Chapter 3

Variables, computer math

The next thing we want to know is how to use variables. We're going to be able to do things like `x=3; y=5; print( x+y );`. But before we do that, we have to know about **types**. As humans, it makes sense to divide things into words or numbers. The computer is much more strict about it, and has formal rules.

We're going to use three **types** for now: words, whole numbers, and decimal numbers. The official names are `string`, `int` and `float`.

3.0.1 Literals

Constant values in the program, like 3 or `"yam"` are officially called **literals**. Whenever the computer sees a literal, it first decides what **type** it is, based on how it was entered.

`string` literals are what we've seen before. Anything between double-quotes is officially a string. Some more examples of string literals:

```
"duck"  "12"  "12.6"  "3-1"  "Start()"  "}"  ""
```

We knew `"duck"` was a string, from before. We've also seen tricks like how a number or an equation in quotes is a string. To review, as far as the computer cares, `"12"` is no different than `"ab"` – they're both just two-letter words. `"3-1"` is just a three-letter word. The human who typed it clearly put double quotes around it. If they didn't want it to be a string, they wouldn't have done that.

`"Start()"` and `"}"` are also the same tricks as before. They look a little like instructions, but they're in quotes, so are strings.

The last one, `""` is a little special. It's a string with zero things in it. It's like a zero, but for words. It's usually called the *empty string*. You don't need it that often, but you do need it sometimes.

Non-decimal plus/minus numbers are officially called **integers**. The official computer word for them is `int`. Here are some int literals, nothing special:

```
12   4934002   0   -46   +58
```

Some boring rules: There is never a comma for the thousands place, millions place or any other place. One thousand is just 1000. A plus in front for positive ints is optional: 3 and +3 are the same. Most people leave off the starting +.

`float` literals have decimal points, and always have a lower-case f after them. Here are some `float` literals:

```
0.1f  30.1f  -6.001f  0.00000045f  3.0f  -4.0f  0.0f  47000.0f  3f
```

The strangest thing is that they really do all end with an `f`. That seems like a pretty silly rule. There's a reason for it – more on that at the end of the section.

The first four are basic decimal point numbers. Like school math, they can have just the decimal, or before and after the decimal, be negative, or be very small.

The next four are interesting, since they look like integers. This is where the **type** rules matter. 3 is the integer version of three, and 3.0f is the float version of 3. Put another way: if you have the number three, and want to put it in the computer, you have to decide to enter it as an `int` or as a `float`.

The last one, `3f` is just another way to write `3.0f`. Most people write the "point-oh", to make it more obvious, but it's optional.

3.1 Math with types

A way to get a feel for `types` is to see the rules about how math works with them. When the computer sees math, like two things being added together, the first thing it does is look at the **type** of the things being added.

When the computer sees a + symbol with strings, it combines them end-to-end. The official word for that is *concatenate*. Strings don't use any other math symbols. Some examples:

```
print( "a" + "b" ); // ab
print( "b" + "a" ); // ba
print("catcher" + "dog"); // catcherdog
```

Obviously, the order matters for string plus.

Like math, we can use as many plus's in a row as we want:

```
print( "a" + "b" + "c"); // abc
```

It doesn't know or care about what the strings mean. Below, it combines duck and bill to make a fun word, but it doesn't know `thehouse` should have a space. There's a space between "I" and "like" since I put one there, but it still doesn't know that "puppies" should have one:

```
print( "xxx" + "xx" ); // xxxxx

print( "duck" + "bill" ); // duckbill
print( "the" + "house" ); // thehouse
print("I like" + "puppies"); // I likepuppies
```

The rule "if it's in quotes, it's a string" applies here. These are things that look like numbers and equations, but are really strings:

```
print("2" + "7"); // 27
print("10" + "+0"); // 10+0
print( "1+2" + "3"); // 1+23
print( "4" + "5" + "6"); // 456
```

I like those, since they show how "it's a string" is the most important thing for the computer. In `"2"+"7"`, it wasn't tempted even a teeny bit to math-add them and get a 9.

The rule for number math is you get a result of the same type, `int` or `float`. For example `1.5f + 2.5f` gives a nice even 4, but it's float `4.0f` since it came from floats.

Division shows off the same-type rule. `13.0f/5.0f` gives the real answer, `2.6f`. But using integers, `13/5` gives an integer answer of just 2 (it drops the fraction.) Dividing two integers gives an integer answer. More examples:

```
print(  35.0f / 10.0f ); // 3.5
print( 35 / 10 ); // 3

print( 18 / 6 ); // 3 -- if it goes in evenly, we get the normal answer

print( 15 / 20 ); // 0 -- not a special rule. 20 goes into 15 zero times
print( 15.0f/20.0f ); // 0.75 -- back to real math
```

Dropping the fraction seems completely insane. We did that in second grade because we didn't know how to make decimals. We have the rule now because sometimes we want that option. If I have 13 people, I can make 2 5-person basketball teams with 3 people left over. I don't care that technically those extras make 0.6 of a team.

