Chapter 12

Functions, part II - parameters

12.1 Parameters

If we had to write the built-in `print` function using the rules we have now, it wouldn't be very useful – just `print();`. We couldn't tell it what to print. `print("abc");` is an example of giving an input to a function. We can write our own functions that work the same way.

To do this, we're going to need several new rules: what the inputs can be, how the function call should send the inputs, how a function definition says it wants or doesn't want inputs, and how the function uses them.

The basic idea is that everyone, even functions, likes to read their data from variables. So if a function needs an integer input, we'll give it a specially marked integer variable. Then we'll make up rules to get our input into that variable. There are a lot of rules, but they're all just for this basic idea.

To get started, here's a 1-input function that turns integers into words. If you call `showAsWord(1)` this will print `"one"`:

```
void showAsWord(int n) {
  if(n<=0) print("zero");
  else if(n==1) print("one");
  else if(n==2) print("two");
  else print("many"); // a better version would have more numbers
}
```

The new thing in the heading is the `(int n)` inside the parentheses. It says that `showAsWord` must have an integer input. It also says the input will be in a variable named `n`. The computer does that automatically. When you call

`showAsWord(4);` the computer declares `n` and puts a 4 into it. Then it runs the function.

Now, to see how cool this is, cover up the heading. Just look from `if(n<=0)` on down. Those four lines are an ordinary cascading `if`. If we saw this anywhere else, we'd figure that `n` was declared somewhere and somehow got a value, but the exact way doesn't matter.

That's the key idea of inputs to functions. The magic part is where we declare the variable from the heading and fill in the caller's value. But after that, there's no more magic – `n` is just a regular variable.

Just to be sure, here's the function being used, nothing special:

```
void Start() {
  showAsWord(1); // "one"
  showAsWord(0); // "zero"
  showAsWord(2); // "two"
  showAsWord(7); // "many"
}
```

Notice how there's no `n` in this part. We never need one. When we call `showAsWord(1);`, the computer creates local `n` inside the `showAsWord` function, set to 1. And as we know, local variables are invisible to everyone else and never interfere with anyone else either. `n` is for the function's personal use only, which is perfect.

Here's a similar example, written in steps this time. I want a function that tells me whether water is `"ice"`, `"water"` or `"steam"` at a certain temperature. The first part is easy – I'll pretend we have variable `temp` and write a cascading `if`:

```
if(temp<=32) print("ice");
else if(temp<212) print("water");
else print("steam");
```

That's not a function – just regular code – but I figured I need do know how to actually do it before making it into a function.

It needs a name – I'll pick `printWaterState`. I need to decide ahead of time whether the input is an int or float. Temperatures could have decimals, so float. In the code above I already picked `temp` for the variable name. So the heading looks like `void printWaterState(float temp)`. The complete function is:

```
void printWaterState( float temp ) {
  if(temp<=32) print("ice");
  else if(temp<212) print("water");
  else print("steam");
}
```

The same magic happens – when someone calls `printWaterState(60.0f);` the 60 is automatically copied into our `temp` variable.

To be complete, some function calls:

```
void Start() {
  printWaterState(150.6f); // "water"
  printWaterState(7); // "ice" (auto convert int to float works as normal)
}
```

The second one might seem like cheating, but `float n=7;` is legal. The same way, the computer gladly converts 7 to `7.0f` before auto-assigning it to `temp`.

Here's a 1-line, 1-input function to resize a Cube. If you remember, there's an x, y and z scale. Changing just one makes a Cube wider, taller or deeper. Setting all three to the same number makes it evenly grow or shrink. But that's a pain to remember, so I want `resize(1.5f);` to handle all three dimensions for me. It's another front-end function:

```
// sample uses:
resize(0.2f); // tiny
resize(5.0f); // 5x5x5: over half the screen

void resize(float newSize) {
  transform.localScale = new Vector3(newSize, newSize, newSize);
}
```

Think of this in two parts: `transform.localScale = new Vector3(newSize, newSize, newSize);` is a regular command to make you be as big as variable `newSize` says. We've used it before. The new part is how function magic declares `newSize` and gives it a value.

This is a really nice use of the function-as-new-command idea. `resize(0.2f);` is simple and obvious. The new size we want is right there in the function call.

