p1`:

```
// this is just a regular FullName, to show it works the same:
customer.first = "Lom";
customer.last = "Lomson";

p1.hourlyRate = 24.50f; // nothing special here

p1.name.first = "Larch";
p1.name.last = "Larchenson";

// alternate:
p1.name = customer; // plumber's name is now Lom Lomson
```

Same as before, `p1.first` would be an error. You can't skip past `name`.

There's no limit to putting structs in structs in structs. You just use as many dots as you need.

19.7 Structs as function inputs

Since a struct is a type, it can be used as a parameter. Here's an example of a function which turns a `Cow` into a nice string (using my first simple `Cow`):

```
string cowDesc(Cow c) {
    // for fun, make each part, then add them at end:
    string wn = "name: " + c.name;
    string wa = c.age + " years";
    string ww = c.pounds + "lbs.";

    return wn+", "+wa+", "+ww; // Ex: name: Bessy, 3 years, 220lbs.
}
```

We could run it with `print(cowDesc(c1));`. As usual, parameter `c` becomes a copy of `c1`. When we call it, the system declares local `Cow c` and copies `name`, `age` and `pounds` into it. The inside is just usual struct stuff, nothing special at all.

This sort of function is common – structs can't print themselves, so we have to write a function to do it.

The same trick works with built-in structs. This is a somewhat silly function that tells us how bright a color is (it averages red, green and blue):

```
float getBrightness(Color c) {
    float sum=c.r+c.g+c.b;
    return sum/3.0f; // range is 0-1
}
```

You might notice I also used `c` in `cowDesc`. The usual function parameter rules say that isn't a problem. They're both local variables, so only matter inside of their own functions.

Here are two more fun “struct to string” functions, using `FullName` and `Plumber`. It uses some old tricks:

```
string FullNameDesc(FullName f) { return f.first + " " + f.last; }

string PlumberDesc(Plumber p) {
    string nm=FullNameDesc( p.name );
    return nm+", Hourly rate: $" + p.hourlyRate;
}
```


`FullNameDesc` is nothing special – it’s like `cowDesc` above.

`PlumberDesc` is more interesting – the first line calls `FullNameDesc` to get the plumber’s name. Nested structs often use nested function calls that way. If plumber names were printed some special way, we’d do it the long way – get `p.name.first` and `p.name.last` and combine them ourselves.

Passing structs to functions is another of the main reasons we make structs. It’s a lot clearer for a function heading to say it takes a `Cow` than to say it takes a string, ints and float. It’s also a lot easier to pass `cow1` to a function, than it is to pass a string, int and float.

19.8 Structs as return values

Functions can also return **structs**. There’s no special rule for it – these are just examples:

This one returns a random `Color`. I like it because it follows my standard pattern: declare variable for the answer, fill it, return it:

```
Color randCol() {
    Color ans;
    ans.r = Random.Range(0.0f, 1.0f);
    ans.g = Random.Range(0.0f, 1.0f);
    ans.b = Random.Range(0.0f, 1.0f);
    ans.a=1;
    return ans;
}
```

We can use it with `Color c1 = randCol();`. Just so you know, this method tends to get a lot of ugly browns and greys.

This one does a table look-up (pretend we have Colors 0, 1 and 2, and use an `int` to say which one to use):

```
Color numToCol(int colNum) {
    Color ans;
    if(colNum==0) ans=new Color(1,0,0); // red
    else if(colNum==1) ans=new Color(0,1,0); // green
    else if(colNum==2) ans=new Color(0.5f, 0.5f, 1); // light blue
    else ans=new Color(1, 0, 1); // error purple
    return ans;
}
```

It’s just a standard table-making cascading `if`, with a none-of-the-above in the final `else` (purple.) I used the constructor short-cuts to make the colors, but didn’t have to (I could have used curly-braces and set `rgb` by hand.)

A sample use is `GetComponent<Renderer>().material.color=numToCol(1);`.

A lot of programs use 0-255 for color values. For example, (255,128,0) is orange. This function makes a usable Color from 0-255 values. For example, `Color c = colFrom255(255,128,0);` will give us (1,0.5,0), which is the orange value in Unity:

```
// lets us make a Color using 0-255 numbers:
Color colFrom255(int r, int g, int b) {
    Color ans;
    ans.r = r/255.0f;
    ans.g = g/255.0f;
    ans.b = b/255.0f;
    return ans;
}
```

We could use `GetComponent<Renderer>().material.color = colFrom255(210,0,255);` to turn ourself blueish-purple.

The next, slightly fakey, function returns a `Vector3`. It takes a y coordinate and returns a point on the left edge of the screen (it just fills in -7 for x and 0 for z):

```
Vector3 leftPos(float y) {
    Vector3 ans;
    ans.z=0; ans.y=y;
    ans.x = -7; // left edge
    return ans;
}
```

We might use it like `transform.position = leftPos(-3);`, to put us on the left near the bottom.

It also shows the “field names only matter in the struct” rule. Input y is a regular float. `ans.y = y;` copies the input to the y field inside `ans`.

Finally, we can do both – use a struct for input and output. This one brightens (or darkens) the input Color. It uses the “modify the input” trick, to save declaring an extra variable:

```
Color brighterCol(Color c, float amt) {
    c.r+=amt; if(c.r>1) c.r=1;
    c.g+=amt; if(c.g>1) c.g=1;
    c.b+=amt; if(c.b>1) c.b=1;
    return c;
}
```

It’s also an example of a function that seems like it might change you, but doesn’t:

```
brighterCol(c1, 0.1f); // does nothing
Color c2 = brighterCol(c1, 0.1f); // c1 unchanged, c2 is a brighter c1
c1=brighterCol(c1, 0.1f); // makes c1 brighter
```

19.9 Uninitialized structs

This section is about one funny, extra “uninitialized” error you can get with structs: if you try to use the *whole* struct, you have to have a value in each part, even if you never look at it.

Before we get to this new error – it’s 100% legal to fill in some values in a struct, leave the rest uninitialized, and use only the ones you filled in. For example: `Cow c1; c1.age=12; int n=c1.age*2;`. The Cow’s name and weight don’t have any values, but we aren’t using them, so everything is fine.

The two times we get this special error are = (which was mentioned before) and passing to a function. Here’s another error example with =:

```
Cow c1; c1.name="harriet"; c1.pounds=1200; // didn't give age anything
Cow c2; c2=c1; // ERROR. c1 isn't all made
```

It seems as if this could be legal, and `c2.age` would be un-initialized. But that might be really confusing – `c2=c1;` looks like it initializes `c2`. So, they made copying a partially initialized struct be an error.

Here’s an example using a function parameter. `normalCowWt` only reads the age, but it still requires you to fill in the entire `Cow`:

```
float normalCowWt(Cow c) {
    return 20.0f + 40.5f*c.age;
}

Cow c1; c1.age=8;
print( normalCowWt(c1) ); // error
```

The function only reads `age`, and we filled in `age`, but it’s still an error. The function-calling rules say that `c1` has to be entirely filled-in. It tells us *Use of unassigned local variable c1*, which really means that not *all* of `c1` was initialized.

As a style thing, you should probably initialize all fields anyway. For example, if you don’t know how old a `Cow` is, maybe -1 is better than leaving it un-initied. Some people always start with the constructor `Cow c1 = new Cow();`, which initializes all fields to 0’s.