Looking at the division rule another way, if you want to get the "real" answer, divide using floats. If you ever want the "dropped fraction" answer, divide using ints.

The remainder in **int** division is completely gone. It isn't invisibly saved for later. Here are some fun examples showing that:

```
print( 1/2 + 1/2 ); // 0
```

Both parts are integer 1 divided by 2, which is is 0. Then we get 0+0 is 0. It's tricky because it's so simple. The computer doesn't look ahead or try to figure out what you really wanted.

```
print( 2/3 + 3/4 ); // 0
print( 9/10 + 9/10 ); // 0
```

These are both 0+0, for the same reason. Nine-tenths is a lot closer to 1 than to 0, but the rule is to drop the fraction, not to round. `9/10` is still 0.

This next one tries to trick you with white space. Division still goes first, so this is really 1+1:

```
print( 6/ 5+5 /4 ); // 2
```

Then here's an ugly example you wouldn't use for real. Division goes from left to right, so we get `(10/3)` = 3 first, then `3/2` is 1:

```
print( 10/3/2 ); // 1
```

But `10/(3/2)` is 10. The `(3/2)` is first, which makes 1, and `10/1` is 10.

3.1.1 mixed types

The rules so far are what to do with two things of the same type. We need some rules for what happens when you have two things of different types. For example: `"goat" + 7` and `7.0f / 4`.

For the three types we have now, there are two mixed-type rules (the first two are the rules):

- Adding a string to a number (int or float) converts the number to a string. For example, `"goat" + 7` is `"goat7"`.

  floats are printed in a nicer format, since humans will be reading them: the `f` isn't printed, and `.0` is left off. `"a"+4.0f` is `"a4"`

- Mixing an `int` and `float` causes the `int` to be **promoted** to a float, by having a `.0f` temporarily added to the end. For example, `5/4.0f` is `1.25f`.

- It doesn't matter which comes first. `5+"2"` is still a string and an int, so the 5 gets turned into `"5"`. `9/2.0f` and `9.0f/2` are both 4.5, since they both have a mixed float and int.

- Normal precedence rules apply, and conversions don't happen until they need to. For example, `1+2+"glow"` is `"3glow"` (since 1+2 goes first.) But `"glow"+1+2` is `glow12` (since `"glow"+1` goes first.)

Examples using these rules are fun, and give more chances to see how important the type is to the computer:

```
print( 3/2 + 1.5f ); // 1 + 1.5f = 2.5f

print( (0.0f + 6) / 4 ); // 1.5f

print( 12/8/2.0f ); // 1 / 2.0f = 0.5f

print( "abc" + 1 + 2 ); // abc12
print( "abc" + (1 + 2) ); // abc3
print("abc" + 2 * 3 ); // abc6
```

The steps for all of these are the same: figure out which math symbol goes first, find the types of the two things around it, fix mixed-type problems if you need to, and do the math. Then repeat until it's done.

3.2 Variables, declarations, simple assignment

Computers can use x and y somewhat like in algebra. But first, every variable needs to have an official Type – `string`, `int` or `float` Before you can use any variable, you have to say what **type** it will permanently hold.

The command to say "this variable is this type" is called a **declaration** (that word will sometimes appear in error messages, so good to know it.) The rule for **declaring** a variable is simple: write any type, a space, then the variable name. Ex's:

```
int x;
float y;
string z;
```

Those lines declare `x` as an `int`, declare `y` as a `float` and `z` as a `string`. Notice the semi-colons – declarations are statements, so need the ending semi-colon.

It often makes sense to think of the declaration as creating the variable, or setting it up. So the `int x;` line says "please make a little box for variable x, which can hold whole numbers."

Here are some rules for declaring. You can skim them, since you'll pick them up from examples later:

- You have to declare a variable before you can use it. If you write `x=6; int x;` the computer will complain that `x` wasn't declared. The real error is those lines should be flipped.
- The type is permanent. In the above, `x` is locked as an `int`. It can never hold `"cow"` or `3.0f`. Put another way, for every `x` in the program, you won't know the exact value ahead of time, but you know it will always be some integer.

- You can't declare a variable twice. `int x; string x;` gives an error about `string x;`. Even `int x; int x;` gives you an error on the second `int x;`.

