Chapter 13

Casting

This section is about two things that aren't super important. It's here as a little break. One is the official way to say "turn `int n` into a `string`." The other is a new type for just 1 letter, which we'll need a lot later, when we look inside of `string`s.

I won't use either of these for a while – you could skip to the next chapter on more functions, if you wanted.

13.1 casting

We know the computer will sometime automatically turn `int`s into `float`s when it has to. For example, `3+2.2f`. Before it can add them, the computer has to turn 3 into `3.0f`. There's an official word for that – `casting`. Officially, the computer **casts** 3 into `3.0f`.

The fun thing is, we don't have to just let the computer cast ints into floats because it needs to. There are also commands to go between them, or between other types, just because we want to.

Letting the computer do it for us is called an **implicit cast**. Running a command to directly tell the computer is an **explicit cast**. The word implicted comes from "imply" – you didn't say to turn 3 into 3.0, but you implied it. Explicit means "spelled out" – I explicitly asked you to convert that 3 to 3.0.

The rule for an **explicit cast** is you put a `type` in parentheses before the value, like `(int)` or `(float)`. It will try to change the next thing into that type. Some examples (these are explained more in their sections):

```
print( (int)3.7f ); // 3 -- turns 3.7 into an int
float f = (float)5/4; // 1.25 -- turns 5 into float 5.0f
```

The parens around the type aren't math parens – they're special required cast parens. As usual, you can add more math parens if you need to: `((float)(5/4))`

has one set of required cast parens, and two sets of math parens.

You can attempt to cast from any type to any other. Some work, some don't, some work in a funny way. The exact rules aren't that important to know – feel free to skim. If you sort of get the general idea, that's fine.

13.1.1 `(int)` from float

Using the `(int)` cast on a float drops the fraction. For example `(int)5.9f` is 5. Some examples:

- `print( (int)2.9f );` prints 2.
- `print( (int)0.3f );` prints 0. This is the same rule, I just wanted to show it even drops the fraction after a zero.
- `float r = (int)2.9f;` makes `r` be 2.0f. Like all math, it goes one step at a time and won't look ahead. So first we turn 2.9 into 2. Then we make it `2.0f` so it can go into `r`.
- `int n = (int)3.7f;` is legal, and makes `n` be 3. The key is, it doesn't just drop the fraction, it turns it into an integer.
- `int n=(int)9.0f;` is legal (but very silly.) It makes `n` be 9. Technically, it drops the point-zero fraction. You can think of it as perfectly turning `9.0f` into an `int`.
- `int n = (int)f;` is legal. Casting works on variables.
  Like other math, it doesn't actually change `f`. If `f` was 7.35, `n` would become 7, but `f` would still be 7.35.

You can use it inside longer expressions. The rule is that a cast happens before math does, but I usually use extra parens just in case. Here are some somewhat silly examples of casting in longer equations:

- `(int)9.7f/2 );` is 4. Casting goes first, so this becomes `9/2`. Those are both `int`s, so we get 4.
- `9/(int)2.9f` is also 4. The `(int)2.9f` goes first, turning it into `9/2`.
- `(int)(4.8f/1.5f)` is 3. The parens make `4.8f/1.5f` go first, giving 3-point-something. Then the `(int)` chops it down to 3.
- `(int)4.8f/1.5f` is a funny 2.66666. The 4.8 gets int-ed, to make `4/1.5f`. But then mixed-type rules turn it into `4.0f/1.5f` and does grown-up math.

- This is kind of a long one, and just for fun. Suppose we have `float a=5.2f, b=2.8f`. Using `(int)a/(int)b` gives 2 (counts as 5/2.) Using `(int)(a/b)` gives 1 (counts as 5.2f/2.8f, which rounds down to 1.) `a/(int)b` gives 2.6 (counts as 5.2/2.)
  This is really just saying we can int-ize just `a`, or just `b`, or both, or the result.

An old trick to round to the nearest is `(int)(f+0.5f))`. It adds 0.5 first, then chops the fraction, so 4.6 become 5.1 then chopped to 5, but 4.3 goes to 4.8, then gets chopped down to 4 anyway. It's very clever.

