Pointer Basics

/run/media/lijunjie/资料/Learning Video/pointers-arrays-recursion/01_pointers/02_pointers-conceptually/01_pointer-basics_instructions.html

Pointers are way of referring to the *location* of a variable. Instead of storing a value like 5 or the character 'c', a pointer's value is the location of another variable. Later in this chapter we will discuss how this location is stored in hardware (*i.e* ., as a number). Conceptually, you can think of the location as an arrow that points to another variable.

Just like we can make variables that are integers, doubles, *etc.* , we can make variables that are pointers. Such variables have a size (a number of bytes in memory), a name, and a value. They must be declared and initialized.

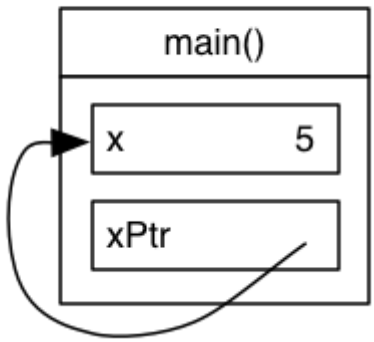
Declaring a pointer

In C, "pointer" (by itself) is not a type. It is a *type constructor* —a language construct which, when applied to another type, gives us a new type. In particular, adding a star (*) after any type names the type, which is a pointer *to that type*. For example, the code **char * my_char_pointer;** (pronounced "char star my char pointer") declares a variable with the name **my_char_pointer** with the type *pointer to a char* (a variable that points to a character). The declaration of the pointer tells you the name of the variable and type of variable that this pointer will be pointing to.

Code

```
1  main () {
2
3      int x = 5;
4      int *xPtr;
5
6      xPtr = &x;
7      *xPtr = 6;
8      printf("x = %d\n", x);
9  }
```

Conceptual Representation



Assigning to a pointer

As with all other variables, we can assign to pointers, changing their values. In the case of a pointer, changing its value means changing where it points—what its arrow points at. Just like other variables, we will want to initialize a pointer (giving it something to point at) before we use it for anything else. If we do not initialize a pointer before we use it, we have an arrow pointing at some random location in our program, and we will get bad behavior from our program—if we are lucky, it will crash.

Assignment statements involving pointers follow the same rules that we have seen so far. However, we have not yet seen any way to get an arrow pointing at a box—which is what we need to assign to a pointer. To get an arrow pointing at a box (technically speaking, the *address* of that box in memory), we need a way to name that box, and then we need to use the **&** operator. Conceptually, the & operator (the symbol is called an "ampersand", and the operator is named the "address-of" operator) gives us an arrow pointing at its operand. The code **xPtr = &x;** , for example, sets the value of the variable **xPtr** to the *address of* **x** . After it is initialized, **xPtr** points to the variable **x** . On the left of the figure above you can see code that declares an integer **x** and a pointer to an integer **xPtr** . In line 3, the value of **x** is initialized to 5 on the same line in which it is declared. In line 6, the value of **xPtr** is initialized to the location of **x** . Once initialized, **xPtr** *points* to **x** .

The address-of operator can be applied to any lvalue (recall that an lvalue is an expression that "names a box") and evaluates to an arrow pointing at the box named by the lvalue. The only kind of lvalue we have seen so far is a variable, which names its own box. Accordingly, for a variable **x** , the expression **&x** gives an arrow pointing at **x** 's box. It is important to note that the address of a variable is not itself an lvalue, and thus not something that can be changed by the programmer. The code **&x = 5;** will not compile. A programmer can access the location of a variable, but it is not possible to change the location of a variable.

Note that in the context of pointers, the & symbol is a unary operator—an operator that takes only *one* operand. It is used before the lvalue whose address should taken. This operator is not to be confused with the *binary* operator—an operator that takes *two* operands. The binary operator & performs a bitwise AND (performing a boolean AND on each binary bit of the two operands).

Dereferencing a pointer

Once we have arrows pointing at boxes, we want to make use of them by "following the arrow" and operating on the box it points at—either reading or changing the value in the box at the end of the arrow. Following the arrow is accomplished using the star symbol * , a unary operator that *dereferences* a pointer. An example of the use of the derference operator is shown in line 7 of the figure above. ***xPtr = 6;** changes the value that xPtr points to (*i.e.* , the value in the box at the end of the arrow—namely, x's box)—to 6. Notice that the green arrow indicates that this line has not yet been executed, hence *x* still has the value 5 in the conceptual representation. Once line 7 is executed, however, the value will be 6.

Do not be confused by the two contexts in which you will see the star (*) symbol. In a variable declaration, such as *int *p* ;), the star is part of the type name, and tells us that we want a pointer to some other type (in our example, **int** * is the type of p). In an expression, the star is the dereference operator. For example, the code *r = *p* ; gives the variable *r* a new value, namely the value inside the box that *p* (which is an arrow) points to. The code **p = r* ; changes the value inside the box that p points at to be a new value, namely the value of the variable *r* . As always, remember that you can think about

declaration with initialization as two statements: **int** * *q = &y* ; is the same as the two statements int *q; q = &y;. Generally when you work with pointers, you will use the star first to declare the variable and then later to dereference it. Only variables that are of a pointer type may be dereferenced.