- A shortcut: you can declare several variables of the same type, using the same declaration, by putting commas between them: `int a, b;` is a shortcut for `int a; int b;`. You can have as many as you like: `float w, x, y, z;` is fine. The comma is special for just this rule.

3.3 Identifiers

In a computer, a variable name can be just about anything. `x, y, n1, n2` are fine. But also `weight`, `minutes`, `phoneNumber`, `partsPerMillion`, `ppm` .... The technical term for "words we can make up" is **identifier**.

Rules for identifiers (variable names you can pick):

- Can use numbers, upper/lowercase letters, or underscores. No other symbols, no spaces. Can mix them in any order, except can't start with a number.

  These are bad names, but legal: `a1b2c3` `a_b` `_1` `___x___` `zero`. These are illegal: `4frog` (starts with a number,) `goat-home` (no dashes – the computer thinks this is goat minus home,) `tail Len` (no spaces.)

- Can't use a built-in computer word (can't make a variable named `string` or `void`.) In a modern editor, it will turn blue, so you'll know. You can vary the case, or include a built-in word. So `Float`, `float1` and `floater` are legal (but not very good) variable names.

- Bad spelling doesn't matter, since the name never means anything. `int totel;` (misspelled total) will work fine (but it might confuse people about what a `totel` is, so try to avoid it.)

3.3.1 Style

Variable names never really matter – you could use `a`, `b` .... But good variable names make the program a lot easier to read. Here are some notes:

Normal variables almost always start with a lower-case letter: `n` instead of `N`, or `count` instead of `Count`.

It's common for variable names to be a few words pushed together, like `dogWeight` or `catWhiskerLen`. Capitalizing the start of new words like that is sometimes called camel-casing.

The other way is with underscores for the spaces, like `cat_whisker_len`. But that's less common now.

There's an old style where you put the letter for the type in front, like `int iCats;` instead of just `cats`. That's called Hungarian notation – you can look it up. I mention it to show that people can take variable naming very seriously.

For just a quick name, `w` is usually a string. And if you see `i` and `j`, those are usually integers.

It's common to start with an underscore, like `float _len;`, to mark a variable as being sort-of hidden (we'll use that style in a much later chapter.)

Consistency is probably the most important. Don't use `catSize` along with `szDog` – pick one way or the other.

But it's not too difficult to change the names, once you learn Search and Replace.

3.4 Simple assignment, var use

We can put values in variables with lines like x=7;. The official name is an **assignment** statement. Once we have a value in a variable, we can use it like algebra: when the computer sees the variable, it replaces it with the value.

Here's a small program (inside of Start()) to show this:

```
int n;  n=8;

print( n );    // 8

print( n+1 ); // 9
print( n*2+1 ); // 17
print( 3+n+1 ); // 12
print( n*n ); // 64

print( "n" ); // n
print( "n+1" ); // n+1
```

The first looks up the value of `n`. The next four also look up `n`, then do math. Nothing special. `print(n*n);` is a little different – it looks up `n` twice, to get `8*8`.

The last two are showing how the "quotes make it a string" rule also applies to variables.

This next example has two variables with fun names:

```
int cats;
cats = 6;
int dogs;
dogs=4;
```

```
print("number of cat feet:");
print( cats*4 ); // 24

print("total number of pets:");
print( cats + dogs ); // 10
```

That's a more common sort of program. In school you might have gotten used to the style where you have to use `x` and `y`, and you say they stand for cats or whatever. It takes a while to get comfortable using regular words for variables. But once you do, it seems crazy *not* to use `cats` for how many cats you have.

An example with floats:

```
float length;  length = 3.5f;
float width;  width = 9.1f

print("area:");
print( length*width );

print("perimeter:" + (length*2 + width*2) );
```

In this one, `length` and `width` look like they have some special meaning, but they're just normal variables, no different from `x` and `y`.

I printed area on two lines just to make it easier to read. All on one line, the way I printed perimeter, is more common (notice how the extra inside parens are required, to force it to do all the math before doing the string add.)

An example with string variables. This one uses string math to combine the variable with some words:

```
string animal; animal = "mouse";
string name; name = "Henry";

print( "You are a " + animal );
print( "I think I will call you " + name );
```

This is still just math – the computer doesn't know what it's doing. For example, if the animal was `aarvark` it would print simply `"You are a aarvark`, without knowing to change `a` to `an`.

3.4.1 More assignment statement rules

In regular math, x=3 and 3=x are the same thing. In a program, = works a little differently. It says to change the variable on the left, into the thing on the right. This means the thing on the left must be a single variable. The right side

can be any type of math. The computer works the math, then puts the result in the variable, erasing the old value.

