Chapter 14

Functions, part III - return

14.1 Introduction

You may have noticed that several of the old functions computed something and then just printed the answer: the ice/water/steam function, or finding the word for a number, or even the random New Clarkville town generator. Printing the answer is just terrible. We want them to *be* the answer. The way 2+3 is 5. we want `waterState(25)` to just be `ice`.

Our goal is to be able to do things like this:

```
string w=waterState(20);
print("The lake is "+w); // The lake is ice

string townName=getRandomTown();
print(townName+" was overrun by rats"+); // Tomaberg was overrun by rats
```

14.2 return values

Our old functions did things. They went all by themselves on a line, like `applyColor(1.0f, 0.5, 0);`. applyColor doesn't have an answer or a value, and we wouldn't want it to.

Our new functions will be part of longer lines, and will work like math:

```
int n = 2+3; // 2+3 turns into 5
string w = waterState(25); // waterState(25) turns into "ice"
```

We already have the rule that functions come back to where you called them. Now that rule means we can pause midway through a line. Above, `string w=` is waiting for `waterState(25)` to come back with the result.

The new rule for this is the `return` statement. It quits the function and returns the answer. Here's the improved `waterState` function, using a return:

```
string waterState(float tempInDegs) {
  string w;
  if(tempInDegs<=32) w="ice";
  else if(tempInDegs<212) w="water";
  else w="steam";

  return w;
}
```

The print statement at the end is gone, Instead we have `return w;`. The only thing it does is send back the answer.

Let's to a quick run-through. `s=waterState(31);` marks where we came from. We're midway through the line, with `s=` waiting. We copy 31, run waterState, return `"ice"` and pop back. Our program now counts as `s="ice";` and keeps running from there.

There's one more change. The very first thing in the function, before the name, is the type of the return value. It replaces the `void`. In this case, it clearly has to be `string` since the 3 possible answers are strings.

Those are the only changes: put the return type in front, and use a `return` statement at the end.

The turn-int-into-a-word function can be rewritten. It's not very exciting, except for how it now works properly. `string w=intToString(2)` now gets `"two"`:

```
string intToString(int n) { // <- string return value
  string numWord="unknown";
  if(n==0) numWord="zero"; // this is code from the if chapter,
  if(n==1) numWord="one"; // nothing special about it
  if(n==2) numWord="two";
  // ... until 9
  return numWord;
}
```

The 90/80/70/60/50 letter grade `if` is pretty much the same. The answer is one letter, so a function can return it:

```
string letterGrade(int n) {
  string gr;
  if(n>=90) gr="A";
  else if(n>=80) gr="B"; // copied straight from the...
  else if(n>=70) gr="C"; // ...chapter on cascading ifs
  else if(n>=60) gr="D";
  else gr="F";

  return gr;
}
```

`string grade1 = letterGrade(72);`. is now allowed. `grade1` is a B.

We think of these as "pure" functions – basically like math. They get all input from parameters, and don't change anything in the program. They just give the answer back and let the main program use it however it likes. These are considered the best type of function.

A fun thing is to write out some real math functions. Some of these are built-in but they're fun to write anyway.

Finding the largest of two numbers is usually called `max`. This function is the same `if/else` from a few chapters ago, with a `return`:

```
float max(float a, float b) { // <- answer will be a float
  float ans;
  if(a>b) ans=a;
  else ans=b;
  return ans; // <- no printing. return the answer
}
```

If we have `float q=max(3,8);`, then `q` will be 8.

This is our first `float` return-type. It makes sense, since the answer is one of the inputs, which are floats. It's also our first returning function with two inputs. A `return` can only give one answer, but a function can always have all of the inputs it needs.

Absolute value (`abs`) is a fun, short function:

```
float abs(float a) {
  float ans=a;
  if(a<0) ans=a*-1;  // if it was negative, make it positive
  return ans;
}
```

`abs(6);` is 6 and `abs(-12)` is 12. This function barely does anything – it's one `if` that we could write ourselves. But calling `abs(n)` is shorter and easier to read, so this is a good function.