Using the same idea, `(int)(num+0.99999f)` will round up.

A cute way to round to 2 decimals is to multiply by 100, cast to an `int`, then divide by 100. It's like we slide things over, chop, then slide them back. This example shows it step-by-step, then all in one line:

```
float f = 10.0f / 6.0f; // 1.666666  starting number

// step-by-step:
float f2 = f*100; // 166.6666
f2 = (int)f2; // 166
f2 = f2/100.0f; // 1.66

// all in one line:
float f2 = ((int)(f*100))/100.0f; // 1.66
```

13.1.2 `(float)` from int

A float cast looks like `(float)7`. It adds point-zero to the end. Technically it converts `int` 7 to `float` 7.0f.

This is what we were doing automatically before, so we never really need to use it. But it can still look nice and is good for explaining things:

- `(float)3;` is just a terrible way to write 3.0f.
- We already know the trick how `1.0f*a/b` will trick ints into doing grown-up float division. `(float)n` is another way to do that: `(float)a/b`.

  Technically, `1.0f*a` causes an implicit cast, whereas `(float)a` is an explicit cast. But they both do the same thing. Most people think the `(float)` version looks better.
- For fun, you can write out every implicit cast as an explicit one:

  ```
  float f = 9/1.2f + 4/3;
  // computer turns it into:
  float f = (float)9/1.2f + (float)(4/3);
  ```

There's no reason to do this. It's just a way to explain the automatic "turn ints into floats when you need to" rules.

You can also use `(float)` on `double`s (if you don't remember doubles, just skip to the next section.)

Remember than 3.1 is a `double`, so `float f=3.1;` is a *cannot convert double to float* error. You can convert yourself: `float f = (float)3.1;` (but using `3.1f` is just better.)

If for some bizarre reason you have a `double` variable, and need to use it for a `float`, you can cast it:

```
double x; // why not a float? Pretend there's some reason
  ...
transform.position = new Vector3( (float)x, 0, 0);
```

13.1.3 `string` casts, conversions

There is a way to turn `string`s into numbers, but it's a little funny.

Casts with strings are disabled. In other words `(string)4` and `(int)"37"` should be legal, but aren't, just because. The error messages all sound like "I know you're trying to cast, but the one you're trying isn't allowed."

These are all proper uses of a cast, but all give errors:

```
int n= (int)"4"; // cannot convert string to int
string w= (string)7; // cannot convert int to string
float f= (float)"3.6"; // cannot convert string to float

int n= (int)"abc"; // cannot convert string to int
```

The last one shows the 1-step-at-a-time rule. It doesn't complain `"abc"` isn't even a number, since it never gets that far.

Instead of a casts, there's a function:

```
int n = System.Convert.ToInt32("17"); // n is 17

string w = "-46";
n = System.Convert.ToInt32(w); // n is -46
```

The example shows it works on constants or variables, same as everything else.

The name of the function is really `ToInt32`. It's in the namespace `Convert`, which is inside the `System` namespace.

It takes one `string` input, and uses a rule we haven't seen yet for the `n=` in front (next chapter is about that rule.)

For real, we rarely need to convert a string into an int. The most common is a text area repurposed for numbers. When the user types 35 it's really string `"35"` and we need to convert.

13.2 `char` type

Back in chapter 2, you may have noticed that `string`s are much more complicated than `int`s and `float`s. The basic type that really goes with those two is **character**, which is one "keyboard symbol." Strings are really lists of characters. For example, strings `"cats"` and `"1?->"` are each 4 characters.

Characters are primitive, compared to strings. They're good to see, since they're a common, standard type. We're also going to need them later, when we look inside of `string`s, and they also have some really fun casting rules.

But, again, these are really just details. At some point, you should know what characters are, but the rest of this is just for fun.

13.2.1 `char` examples

The official name is `char` (pronounced `"care"`, like the first syllable of character.) A character literal has single quotes around it. On a standard keyboard it's just under the double-quotes (not the slanted one on the upper-left. That's called a "tick," and isn't used for anything.)