Here's the same idea, but with colors. A fun fact: if you set red, green and blue all to the same value, you get a "grey scale" color – black, white, or any shade of gray in-between. This shortcut color function will do that for us:

```
void setGreyScale(float bright) {
  GetComponent<Renderer>().material.color =
         new Color(bright, bright, bright);
}
```

`setGreyScale(0.25f);` would give a grey that barely stood out from a black background; `setGreyScale(0.9f);` would turn us almost white.

Like any other functions, functions with inputs can also do math. Suppose we have a lot of artist-programmers who prefer the standard 0-255 `int`s for color values. We can write a function that turns us red, using 0-255 for how red we should be:

```
void setRed(int amt) {
  float r = amt/255.0f; // 255.0f must be a float, to keep the fraction
  GetComponent<Renderer>().material.color = new Color(r, 0, 0);
}
```

Now `setRed(80);` turns us a darkish red, and `setRed(240);` is almost maxed-out red (if you've used image programs, those numbers would seem completely natural.)

Here's a longer example with a `string` input. It's nothing special, and no new rules – I wanted to show a longer function, with more programming in it.

I want the function to move us into a corner of the screen, using `"UL"`, for upper-right, then `"LL"`, `"UR"`, and `"LR"` for the other corners. I'll call the function `moveToCorner`. To make the rest of the heading: the input is a `string` and I'll call it `cornerName`. So the heading is like this:

```
void moveToCorner( string cornerName ) {
```

My plan is to first figure out the left&right numbers for x, and the top&bottom numbers for y. Then I'll check the input and move us to whichever corner:

```
void moveToCorner( string cornerName ) {
  // figure L/R and U/D numbers ahead of time:
  float left=-5.0f, right=5.0f;
  float down=-3.0f, up=3.5f;

  float xx, yy; // set these to the pos we want

  if(cornerName=="UL") { yy=up; xx=left; }
  else if(cornerName=="LL") { yy=down; xx=left; }
  else if(cornerName=="UR") { yy=up; xx=right; }
  else if(cornerName=="LR") { yy=down; xx=right; }
  else { xx=0; yy=0; print("bad cornerName"); }

  transform.position = new Vector3( xx, yy, 0);
}
```

I like it because it starts by going about normal function business – declaring and setting some local variables, then declaring `xx` and `yy` as temps to hold where we want to go. And then, finally, it looks at `cornerName`, which is the input.

For fun, I decided that bad inputs, like `"topEast"`, just put us in the middle and print an error message.

To finish it, some function calls:

```
moveToCorner("LL"); // puts us in lower left
moveToCorner("ur"); // Oops! puts us at 00 -- string == is case-sensitive
string cWord = "LR";
moveToCorner(cWord); // put us in lower-right corner
moveToCorner("L"+"R"); // same
```

The last two show that function calls can use the same math inside the parens as everyone else. It's no different than `print(cWord);` or `print("L"+"R");` (more on that later.)

Of course, running these in a row like this would put us where-ever the last one said.

This is also another example of why `string`s often make bad inputs. It's too easy to mess up upper/lower case, or misspell them, and the pop-up can't give you any help.

12.2 Rules

Most of these are just formal ways of saying stuff you already know, but some of it might be new. Plus it's just nice to see it written up:

- In a function call, the input goes inside the parens, the same as `print("a");`. Our functions from before, like `applyPos()`, also follow this rule – the inputs (none) are inside the parens.
- The input when you call a function is called an **argument**. The input the function asks for is called a **parameter**. In `showAsWord(1);`, 1 is the argument for the parameter `int n`.

  These words only matter because the computer uses them, a lot, so they're good to know. They don't really relate to normal english usage – you jmemorize them after a while. They often get shortened to `arg` and **parm** (parm isn't even the first four letters of parameter, but oh well.)
- The argument (the thing in the function call parens) can be anything that counts as the correct type. `showAsWord(1+b*6)` is fine (if `b` is an `int`.) It will fix any wrong types that `=` would fix, so 3 is fine for a `float` parameter.
- Function calls have to have the exact number and type of arguments. These are all errors: `showAsWord();`, `showAsWord(1.3f);`, `showAsWord("1");`.