14.3 Using value-returning functions

In our minds, value-returning functions work like math. Since they don't do anything, we can run them whenever we want, saving the answer for later. This prints comments about 2 grades:

```
string gw1=letterGrade(g1), gw2=letterGrade(g2);

if(gw1=="F") print("Arrg! I can't read any more!");
```

```
else {
  if(gw2=="F") print("Arrg! So close.");
  else print("Yay! Passing!");
}
```

We might not even use `gw2`, but it seems easier to compute them both at the top, the same as we like to do with all of our math.

This uses `max` to figure out if anyone is old enough to rent a car:

```
int oldest = max(amyAge, bradAge);
if(oldest>=18) print("Can rent a car");
```

We're allowed to use these functions inside of larger math. These two add waterState to another string:

```
print("Look, it's some " + waterState(10)); // Look, it's some ice

string w2 = waterState(270) +" world"; // steam world

print("H2O at "+ t1 + " degrees is " + waterState(t1) );
```

There aren't any special rules here. `waterState(270)` always becomes `"steam"`, no matter where it is. `4+waterState(0)+9` is `4ice9`, and so on.

These use our new math functions inside a larger equation:

```
n = max(3,6) + abs(-2); // 8 (6+2)
n = 2*max(-12,3)+100; // 106 (2*3+100)
```

Using the same function several times works just fine. The computer calls them in order, replacing values as it goes:

```
n = max(3,7) + max(1,-6) + max(2,2); // 10 (7 plus 1 plus 2)
n = abs(3) * abs(-5); // 15 (3 times 5)

n = max(10,2) * abs(-3) + abs(5); // 35 (10*3 + 5)
```

Here's a picture of the steps in the last one:

```
n = max(10,2) * abs(-3) + abs(5);
// makes three function calls:
// n =  10    * abs(-3) + abs(5);
// n =  10    *    3   +  abs(5);
// n =  10    *    3   +    5;
```

To really check this out, we can add a `print` inside of `max`. That's a terrible idea for real, but good for testing:

```
float max( float a, float b ) {
  int ans;
  if(a>b) ans=a;  else ans=b;
  print("In max. a="+a+" b="+b+" answer="+ans); // testing!
  return ans;
}
```

Now we can run a line with two `max` function calls and watch them run:

```
print( max(2,6) + max(1,1) );
// In max. a=2 b=6 answer=6
// In max. a=1 b=1 answer=1
// 7  <-- the print we wrote
```

A fun and completely legitimate trick is using functions inside of an `if`. Some examples:

```
if( max(a,b)>10) ) print("both must be 10 or less");

if( waterState(temp) == "ice" ) print("can walk on it");

if( waterState(t1)=="ice" || waterState(t2)=="ice")
  print("at least one is ice");
```

The neat thing is, it's still not a new rule. Same as before, functions are called, run, and count as their answer. Then the line continues.

14.4 More math functions

Here are some more examples of math-like functions.

`sign` returns -1, 0 or +1. The idea is, it really says negative, positive or zero, but -1/0/+1 is the best way to say that. As usual, the inside is completely boring, except for the `return`:

```
int sign(float a) {
  int ans;
  if(a<0) ans=-1;
  else if(a>0) ans=+1;
  else ans=0;
  return ans;
}
```

For examples `sign(6)` is 1, `sign(-0.43f)` is -1.

Previously we saw a pair of `if`'s to make sure a number is between 1 and 10, or some other range. We can put that in a function:

```
float clamp(int n, int min, int max) {
  float ans=n; // so far
  if(n<min) ans=min;
  else if(n>max) ans=max;
  return ans;
}
```

A common use would be `cats=clamp(cats,1,10);`. If cats was outside of 1-10, that line fixes it. If cats was 7, the answer is 7 and it changes cats to 7, which seems odd, but it works.

