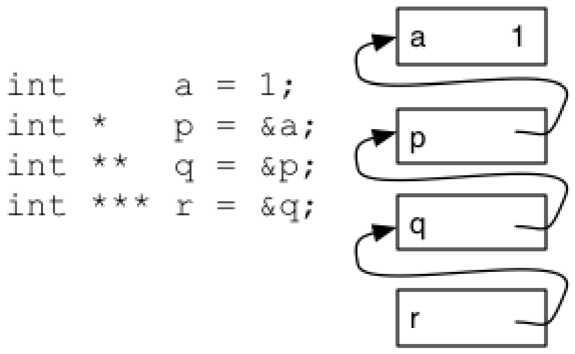
# Aliasing | Coursera

coursera.org/learn/pointers-arrays-recursion/supplement/Nr8XY/aliasing

## Aliasing

## Aliasing

In our discussion of pointers, we have alluded to the fact that we may now have multiple names for the same box; however, we have not explicitly discussed this fact. For example, in the figure below, we have four names for a's box: *a, *p, **q*, and ****r*. Whenever we have more than one name for a box, we say that the names *alias* each other—that is, we might say **p* aliases ***q*.



```
int       a = 1;
int *     p = &a;
int **    q = &p;
int ***   r = &q;
```

Aliasing plays a couple of important roles in our programming. First, when we are working through Step 2 of our programming process, we may find that we changed a value, but there are multiple ways we can name what value we changed. When we write down precisely what we did, it is crucial to think about which name we used to find the box when we worked the problem by hand in Step 1. If we are not sure, we should make note of the other names we might have used—then if we have trouble generalizing, we can consider whether we should have be using some other name instead.

When we are debugging (or executing code by hand), it is also important to understand aliasing. Novice C programmers often express surprise and confusion at the fact that a variable changes its value without being directly assigned to: "I wrote *x = 4*;, then look I don't assign to *x* anywhere in this code, but now it is 47!" Generally such behavior indicates that you alias the variable in question (although you may not have meant to). If you have problems of this type, using *watch* commands in *gdb* can be incredibly helpful.

If we want to live dangerously, we can even have aliases with different types. Consider the following code:

```
3

1

2

printf("%d\n", *p);

float f = 3.14;
```

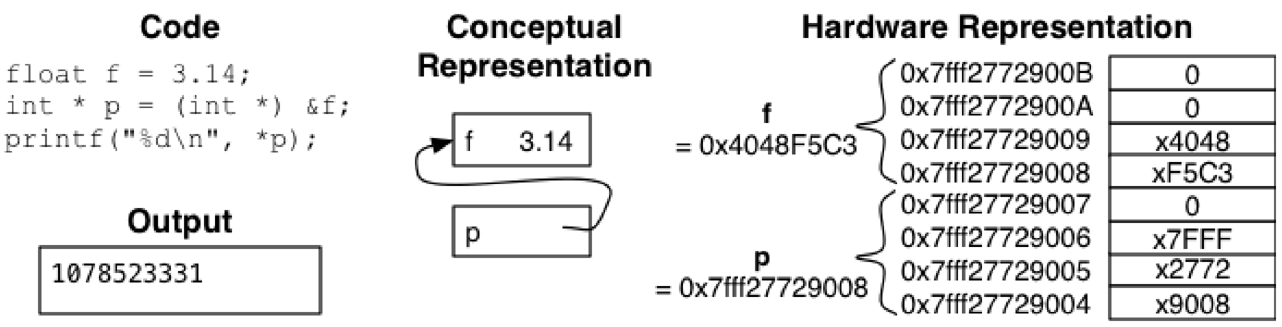
```
int * p = (int *) &f;  // generally a bad idea!
```

Here, *f* is an alias for **p*, although *f* is declared to be a float, while *p* is declared to be a pointer to an int (so we have told the compiler that **p* is an int). What will happen when you run this code? Your first reaction might be to say that it prints 3, the following (similar-ish) code would print 3:

```
float f = 3.14;

int x = (int) f;

printf("%d\n", x);
```



Code

```
float f = 3.14;
int * p = (int *) &f;
printf("%d\n", *p);
```

Output

```
1078523331
```

An example of aliasing with different types. Storing the floating point bit encoding for *3.14* in *f*, then reading it out as though it represented an integer.

However, if we run the first code snippet on the computer, we will get 1078523331! This result may seem like the computer is just giving us random nonsense, however, what happened is perfectly logical (if a bit low level). When we cast a float to an int (in the second snippet of code), we ask the compiler to insert instructions which convert from a float to an int. However, when we dereference a pointer to an int, the compiler just generates instructions to read the bit pattern at that location in memory, as something that is already an int. In the first snippet of code, initializing **float** *f = 3.14*; writes the bit pattern of the floating point encoding of *3.14* into *f*'s box. Without going into the details, the floating point encoding of *3.14* works out to *0100 0000 0100 1000 1111 0101 1100*

*0011 (= 0x4048F5C3* in hex, *1078523331* in decimal). When we dereference the pointer as an int, the program reads out that bit pattern, interprets it as though it were an integer, and prints *1078523331* as output, as the figure above illustrates.

This last example is not something you need to use in programs you write, but rather a caution against abusing pointers. Unless/until you understand exactly what is happening here and have a good reason to do it, you should not cast between pointer types.