  Another way of seeing this, you can only run the function if you give it usable inputs.
- To make a function have a parameter, declare a variable in the heading, like `void A(string w)`. Can use any type, and any name.

- After the function starts, the parameter counts as an ordinary variable. You can add to it or change it like anything else.

Again, just a summation of the rules from the examples, but I tried to use the official words. Plenty of rules are like this – pretty intimidating until you understand them.

12.3 More rule-showing examples

The name of the parameter really is just any name you pick. My original `showAsWord` used `n` as the parameter name, but anything else would work. This looks worse, but works fine – all that matters is the code is reading the input variable:

```
void showAsWord(int xyz) {
  if(xyz<=0) print("zero");
  else if(xyz==1) print("one");
  ...
```

The style rules for parameter names are the same as for any variables: try to pick one that says what it does, or just something boring like `n`, `w`, or `x` if you can't think of a good name.

Once the function is running, the parameters count as regular variables. You're allowed to change or copy them if you like. In this example, I decided to "fix" the input before using it:

```
void showAsWord(int n) {
  if(n<0) n=0; // <-- can change input n
  if(n==0) print("zero");
  else if(n==1) print("one");
  ...
}
```

If I call `showAsWord(-4);`, it starts with `n=-4;`. But then the first line changes `n` to 0, which is legal, and seems fine.

After they get the inputs, those variables don't have any special "input goodness." In this example, I picked a long variable name for the heading – hoping it makes it easier for people to see what the function does. But I want to use a shorter name inside, so I copy it to `x`:

```
void showAsWord(int numToPrintAsAWord) {
  int x = numToPrintAsAWord; // copy to a shorter-name var
  if(x<=0) print("zero");
  else if(x==1) print("one");
  ...
}
```

You're even allowed to ignore or destroy the inputs if you want. In this example, I want to test how the function works on 29. This temporarily ruins the function, but it's legal, and is fine for testing:

```
void showAsWord(int n) {
  n=29; // <- testing line
  if(n==0) ...
  ...
}
```

This one simply ignores the input, so is useless, but it's also legal. And also the kind of thing people write for testing:

```
void showAsWord(int n) {
  print("NUMBER");
}
```

Sometimes we call that a *Stub*. The idea is: if you want to finish a function later, as least make the heading correct. That way everyone can write their parts using it.

The parameter name really is a local variable, only inside that function. In this example, Start can also have an `n`:

```
void showAsWord( int n ) { ... }

void Start() {
  int n=47; // <- a different n
  showAsWord(1); // "one"
  print("n="+n); // our n is still 47
}
```

Start gets to have an `n` with no surprises. As usual, the `n` inside the function is completely different. It's just an unimportant coincidence the names happen to be the same.

12.4 Fun errors

I mentioned some of these before, but here are almost all the function errors and error messages. Same idea as before – if we cause them on purpose and read them, they won't surprise us when we see them for real:

- Input when we shouldn't have one: `applyPos("carrot");` gives error: *No overload for method 'applyPos' takes 1 arguments.* This is just telling us that `applyPos` (from last chapter) doesn't take any inputs. Method is another word for function and remember argument is the technical term for the input when you call it.

- Missing an input is the same error: `showAsWord()` says: *No overload for method 'showAsWord' takes 0 arguments.*
- The wrong `type` of input: `showAsWord("hat")` gives *The best overloaded method match for 'testA.showAsWord(int)' has some invalid arguments.* It's just showing you the function, and how it wants an `int`. A funny thing is, other errors like this would say "cannot convert 'hat' from string to int." This error is more like "here's how it should look – yours doesn't match."
- Adding the type in the call: `numAsWord(int 4);` is an error. It says *unexpected symbol 'int'.* It's easy to get the function call and definition confused. The definition has to give the type, the call just has to match it. Another way to remember is you wouldn't write: `n = int 4;`.
- Declaring the parameter also as a local variable:

  ```
  void A(int n) {
    int n=4;
    print("input is " + n);
  }
  ```

  This gives you: *A local variable named 'n' cannot be declared in this scope because it would give a different meaning to 'n', which is already used in a 'parent or current' scope to denote something else.* You can't have two local variables both named `n`.

  The error looks so odd because parameters aren't exactly local variables.