That's why we make a point of calling `=` an **assignment statement**.

In the example below, `x` is assigned using a formula. It's no different than a formula inside a `print`. Same as before, the computer works out the math and converts it to a single number before doing anything else:

```
int x;
x = 5+2*2;
print( x ); // 9
```

The same idea using strings. For real, no one would ever do this:

```
string w; w = "cow" + "bell";
print( w ); // cowbell
```

The computer won't do any work to "solve" equations. It will do regular math to work out the right side, but the left side is just where the answer goes. It has to be a single variable. These are errors:

```
int x;
5 = x; // ERROR. Left side must be a variable
x + 1 = 7; // ERROR. Left side must be a variable by itself

int y; y=5;
x + y = 8; // same error. Left side must be a variable all by itself
```

Like other statements, assignments run in order. In algebra, if we see a system of several equations, we know we should move them around, and pick one to solve first. Assignment statements are much simpler than that.

A good way to think of assignment statements, is each variable is a box with its current value. When you have something like `x=4+3*9/2;`, it does the math, `x` becomes 17, and the line is done. All that matters is `x` is now 17. The way it got there isn't important any more. An example:

```
int n1; n1 = 4+2; // 6
int n2; n2 = n1+3 // 6+3 = 9
int n3; n3 = 2*n2-4; // 2*9-4 = 14
```

Line one puts a 6 in `n1`. Line two simply looks up `n1` and sees 6, then adds 3. Line three simply looks up `n2` is 9, then finishes the math. No matter how complicated it was to compute a variable, afterwards it's just a simple number in a box.

Extra assignment statements erase the variable's old value. In this next example, the second equals "wins." `x` is 7 for just a little while, then changes to 2. The line `x=7;` is a waste, except as an example:

```
int x;
x=7;
x=2;
print( x ); // x is 2
```

But it still does things one at a time. This next one prints 7 then 2. Changing `x` to 2 won't go back and change the first print:

```
int x; x=7;
print( x );
x=2;
print( x );
// output is: 7, 2
```

One assignment statement that can fool you is `x=y;`. In Algebra, we're so used to `x=y` and `y=x` being the same thing. In computer assignment statements, they're completely different – copy `y` to `x`, or the reverse:

```
int x; x=4;
int y; y=20;
x=y; // now both are 20
print( x ); // 20 (it was changed to y)
print( y ); // 20
```

If we flipped it to `y=x;`, then it would print 4 both times:

```
int x; x=4;
int y; y=20;
y=x; //  <- this line is different. Now both are 4
print( x ); // 4
print( y ); // 4 (it was changed to x)
```

For the variable in front of the `=`, the old value never matters. If you have `x=3;`, no matter what `x` used to be, it's just 3 now. There's no undo, so the old value really doesn't matter.

Even strings work that way. It seems like the old length might matter, but it doesn't. Say you have `string w; w="abc";` then shrink it with `w="z";`. There's no left-over junk from when it had 3 letters – it's completely just `"z"`.

3.4.2 Initialization, unassigned vars

Variables have no starting value. `int x; print(x);` gives the error *Use of unassigned local variable x*. You get the same error with `int x; int y; y=x;`. Since `x` doesn't have a value yet, we can't copy it anywhere.

You might think `int`s should automatically start at 0. We made it an error on purpose to remind ourselves to always put something in there.

Giving a variable a starting value is often called **initializing** it. That's not an official term – if a variable got up to 100 and you reset it to 0, you could say that you're initializing it back to 0, or not. There's no special initialize statement. We just say the first assignment is initializing it.

In some circumstances, the computer will give a starting value. 0 for ints, `0.0f` for floats, and `""` for strings. There are rules for when it will and won't. But it's usually better not to count on that, and always give it an `x=0;` to be sure.

3.4.3 Computer math with variables; types

We know when the computer sees `3+"6"` it checks the types first, then decides what kind of `+` to use. It works the same way with variables. If we see `w+n` we check the types by looking how they were declared.

That's really why we have to declare variables. If `w` is a `string`, even though we won't know exactly what's in it, we know 100% that `w+n` uses string-plus.