It feels so much like a real math command that people use the word clamp to mean "adjust into a range"

We often need to get a percent from one number to another. For example 10% of the way from 4 to 7. In math that's a linear interpolation, abbreviated lerp. The math is easy, but we often write it as a function:

```
float lerp(float start, float end, float pct) {
  // NOTE: pct should 0 to 1. 0.3 is 30%
  float gap=end-start;
  float distFromStart = gap*pct; // ex: 10% of the way
  float ans = start+distFromStart;
  return ans;
}
```

A sample use, our screen from the Cube examples goes from -7 to 7. To put something 30% of the way across, we can use `float xx=lerp(-7, 7, 0.3f);`.

Squaring a number is super-easy, just one line, and a new trick:

```
float square(float n) { return n*n; }
```

Notice how we did math after the `return`. That's completely legal and works the normal way. We usually think it looks better putting the answer into a variable, unless the whole thing is one equation like here.

To see why this might be useful, consider squaring x+y. `square(x+y)` looks nicer than `(x+y)*(x+y)`.

We aren't limited to real math. Suppose your magical manna points are your Intelligence stat or Wisdom stat, which-ever is higher, plus half of the other one, but at least 5. We can write that:

```
int mannaPoints(int intell, int wis) {
  int ans;
  if(intell>wis) ans=intell+wis/2;
  else ans=wis+intell/2;
  if(ans<5) ans=5;
  return ans;
}
```

Even though this is totally made up, it works the same as any other math function. `int myManna = mannaPoints(10,5);` gives 12. `mannaPoints(1,2);` is 5.

In the old days if you wanted to know how a game worked, you looked through the code for things like this. Some places called the mannaPoints function, you searched for it, and voila – now you know the formula the game uses for manna.

14.4.1 Extra return rules

The `return` statement jumps out of the function even if it's not at the end. The function quits right then – any lines after the `return` are skipped. That should make perfect sense – once you have the answer, the function is done.

You're allowed to have as many `return` statements as you need, and they can have any kind of constant or equation after them. Here's `intTostring` rewritten using the "return as soon as you know it" method:

```
string intToString(int n) {
  if(n==0) return "zero";
  if(n==1) return "one";
  if(n==2) return "two";
  return "three";
}
```

That `return "three"` at the end looks mighty suspicious. There's no `else` in front of it, so it looks like the answer is always `"three"`. But the function works by abusing the `return` rules. If `n` is 0, the function quits right then with answer `"zero"`, and never gets past line one. It only gets to `return "three"` if the `if`s were all false.

Here's `max` rewritten to have return-abuse:

```
float max(float a, float b) {
  if(a>b) return a;
  else return b;
}
```

It doesn't even have a final `return` by itself, since the if/else always does it.

You can use the return-early trick for testing. Here we adding an extra line to temporarily make `intToString` always say NUMBER_WORD:

```
// abusing return for testing:
string intToString(int n) {
  return "NUMBER_WORD"; // temporary for testing
  // this is all skipped:
  if(n==0) return "zero";
```

```
  if(n==1) return "one";
  if(n==2) return "two";
  return "three";
}
```

Using the trick for mannaPoints shows why we have to be careful with it. Our answer may not be completely done:

```
int mannaPoints(int intell, int wis) {
  int ans;
  if(intell>wis) return intell+wis/2;
  else return wis+intell/2;

  // oh, no! Those returns skipped the "at least 5" rule
  if(ans<5) ans=5;
  return ans;
}
```

14.4.2 Optional empty return

If you have a function with no return type (it has `void` in front,) you're allowed to use just `return;` (with nothing after it) to jump back. You can put it at the end, for fun, and can also use return-from-middle.

Here's just a decorative end `return`. It's a little like writing The End:

```
void reset() {
  x=-7; y=0; speed=0.05f;
  return; // not needed, but looks pretty
}
```

Here's the early-return trick on a wrap-around function. It uses the `return` to quit and not wrap if it doesn't need to:

```
void wrapAround() {
  // assuming x is a global
  if(x<7) return; // not too big, so just quit
  x=-7;
}
```

You can even use return-from-middle in `Start` or `Update` – they're functions with `void` return values. Here's working move&wrap code with return-from-middle:

```
public float x=-7;

void Update() {
  x=x+0.1f;
```

```
  transform.position = new Vector3(x,0,0);

  if(x<7) return;
  x=-7;
}
```

Normally, `Update` quits when it gets to the end. Quitting early does the same thing, only faster. Either way, the "auto-run over and over" rule still happens.

