Converting Strings to ints

One important thing to remember when using strings is that they cannot be implicitly converted to integers (or floating point types) by casting—either implicit or explicit. Consider the following code fragment:

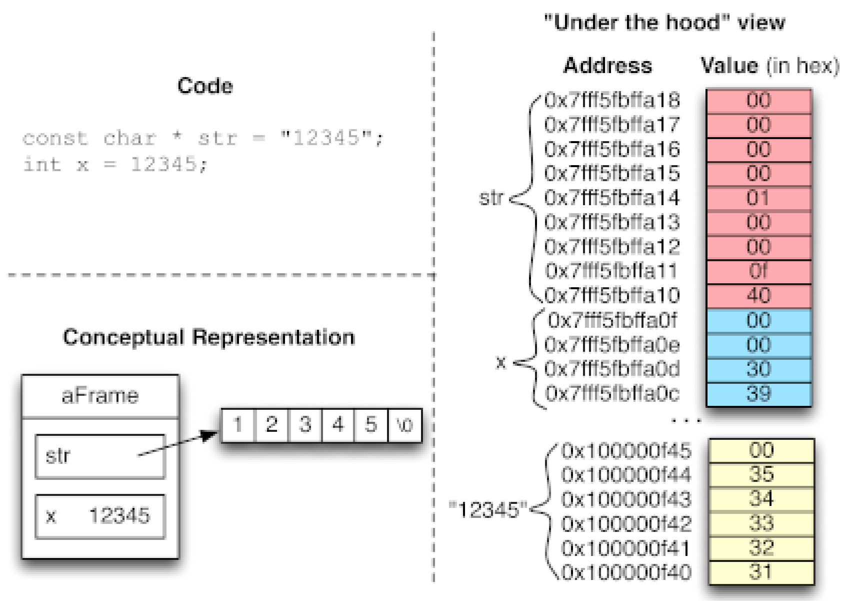
```
int x = str;

const char * str = "12345";
```

Attempting to compile this piece of code results in the error message:

```
initialization makes integer from pointer without a cast
```

This error arises because the assignment does *not* convert the number that the string represents textually into an integer (that is, it does not result in *x=12345*). Instead, it would take the numerical value of *str* (which is a pointer, thus its numerical value is the address in memory where the sequence of characters 12345 is stored) and assign it to x. This behavior follows exactly the rules we have learned for assignment statements: evaluate the right side to a value (which is an arrow, meaning its numerical value is an address), and write it in the box named by the left side.



```
const char * str = "12345";
int x = 12345;
```

To help better understand why simple assignment does not work, the figure above gives a peek "under the hood" for code which has a **char** * pointing at the string "*12345*", and an *int* with the value 12345. The left side of the figure shows the conceptual representation, which we typically work with. On the right side, the figure shows the same state of the program, but with addresses and the numeric values contained in those addresses. The variable *str* (whose bytes are colored in red) is a pointer, which is 8-bytes on this particular system. Its numeric value is the address in memory of the bytes of the string literal "*12345*", which is 0x100000f40. The contents of memory locations 0x100000f40–0x100000f45 are the characters of that string—the numeric values for the characters '1' (0x31), '2' (0x32), '3' (0x33), '4' (0x34), '5' (0x35), and '*\0*' (0x00) in that order. The variable *x* which is an *int* occupies 4 bytes, which hold the value 0x00003039, which is 12345 in decimal.

If we were to convince the compiler to allow us to assign *x* = *str*, we would copy the **value** from *str* (which is 0x100000f40) into *x*. Of course, since *x* cannot hold this entire value (remember that on this particular system, pointers are 8 bytes, but ints are 4 bytes), the value will be truncated. *x* would end up being assigned the value 0x0000f40 (the lowest four bytes of 0x100000f40), which is 3904 in decimal—still not what we want.

Another incorrect (at least for this task) approach would be to write *x* = **str*, dereferencing str to get the value at the end of the arrow, rather than the pointer itself. Here, we would read one character out of the string (**str* evaluates to '1', which is 0x31). We would then assign this value (0x31) to *x*. Now, we would end up with *x* being 0x31 (which is 49 in decimal)—also not what we desire!

While the previous example may seem a bit contrived due to the use of a literal string (why not just write *int x = 12345*;), consider a more useful example:

```
          // we'd like to store it as an int...
```

This example not only illustrates *why* we might want to perform this sort of operation, but also leads us into understanding one of the complexities in such a task. What if the user enters "xyz"? How do we then convert that to a number? For that matter, what if our code instead read (note the “in hexadecimal”):

```
int x = str; // they will enter a string.
             // we'd like to store it as an int...
printf("Please enter a number in hexadecimal:");
char * str = readAStringFromTheUser(); //we'll learn how later
```

Now, if the user enters the sequence of characters *12345*, they do not mean the number 12,345 (twelve thousand, three hundred, and forty five), since we told them we would interpret it as hexadecimal. Instead it is 74,565 in decimal (you can work out this conversion yourself for practice).

If we wanted to perform such a conversion ourselves by hand, we would need to iterate over the characters in the string and perform math. However, as this type of conversion is a common task, there are C library functions which perform it for us. The *atoi* function is the simplest of these—it converts a string to an integer by interpreting the sequence of characters as a decimal number. If there is no valid number at the start, it returns 0. A slightly more complex function is *strtol*, which lets you specify the base (decimal, hexadecimal, etc…), as well as to pass in the address of a **char** * which it will fill in with a pointer to the first character after the number. That is, if you give it the string *123xyz*, it will set this pointer to point at the '*x*' (you can also pass in NULL, if you do not need this extra information, in which case it skips this part). This extra argument allows you to do error checking, and/or figure out what comes after the number (if it part of a larger input you need to process, such as a comma-separate list of numbers).

This concept can be a little bit confusing at first, as you are used to seeing the textual representation of a number (*e.g.*, its decimal form written on paper and thinking of it as the number itself). The best way to understand this concept is to write these functions which convert from strings to **int**s yourself (in general, it is best to learn by doing). If you want some extra practice in this area, we recommend you try the following optional exercises:

1. Write the function **int myatoi(const char * str)** which behaves like the *atoi* function, except that you write it yourself (without using *atoi* or *strtol*)
2. Write the function **int myatoiHex(const char * str)** which behaves like the *atoi* function, except (a) it interprets the string as a hexadecimal (base 16) number rather than decimal and (b) you write it yourself (without using *atoi* or *strtol*)
3. Write the function **long mystrtol(const char * str, char ** endptr, int base)** which behaves like *strtol*, except that you write it yourself (without using *atoi* or *strtol*)

Here are a few hints to help you with these problems:

- Remember that everything is a number, including characters. This means that you could do things like take a character **c** and compare it to other characters (e.g., **c >= '0'**) , subtract one character from another (**c - '0'**) or any other math you might want to do. Note that the digits 0--9 all have consecutive numeric values so '7' - '0' is 7.
- Remember the in decimal numbers, each place is 10x as large as the previous: 12 is (10 *1) + 2. 123 is 100 * 1 + 10 * 2 + 3. It may help to think of this as (1 * 10+ 2) *10 +3. Likewise, 1234 is ((1 *10 +2) *10 +3) * 10 + 4.
- For base 16, similar principles apply, but each place is 16x as large as the previous: 0x1fc is (1 * 16 + 15) * 16 + 12.

✓

Completed