Some examples of mixed math with variables:

In this first example, we divide two variables by 2. The `int` variable drops the fraction, the `float` doesn't:

```
int x;  x=7;
float y;  y=7.0f;

print( x/2 ); // 3 -- both are ints
print( y/2 ); // 3.5 -- float divided by int
```

Here's the same thing, except we divide by `x` and `y`:

```
int x;    x=2;
float y;  y=2.0f;

print( 5/x ); // 2 -- both are ints
print( 5/y ); // 2.5 -- int divided by float
```

This next one compares adding 3 to a `float` and `string` variable:

```
float a;   a=7.0f;
string b;  b="7.0f";

print( a+3 ); // 10
print( b+3 ); // 7.0f3 -- a 5-letter string
```

This next one uses addition where *both* sides are variables. As usual, it checks the type of both; if either is a string, it does string-combine. For real, we wouldn't have such terrible variable names as `a`, `b`, `c` and `d`. But they're good for a do-nothing example like this, which just shows the rules:

```
string a; string b; a="4"; b="6";
int c; int d;  c=4; d=6;

print( a+b ); // "46" -- both are strings, so concatenate
print( c+d ); // 10 -- regular addition

print( a+c ); // "44" -- string+int converts to string
print( d+a ); // "64" -- int+string converts to string
```

There aren't any new rules here. It's just the same old rule: "if one part of a `+` is a string, do string combine."

3.4.4 mixed-type assignments

Assigning the wrong type to a variable is another way to show how much the computer cares about declarations and types. If you try to assign the wrong type, the computer will either fix it, or give an error.

For all these, I'll use `int n; float f; string w;`.

Assigning an `int` to a `float` variable works. It uses the "promote" rule from earlier:

```
f = 4; // legal. counts as f=4.0f
print( f/3 ); // 1.3333 -- f is still a float
```

The interesting thing is `f` is always a `float`, since we declared it as one. In `f=4;`, there's no way the computer is going to try to turn `f` into an `int`.

No other wrong type-assignments work. `float` into `int` is an obvious problem – you'd have to drop the fraction. We decided it's better as an error:

```
n = 3.2f; // ERROR -- can't convert float to int
n = f; // same error, since f is a float variable
```

It's not even legal if the float ends in point-zero. That would be a confusing exception, and would look weird:

```
n = 3.0f; // error - cannot convert float to int
f = 3.0f; n=f; // same error
```

`string`s won't let you assign `int`s directly:

```
w = 5; // ERROR -- can't convert int to string
w = n; // same error
```

That seems funny, since we already know the computer can fix the 5 in `"ant"+5`. But we decided it's better to make you do it yourself. Here's the common, clever trick:

```
w = ""+5; // legal (but it would be easier to write just "5")
w = ""+n; // legal. This is the most common way
```

Even though `""` has zero letters, it still counts as a string, so it forces `n` to convert. It's a very sneaky use of the empty string.

Assignment also uses the "do things in order" rules. Of course, the assignment always goes last, and the computer won't look ahead to see what type it is. Fun examples:

```
f = 12/5; // 2.0f
f = 2.5f + 8/3; // 4.5 -- 2.5+2
```

In the first line, the computer doesn't care that 12/5 will be going into a `float`. It sees two integers, so rounds the result down to 2. Later, it converts into 2.0.

The next one is the same idea. `8/3` goes first, and is just a 2. It doesn't look ahead to the floats.

3.5 More assign and variable rules and examples:

One of the most common things we want to do is to "work on" a variable.

Typical things are adding 1 to a counter, or adding `n` to a running total. Or doubling a variable, or increasing it by 10%. We can do these with regular assignment statements, with a trick.

`x=x+1;` increases `x` by 1. The trick is to remember it's not algebra. The computer calculates the formula on the right, and puts it into `x` on the left. An example:

```
int x; x=4;
x=x+1;
print(x); // 5
x=x+1;
print(x); // 6
x=x+1;
print(x); // 7
```

You can read it as "the new value of `x` is the old value plus 1." The cool thing is that it's not a special rule. Suppose `x` is 6 right now. `x+1` is 7, so `x=x+1;` makes `x` be 7.

This is one of our main tricks. `x =` *x with some math* `;` is the basic way to change a variable.

The best way to try this is to write `x` on a piece of paper with a line under it, and think of it as a box. Then play computer.

`x=4;` puts a 4 in it. Then run `x=x+1;` in your head. That makes `x` be 5, so cross out the 4 and put a 5. You never have to think about the history of `x`. Whenever you want to know the current value, just look in the box. That's what the computer does.

You usually have to hand-run a few programs this way, to get a feel for the rules.

The same trick can add 5, subtract 1, or double us:

```
int x; x=4;
x=x+5; // 9

x=10;
x=x-1; // 9
x=x-1; // 8

x=12;
x=x*2; // 24
```

It works just as well with `float`s. But I'll mostly stick with `int`s since they're simpler. This increases us by 50%:

```
float f; f=20.0f;
f = f*1.5f; // f is 30
```

3.5.1 examples

These are just traditional exercises to get practice with the idea of changing a variable. They do nothing useful, except to be examples.