A use of this trick is to crudely comment out code. This causes Update to do nothing:

```
void Update() {
  return; // for testing. update quits here
  x=x+0.1f;
  ...
}
```

Another legitimate use might when something is killed. We can do death stuff to it, then `return` to make sure nothing else happens:

```
void Update() {
  ...
  if(health<=0) {
    // set color to black
    // do other death stuff
    // now prevent the rest of Update from changing color or position:
    return;
  }
  ...
}
```

Obviously return-early can cause lots of confusion if used too much. Your update might have a line adding to `x`, but `x` is not changing. After an hour you spot the `return` up above. The compiler gives a warning when a naked return cuts off the rest of a function, but not for one in an `if`.

14.4.3 `return` can use math

Sometimes it looks nicer to put equations after a `return`. Like anything else, the computer works them out first – `return 1+3;` is the same as `return 4;`.

Some examples. This `abs` use two returns, one of them using math:

```
float abs(float a) {
  if(a<0) return -1*a; // could also use -a
  return a;
}
```

This redo of `lerp` does the math in the return line. As usual, it does the math first, then the `return`:

```
float lerp(float start, float end, float pct) {
  float distBetween = end-start;
  return start + distBetween*pct;
}
```

This mad-lib sentence maker builds a giant string in the `return`:

```
string walk1(string animal, string place) {
  string action="walk";
  if(animal=="fish" || animal=="dolphin") action="swim";
  return "My "+animal+" and I went for a "+action+" in the "+place;
}
```

14.4.4 Errors

If you say you're going to return something of a certain type, you have to. And it you say you won't return anything (by using `void`,) you can't. That seems pretty obvious – a function breaking that rule won't even make any sense. But there are some funny cases and seeing the error messages is helpful:

It's easy to write a function that usually returns something, but for some values it just reaches the end without ever hitting a `return`. For example this gives compiler error: *not all code paths return a value.* because of 10 through 100:

```
string badFunction(int num) {
  if(num>100) return "big";
  if(num<10) return "small";
  // return ""; // this would fix the error
}
```

It's a compile error – you can't even run the program. Maybe you just forgot. Or maybe you know the input can never be 10-100. The fix is adding a dummy return at the end. Sometimes I'll use something like `return "BAD VAL";`.

This one's a little tricker. We can see it always `return`'s something, but the computer can't tell. It incorrectly gives the *not all code paths return a value* error:

```
string sayNum2(int num) {
  if(num<0 || num>1) return "bad";
  if(num==0) return "zero";
  if(num==1) return "one";
}
```

Adding `return ""; // can't get here` as the last line will make the compiler happy. Or, rewrite it as a cascading if/else.

When using `return;` to quit early from a no-answer function, it's easy to accidentally write `return 0;`. Zero seems like nothing, but it's not nothing – it's a number that means nothing (who knew computer science was so philosophical?):

```
void moveAndWrap(float n) {
  n+=0.1f;
  if(n<7) return 0; // ERROR, use "return;"
  n=-7;
}
```

Returning the wrong type is clearly an error. But the computer will do any conversions that an `=` would do. `return 3;` is fine when you need a float.

Functions starting with `void` don't have answers. Using them inside of math gives an error. This function prints a string, but it has no official answer, so trying to assign from it won't work:

```
void sayMoo() { print("mooo"); }

  string w = sayMoo(); // ERROR
```

It says: *Cannot implicitly convert type void to int.* That's not quite right, but it lets you know where the problem is.

Oddly, it's *not* an error to ignore a return value. This is legal, but useless:

```
max(3,15); // not an error. Answer is 15, but we don't use it for anything
```

This might be useful for testing. You might have a bunch of printing lines inside and just want to run it to check what they say.