Box-and-Pointer Diagrams – How to make and use them in C

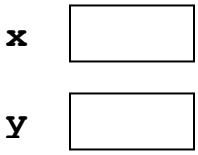
Discussion: ***Box-and-pointer diagrams*** help you trace by hand and understand code, especially code that features ***pointers*** (and/or arrays, which can be thought of as pointers).

The concepts that follow apply to both C and Python (and also to other languages), but some of the details are particular to C.

a. Variables are stored at locations (aka *addresses*) in memory. For example, the declarations

```
double x;
double y;
```

allocate storage for two double-precision floating-point numbers. We notate this using the ***box*** part of a ***box-and-pointer diagram***:



The boxes are shown as empty to indicate that the values of the variables are *garbage* at this point; the variables have been declared but not initialized. Their values are whatever bits happen to be in memory at those locations.

b. There are two ways to put values into variables. The simplest is ***ordinary assignment***, e.g.:

```
x = 15;
y = x;
```

This puts the *value* on the *right-hand-side* of the assignment into the *location* specified by the *left-hand-side* of the assignment. We notate this by:

Note that the second assignment (`y = x`) *copies* the bits from the location specified by ***x*** to the location specified by ***y***. There is no association between ***x*** and ***y***; they simply happen to contain the same bits in this example. For example, the subsequent statement:

```
y = 23;
```

causes the diagram to change to:

c. A second way to put values into variables is to ***make a function call***. Consider:

```
void foo(double a, double x) {
    ...
}
```

and consider the function call:

```
foo(x, y);
```

where ***x*** and ***y*** each have values 15 and 23, respectively (as shown above). This function call causes the following to happen:

Step 1: Space is allocated in memory for variables named ***a*** and ***x*** (the ***parameters*** of function ***foo***).

Memory allocated by ***main*** (or wherever the function call occurs):		Memory allocated when the ***foo*** function is *called*:	
x	15	a	
y	23	x	

Note that the space allocated for the ***x*** associated with ***foo*** has nothing to do with the space allocated for the ***x*** in ***main*** – variables are *local* to their function.

Step 2: The memory locations allocated for ***a*** and ***x*** are assigned values by copying the bits from the ***actual arguments*** (***x*** and ***y***) into the corresponding parameters (***a*** and ***x***).

Memory allocated by ***main*** (or wherever the function call occurs):		Memory allocated when the ***foo*** function is called:	
x	15	a	15
y	23	x	23

You should now understand exactly what happens when a function is called. If not, please ask now.

d. The ***compiler***, along with its run-time associate the ***linker***, determines where in memory your variables are stored. All you can count on is:

   i. Once space is allocated for a variable, the variable refers to the same space for its entire lifetime.

   ii. Each element in an array is in the next location in memory from the previous element of the array.

e. In C, *unlike* most languages, your program can find out where a variable is stored in memory, by using the **`&`** ("***address of***") operator. For example, consider the code:

```
double x;
   ... &x ...
```

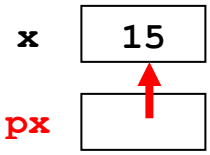
The value of the expression **`&x`** is the ***location*** (aka ***address***) in memory where ***x*** is stored. For example, if ***x*** happens to be stored at location (say) ***20480012***, then the value of **`&x`** is ***20480012***.

f. Pointer variables are variables whose value is a location (aka address) in memory. For example:

```
double x = 15;
double* px;
px = &x;
```

Note how pointer variables are declared: you just append an asterisk to the type.

In this example, the pointer variable ***px*** has as its value the location in memory where ***x*** is stored. We notate this with the ***pointer*** part of a ***box-and-pointer diagram***:

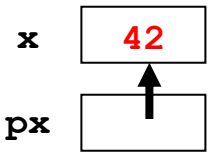


We don't put a value inside the ***px*** box, because we don't know that value (the compiler and linker chose it). However, we do know that the value in the ***px*** box is the location in memory of the ***x*** box, so we draw an arrow in the ***px*** box that points to the ***x*** box.

g. C uses the notation **`*px`** to refer to the ***pointee*** of ***px*** – that is, the box that pointer variable ***px*** points to. Continuing the above example, the statement:

```
*px = 42;
```

causes the box-and-pointer diagram to change to:



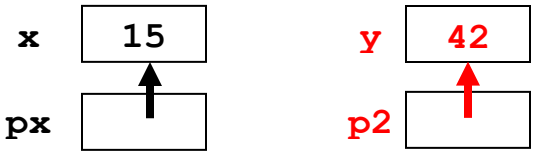
Thus, the statement **`*px = 42`** has the effect of changing the value of variable ***x*** to ***42***.

h. Assignment with pointers works just like assignment with ordinary variables, except that since we notate the contents of pointer boxes by using arrows, assignment of pointers causes the arrows to change.

For example, continuing the above example, first consider the statements:

```
double y = 42;
double* p2;
p2 = &y;
```

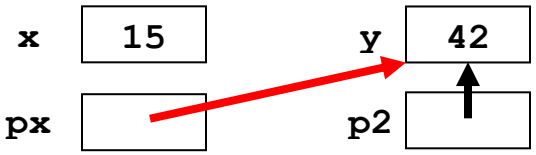
This causes our box-and-pointer diagram to extend to:



Now consider the effect of the assignment:

```
px = p2;
```

Just as with non-pointer variables, this copies the bits in **p2**'s box into **px**'s box. However, we note the bits in pointer boxes by arrows. Hence, copying the bits in **p2**'s box into **px**'s box has the effect of making the arrow from **px**'s box point to the same place that the arrow from **p2**'s box points to:



At this point, both **px** and **p2** are pointers to **y**'s box. Hence, all three of the following statements have the same effect at this point:

```
y = 23;
*px = 23;
*p2 = 23;
```

They all change the contents of the **y**'s box to **23**.

i. So far, we have ignored the question of ***why*** pointers are valuable. Briefly:

- Having parameters that are pointers allows us to send back information from the function to the code that called the function. See How to Use Pointers to Send Information Back From a Function for an example of this.
- The value of a pointer variable is a *location*. Sometimes that location is the beginning of many chunks of related data, as in an array or a structure instance. If we want to send that data to a function, it saves both time and space to pass a pointer to the beginning of that data instead of passing all of the data itself. See How to Use Pointers to Save Time and Space for more about this.

If you have questions about the above discussion, please ask now!