12.5 Multiple Inputs

Functions can have 2 or more inputs. There are only a few extra rules for this, which aren't complicated. Here's an example of a 3-input mad-lib function and some sample calls:

```
void sayStory(string animal, string animalMood, int howMany) {
  string w;
  if(howMany==1) w = "I saw a " + animal + ". It was "+animalMood+".";
  else w = "I saw " + howMany + " " + animal+"s. They were "+animalMood+".";
  print(w);
}

void Start() {
  sayStory("cow", "lazy", 1);
  // I saw a cow. It was lazy.
  sayStory("whale", "happy", 17);
  // I saw 17 whales. They were happy.
}
```

The two new rules are: put commas between them (both places: the function call, and the definition.) Rule two: when the function is called, match args to parms exactly in order.

If you remember, that clunky `applyColor();` used global variables. We can finally make the good version, where we directly give it the color values:

```
void colorChange(float r, float g, float b) {
  GetComponent<Renderer>().material.color = new Color(r,g,b);
}
```

To make orange we can use `colorChange(1, 0.5f, 0);`. This is a good solid use of the "make new commands" idea of functions – one short all-in-one command to fully set your color.

Here's one for resizing that does a little extra thinking for us. A basic resize ends with `=new Vector3(wide, tall, 1);`. The last slot, `z` is almost always 1 (it's deepness, which we can't really see,) but we still have to enter it.

We can write a function that "knows" our rules, filling in that 1 automatically:

```
void resize(float wide, float tall) {
  transform.localScale = new Vector3( wide, tall, 1);
}
```

We'd use it like `resize(2, 0.5f);` to make us wide and short. We don't have to type the longer localScale line and we don't have to think about the thickness. That's a good deal.

Just to show `float` and `string` inputs together, here's an odd resize. It's a little fakey: I want to send the overall size, then a letter whether it's a square (`"S"`,) a tall rectangle (`"T"`) or a wide rectangle (`"W"`):

```
void shapeResize(string shape, float sz) {
  float wide=sz, tall=sz; // so far, may change if not square

  if(shape=="W") { tall/=2; } // wide - chop height in half
  else if(shape=="T") { wide/=2; } // tall - chop width in half
  // else square, so what we have is fine

  transform.localScale = new Vector3( wide, tall, 1);
}

// sample calls:
shapeResize( "W", 5); // giant sideways bar
shapeResize( "T", 1); // smallish "standing" block
shapeResize( "S", 0.2f); // tiny square
```

Like I wrote, this is a bit silly – I just wanted to show we could use two different types of inputs.

12.5.1 More errors

Arguments really are always matched to parameters in the exact order. That means we can now have some new, fun input out-of-order errors. `sayStory` takes inputs in order *(string, string, int)*. Putting the `int` somewhere else is an error:

```
sayStory(5, "dog", "barky"); // ERROR. Wrong order
```

The computer tries to match 5 with `string animal`. It can't, so it's an error.

In our minds, we accidentally put 5 in front, instead of at the end. But the computer won't even think about different orders. As soon as it sees that the 5 doesn't match, it stops checking and gives two errors: *The best overloaded method . . . has some invalid arguments* and then more detail: *Argument #1 cannot convert int expression to type string.*

You have to give the exact number of inputs. Too many or too few is an error. Both of these give a *No overload for method takes __ arguments* error:

```
// sayStory("cow", "lazy"); // ERROR. missing one
// sayStory("cow", "lazy", 5, 7); // ERROR. one extra
```

You might think the second one should just skip the extra 7. We decided it's an error so you have a chance to clear it up. Maybe you meant to write 5+7?

Of course, the computer only checks the types. It has no way to make sure the first input is an animal. If you flip the order of two `string`s, it won't cause an error, but will probably be wrong:

```
  sayStory("sleepy", "chicken", 3); // Ooops:
  // I saw 3 sleepys. They were chicken.
```

You aren't allowed to use the multiple variable shortcut for parameters. You have to list out each type/name pair. This is an error:

```
void setPos(float x, y) { // error, missing second float
  transform.position = new Vector3(x,y,0);
}
```

The computer can't tell whether `y` is supposed to be a float, or if you forgot to put the type in front. My compiler tells me *identifier expected*, which isn't very helpful, but at least it tells me the line.

The correct version is `void setPos(float x, float y)`.