The column to the right has the current value (what you'd get if you were crossing out and changing.) For real, you'd use a piece a paper with room to cross out old values.

```
               n
              ---
int n; n=8;    8
n=n+1;         9
n=n+5;        14 <- adds 5 to the previous 9
n=n/2;         7 <- divides the previous 14
n=n*10;       70
```

This is more of the same, but with some subtraction and a fake-out:

```
               a
              ---
```

```
int a; a=8;    8
a=a-1;         7
a=a-2;         5  <- subtracts 2 from the previous 7
a=a-9         -4
a=12;         12  <- just a=12;, not a=a+12;
a=a-1;        11
```

Second from the bottom, `a=12;` is there to trick you. Same as always, that wipes out the work and just makes it 12.

A thing to be careful about is thinking `x=y;` creates a "link". It's easy to think `x=y;` means that `x` is now tracking the value of `y`. But assignment is always a 1-time thing.

Here's an example where I copy `a` and `b` around to show that:

```
           a   b
          --- ---
int a,b;
a=7; b=a;  7   7
a=11;     11   7  <- b didn't follow a
a=a+3;    14   7  <- b still didn't change

a=b;       7   7
b=0;       7   0  <- a didn't follow b

print("a=" + a + "b=" + b); // a=7 b=0
```

That last line is a typical testing print. If I was running this for real, I'd paste it after each change.

This is more copying two variables back and forth, but with extra math:

```
            a   b
           --- ---
int a, b;
a=5; b=20;  5  20
b=a+1;      5   6
a=9;        9   6  <- b stays 6. didn't follow a
a=b*2;     12   6
b=a*3;     12  36
a=a-b/10;   9  36  <- (b/10 is 3)
```

Then one more example, nothing special, but with three variables:

```
  int a, b, c; a=0; b=0; c=0;

           a   b   c
```

```
          --- --- ---
  c=5;     0   0   5
  a=c+9;   14  0   5
  b=a-2;   14  12  5
  c=a/2;   14  12  7
  a=c*3;   21  12  7

  print("a="+a+" b="+b+" c="+c);
```

Later on, we'll be using things like these where they actually do something useful.

This "change me" trick also works with strings, but all we have is +. An example, `w=w+"s";` adds an `s` to the end of `w`. Even more fun, we can also add to the front. Examples:

```
string w; w="snake";
w=w+"s"; // snakes
w=w+"h"; // snakesh
w="Go "+w; // Go snakesh
```

3.6 Declare/assign shortcuts

As mentioned before, you can combine declares of the same type, using commas. `int a; int b; int c;` can be shortened to `int a,b,c;`.

You're also allowed to combine a declare and an assign. `int frogs; frogs=7;` can be combined to make `int frogs=7;`

Examples:

```
int n=5;
n=n+1; // 6

string w1="shark";
string w2="fish";
w1 = w1+w2; // "sharkfish"
```

The starting value is in no way special. `int n=5;` doesn't make the 5 any more sticky or special than regular `n=5;`.

You're also allowed to combine these shortcuts, doing multiple declares and assigns at once:

```
int n=0, maxCows=20;
// long version:
// int n; n=0; int maxCows; maxCows=20;
```

```
string a1="cat", a2, a3="duck";
// a2 is still empty. We don't have to use equals on them all.
```

Some people make a big declare over several lines with comments in the middle. This next thing just declares six variables (and assigns four of them.) I just made up the meaning (but real programs have comments like this):

```
float g1=1.4f, g2=-0.43f,  // cougar coefficients
      h1, h2=1.0f, h3,     // harfle values
      glom=0.0f;           // standard glom factor
```

You're also allowed to chain assign statements. It makes everything equal to the last item:

```
a=b=0; // same as a=0; b=0;

w1=w2=w3=""; // same as w1=""; w2=""; w3="";
```

About the only time you see this is for a "reset," like the examples above.

This next thing is just a trick. I thought this was a good spot for it:

Suppose you want to divide `a/b` to get a decimal answer, but `a` and `b` are both ints. You just need to temporarily turn one into a float, but you can't write `a.0f/b`. What you can do is multiply one of them by `1.0f`:

```
int a=6, b=5;
print( a/b ); // 1, since they're both ints
print( 1.0f*a/b ); // 1.2
```

I like this trick because it's just being clever with what we have – it's not a new rule. `1.0f*a` goes first, turning `a` into a float.