Here's some character use:

```
char c1 = 'y', c2 = '['; // anything on the keyboard is a character
char c3 = ' '; // even a space

if( c1 == 'g' ) {} // can use regular compares
if( c1 != 'u' ) {}
```

Characters have to be exactly one letter. They can never be 2 or more, or empty. This is the rule that keeps them simple. A character is one box, holding one letter. It's as basic as an `int`:

```
char ch = 'ab'; // ERROR -- Too many characters in character literal
ch = ''; // (2 single quotes, no space) ERROR -- Empty character literal
ch=' '; // 1 space. This is fine.
```

As you might guess, the **type** rules won't let you mix strings and chars. These next examples are all things the computer could do, but instead it gives horrible errors about type mismatches:

```
char ch = "a"; // ERROR -- can't assign string to character
string w; w='a'; // ERROR -- can't assign character to string
```

```
if(ch == "x") {} // ERROR -- can't compare string to a character
if(w=='a') {} // ERROR -- can't compare character to a string
if(w==ch) {} // same error
```

The one way it will mix them is the same way it mixes ints, floats and strings. `+` will auto-convert a `char` into a `string`:

```
string w="horse"; w += 'y'; // horsey
char ch='a'; string w = ""+ch; //  "a"
```

Sometimes you want a character which you can't type, like a return, or a double-quote inside of a string. A standard way to make those characters is to use an **escape sequence**. For example, `print( "***\n***" );` will print two lines of three stars.

The slash says to begin an escape sequence, and `n` is the escape code for newLine. Together \n is one character. The internet does a great job of listing the tables, with examples.

13.2.2 Casting `chars`

The most important thing about this section is that you *really* don't need to know it. Deep-down, characters are stored as numbers, and you can do some cute tricks with that. But it's really an unimportant detail that we don't need to know.

I'm putting it here since it shows off more casting rules, and it's fun, and sometimes knowing how things really work can give you a better feel for it.

The entire trick is that the computer can't really store letters – it stores them as numbers using a chart. We can make the computer go between letters and those numbers. That's the entire trick. The rest is how to use it.

The chart is the ASCII chart, which is part of the larger Unicode chart. You can find lots of copies on-line. Some interesting numbers: A-Z are 65-90, a-z are 97-122, and the keys `'0'-'9'` are 48-57. Some examples of how the letters and numbers are interchangeable:

```
int n='a'; print(n); // 97
n='3'; print(n); // 51
```

These work because the computer turns numbers into letters right away. `'a'` was 97 before the program even started. The character `'3'` is 51, because that's where it is on the table.

This even works with math:

```
print( 'a'+2 ); // 99 (97+2)
print( 'd'*2 ); // 200 (a,b,c,d = 97,98,99,100)
print( '2'+12 ); // 62
```

Again, `'a'` was 97 before we even started, and `'d'` was 100. The last one looks tricky – 2+12 is 62? But it's really letter `'2'`, which is 50 in the table.

An explicit cast can go the other way, forcing the computer to look up the number as a letter:

```
print( (char)100 ); // d
print( (char)65 ); // A
print( (char)49 ); // 1 <- this is really '1'
```

A use for this trick is making special characters which can't be typed (or escaped.) 169 is the copyright c-in-a-circle symbol. `w="Cat crammer"+(char)169;` will add that symbol to the end of your invention.

Some fun tricks we can do are checking whether a character is a lower-case letter. This checks whether `ch` is between 97 and 122, which are the a-z numbers:

```
if(ch>='a' && ch<='z') print("lower case");
```

We can turn `'0'` through `'9'` into a real 0-9 by subtracting the code for `'0'`:

```
int num;
if(ch>='0' && ch<='9') // check if a 0-9 letter
  num=ch-'0'; // ex: '3'-'0' is 51-48 is 3
else
  num=-1; // not a digit
```

For a fun bonus, this will print the alphabet. We start at `'A'`, add 1 each update, and repeat after we hit `'Z'`:

```
int x='A'; // starts x at the number code for A (which is 65)

void Update() {
  print( x + " " + (char)x ); // ex: 65 A
  x++;
  if(x>'Z') x='A';
}
```

It really just says to make `x` spin through the number-codes for A-Z, printing as it goes.