Tip 22: all the casting you'll need

Ben Klemens

14 November 2011

level: intermediate
purpose: still less obsolete cruft in your life

There are two (2) reasons to cast a variable from one type to another.

First: when dividing two numbers, an integer divided by an integer will always return an integer, so the following statements will be true:

```
4/2 == 2
3/2 == 1
```

That second one is the source of lots of errors. It's easy to fix: if `i` is an integer, then i + 0.0 is a floating-point number that matches the integer. Don't forget the parentheses, but that solves your problem:

```
4/(2+0.0) == 2.0
3/(2+0.0) == 1.5
```

You can also use the casting form:

```
4/(float)2 == 2.0
3/(float)2 == 1.5
```

I'm partial to the add-zero form, for æsthetic reasons; you're welcome to prefer the cast-to-float form. But make a habit of one or the other every time you reach for that `/` key, because this is the source of many, many errors. [And not just in C; lots of other languages also like to insist that `int / int` $\to$ `int`. Not that that makes it OK.]

Second: array indices have to be integers. It's the law (C standard §6.5.2.1), and GCC will complain if you send a floating-point index. So, you may have to cast to an integer, even if you know that in your situation you will always have an integer-valued expression.

```
4/(float)2 == 2.0 //this is float, not an int.
mylist[4/(float)2]; //So this is an error: floating-point index

mylist[(int)(4/(float)2)]; //This works; take care with the parens

int index=4/(float)2;//This form also works,
mylist[index];       //and is more legible.
```

Now that I've covered both of the reasons to cast in C, I can point out the reasons to not bother. Notice that the `index` variable above was an integer, but the right-hand value was a floating-point number. C auto-casts in this case, truncating down to the right value. If it's valid to assign an item of one type to an item of another type, then C will do it for you without your having to tell it to with an explicit cast; if it's not valid, then you'll have to write a function to do the conversion anyway.

C++ isn't like this: you have to explicitly cast in all cases. Fortunately, you're writing in C, so you can ignore C++ tutorials that tell you to explicitly cast. [And as a broad rule that universally works for me: don't bother with anything that uses *C/C++* in the title.]

In the 1970s and 80s, `malloc` returned a `char*` pointer, and had to be cast (unless you were allocating a string), with a form like:

```
//don't bother with this sort of redundancy:
float* list = (float) malloc(list_length * sizeof(float));
```

You don't have to do this anymore, because `malloc` now gives you a `void*` pointer, which the compiler will comfortably auto-cast to anything.

If you check the examples above, you'll see that I even gave you options to avoid the casting syntax for the two legitimate reasons to cast: adding 0.0 and declaring an integer variable for your array indices. Bear in mind the existence of the casting form `var_type2 = (type2) var_type1`, because it might come in handy some day, and in a few tips we'll get to declarations that mimic this form. But for the most part, explicit type casting is just redundancy that clutters the page.