For fun, `a/b*1.0f` won't work – `a/b` goes first, and drops the fraction. But `a/(b*1.0f)` will work. So would the long version, `(a*1.0f)/(b*1.0f)`.

3.7 Semi-useful variable and assign examples

Our programs can't do much so far – we can't even read input yet. So the best we can do for real programs is pretty fake. But these give an idea how a real program would look.

As usual, they go in `Start()`, inside the {}'s.

The first one is just the area/perimeter example, with extra variables and a longer print:

```
  float wide=6, high=8;

  float area = wide*high;
```

```
float perim = wide*2 + high*2;

print("A " + wide + " by " + high + " box has:");
print("area of " + area + " and perimeter of " + perim);
```

It's a common trick to compute the answer into a variable, then print. It's easier to focus on just the math first, and focus on printing the variable second. It also allows us to pick helpful variable names.

For this next one, I wanted something that computes things in steps, and the best I could do was the damage-per-second calculations in games. In this one, a fireball shoots every 2 seconds, doing 40-60 damage. The average damage is 50, which is 25/second:

```
int dMin=40, dMax=60; // fireball min/max damage
float castTime=2.0f;

float dAvg = (dMin + dMax)/2.0f; // 50 average each fireball

float dps = dAvg/castTime; // damage-per-second
```

To print it, I'm going to go a little crazy and "compute" the output sentences into strings first. That's overly complicated for just printing it, but I wanted to show computing a string:

```
string line1, line2;
line2 = "Average damage is " + dps + "/second";

// line1 is the raw stats for a fireball:
string wRange, wTime;
wRange= dMin + " to " + dMax + " damage";
wTime= "every " + castTime + " seconds.";
line1 = wRange + " " + wTime;

print(line1);
print(line2);
```

Notice how I compute `line2` first, since it's easier. That's another small advantage of putting things into variables.

This should print:

```
40 to 60 damage every 2 seconds
Average damage is 25/second
```

This next example uses the "work on a variable" idea to compute 10% compound interest for three months. My plan is to just walk `months` and `cash` through each month. This computes for one month, then uses cut&paste for the next two:

```
  float cash=100; // our starting cash. Anything works
  int months=0; // how many months have gone by, so far

  print("starting with $" + cash);

  cash=cash*1.1f;
  months = months + 1;
  print("Month " + months + ": $" + cash); // Month 1: $110

  cash=cash*1.1f;  months = months + 1;
  print("Month " + months + ": $" + cash); // Month 2: $121

  cash=cash*1.1f;  months = months + 1;
  print("Month " + months + ": $" + cash); // Month 3: $133.1
```

A drawback of growing a variable is you can't look at old values. For example, we can't print them all in one big chart at the end. Just to show we can, each month could have it's own variable:

```
  float cash0 = 100; // starting money, 0 months interest
  float cash1, cash2, cash3; // cash after that many months

  // each month is 10% more than the last:
  cash1 = cash0 * 1.1f;
  cash2 = cash1 * 1.1f;
  cash3 = cash2 * 1.1f;

  print("One month: " + cash1 + ", second: " + cash2 + ", third: " + cash3);
```

3.8 More error messages

The same as last chapter, let's preemptively make some errors and see what messages they give us.

A funny thing about many of these errors is they seem like things the computer could fix for us, without giving an error. But getting more errors is better. All these nit-picky errors are the computer's way of making sure your finished program is crystal clear, with no misunderstandings.

Here's the message if you forget to declare a variable:

```
void Start() {
  gpm=6;
}
// The name 'gpm' does not exist in the current context.
```

I think that's a pretty good message. It doesn't say "forgot to declare" because you might have just spelled it wrong:

```
void Start() {
  int gpm;
  gmp=6; // ooppps! flipped the m and p
}
// The name 'gmp' does not exist in the current context.
```

Using a variable before declaring it gives a straight-forward error message:

```
void Start() {
  gpm=6; // too soon
  int gpm;
}
// A local variable ‘gpm’ cannot be used before it is declared
```

That's one where it didn't need to be an error. But using before declaring looks so funny that we decided to make it illegal.

Here's the error again for using a variable that doesn't have a value yet. Most people would say `gpm` was uninitialized, but the computer uses *unassigned* (which is the same thing):

```
int gpm;
print("value is " + gpm ); // ERROR - gpm has no value
int x; x=gpm; // same ERROR - gpm has no value

// Use of unassigned local variable ‘gpm‘.
```

Here are the errors from assigning the wrong types. A funny thing is that the exact wording changes, but they're all just a way of saying "type mismatch":

```
int x; x=1.2f; // ERROR float into int
// Constant value ‘1.2‘ cannot be converted to a ‘int’

string w; w=6;
// Cannot implicitly convert type ‘int‘ to ‘string‘.

int y; y="12";
// Cannot implicitly convert type ‘string‘ to ‘int‘.

float f = 1.3f;
int x=f;
// Cannot implicitly convert type ‘float‘ to ‘int‘
// An explicit conversion exists (are you missing a cast?)
```

Different wording aside, I think they all give you the idea. I didn't make up that last 2-line error for float into int. It really does have that annoying, condescending second line.

