Chapter 13

Casting and `char`

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 one letter, which we won' t need until 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 that in `3+2.2f` the computer has to convert the 3 into `3.0f`. Likewise `"b"+4` requires the 4 be converted into a string. The official word for that is **casting**. As a verb: "4 is cast into a string".

The fun thing is, we don't have to just let the computer cast because it needs to. That's called an **implicit** cast. There are commands that explicitly tell the computer to convert one type into another.

The rule for an *explicit* cast is to 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:

- `(int)2.9f` is 2.
- `(int)0.3f` is 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 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 cast `double`'s to `float`'s (if you skipped that section or hated it, skip this one too).

Suppose we need `x` as a double for extra accuracy. Using it for position or color or size is an error, since they take floats. The computer could cut off the extra 6 decimal places, but it wants to be sure that's what you want. You have to cast it to `float`:

```
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 cast, there are functions. This turns a string into an int:

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

The name of the function is `ToInt32`. It's in the namespace `Convert`, which is inside the `System` namespace. We haven't seen the rule yet about a function having an output – the part where `n=` is legal, but we will, next chapter.

We can already implicitly cast ints to strings with `""+n`, and that works fine. But there's also a function for it:

```
string w = System.Convert.ToString(n); // same as w=""+n;
```

Note how it's also in the `Convert` namespace. It also works for float to string.

13.2 `char` type

Back in chapter 3, you may have noticed that strings are much more complicated than ints and floats. 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. It's just for fun.

Computer can't really store letters. That's probably obvious. They store letters as numbers, using a chart. `char ch='a';` puts 97 into `ch`. The computer will gladly print it as 97. Likewise, if you give the computer a 98, it can use the chart to see that's `'b'`.

The chart is the ASCII chart, which is part of the larger Unicode chart (you can find a copy on-line). Some interesting values from it: A-Z are 65-90, a-z are 97-122, and the keys `'0'-'9'` are 48-57. Basically, we took the numbers 0-255 and arranged every letter, upper-case letter, digit, and punctuation symbol. There's no perfect spot for things, so we put them where-ever. Some early versions didn't even have all of the letters in a row.

Examples of char casting:

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

The computer is happy with this. `'a'` actually is 97. The computer is glad to put in into an `int`. The `'3'` is a little sneakier. We put in between single-quotes, which means it's a char, which means we use the chart and it's the number 51.

This even works with math:

```
print( 'a'+2 ); // 99 (97+2)
print( 'd'*2 ); // 200 (d is 100; 3 steps from 'a')
print( '2'+12 ); // 62 (since '2' is 50)
```

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'
```

One use for this trick is making special characters which can't be typed. 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. We're really comparing numbers, but it looks nicer writing them as letters:

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

if(ch>='A' && ch<='Z') print("upper case");
if(ch>='0' && ch<='9') print("ch is a digit");
```

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

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

This update loop 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';
}
```

This abuses implicit casts from char to int three different places.