If you forget the `f` on the end of a `float`, the exact error you'll see is another cannot-convert:

```
float f=2.5; // forgot the f on the end
// Literal of type double cannot be implicitly converted to type ‘float‘
```

Now for some different errors. A common mistake is to forget the `x=` in front. That gives a funny-looking error:

```
int x=6;
x+1; // oops! meant to write x=x+1;

// Only assignment, call, increment, decrement, and new object
// expressions can be used as a statement.
```

The words in this one don't mean anything. It's a generic message saying you've completely confused it. It sometimes means you forgot something.

Declaring a variable twice is an error. Just so you know, "local" and "scope" are good technical terms, and we'll see them later:

```
int x;
int x; // ERROR

// A local variable named ‘x’ is already defined in this scope
```

The commas trick is only for the declare-and-assign shortcut. Using it in normal assigns is an error:

```
int a=4, b=8; // this is fine

a=6, b=10; // ERROR. change to semi-colon after a=4

// Unexpected symbol ‘,’, expecting ‘;’
```

I usually get this error when I cut&paste that top line somewhere below. I usually remember to take out the `int`, so I don't double-declare, but then I leave in the stupid commas.

Here's the error for using a built-in word for a variable name. *Unexpected symbol* means it knows what `void` is, but not what's it's doing here. Then it gives you a bonus error for the `=`, since it didn't even know this was suppose to be a declaration:

```
int void=7; // ERROR

// Unexpected symbol ‘void’
// Unexpected symbol ‘=’
```

3.8.1 Run-time errors

All the errors so far happen right away, while the program is scanning the code. There's a different type of error you can only get when the program is running, called a **run-time** error.

For real, we won't get these until later. But we can make one now by dividing by zero. Just `4/0` won't fool it – the computer spots that in advance, as a regular error:

```
print( 4/0 ); // ERROR
// Division by constant zero
```

But if you use variables, the computer can't tell ahead of time. The program will run. But then crashes on that line with an error:

```
int a=4, b=0;
print( a/b ); // not an error, yet
print("this will never print");
```

This will run. But when you press Play it crashes on `a/b`:

```
//DivideByZeroException: Division by zero
//testA.Start () (at Assets/testA.cs:12)
```

It still gives you the problem line and what the problem was, and double-clicking jumps there. *Exception* is just the technical term for an error while it's running.

We won't see many of these yet. But eventually reading syntax errors and fixing them will be no big deal. These run-time errors will be the real problems.

3.9 Doubles vs. floats

Having to put an `f` on the end of 2.5 seems like a bad way to make a computer language. If you want to know why, this section explains it. But you don't need to know. I'm not going to use this part later.

But it might help to get a feel for how the computer thinks, so I'm including it here.

The short version is: most programs use just regular `2.5` for numbers, with no `f` on the end. `float`s are a special-purpose type, made to save space, which no one ever needs. 3D games are one of the few rare places where we use them.

The basic problem is that computers can't store `1/3`rd. It's a repeating value, so they store it as 0.333333 with the rest chopped off. Storing only a certain number of places is called **floating point math** (now that you know the term, you could look up longer, better explanations.)

We decided to make one floating point type that held 6 digits, and another that held 12. When we first made computers, memory was expensive, so six digits was consider normal. We named it `float` for floating point, and named the 12-digit one `double`, for double-sized float. The rules for writing them were flipped back then. Just `3.5` was a regular float, and `3.5d` was a double-sized one.

But then computer memory got cheap, and we decided the 12-digit version should be normal, with the 6-digit one only for extreme space-saving. Today, most normal programs use `double x=1.2;`. Everyone but us gets to write normal numbers with no extra letters after them.

Games are one of the few places where space is vital – we want them to run at high frame-rates on cheap, power-hungry cell-phones. So Unity uses those special space-saving floats for all of its built-ins.

You can declare and use `doubles` in a Unity3D program. But there's rarely a reason.

Assigning doubles to floats is an error since you'd be chopping off the last 6 digits. The computer won't do it, not even if they're only zeroes. To the computer, `float f=1.5;` is as bad as `int n=3.0f;`. So we have to type all of those `f`'s.