From pointers to objects: how to avoid confusion

0 Introduction

The other day I was lecturing about the implementation of result parameters (of procedures) by means of value parameters and pointers. It so happened that some of the students got confused, as they began to wonder whether or not, when a pointer is copied, the value referred to by that pointer should be copied as well. This betrays their lack of a proper understanding of the pointer mechanism but they are to be blamed only partly: the way in which pointers are introduced (in Pascal or, if you like, C) really *is* confusing. Moreover, recently I have encountered what seems to be the same confusion in discussions about object-oriented programming. Therefore, it seems appropriate to devote a note to this issue, particularly so because from the right point of view there is no reason for confusion at all.

This note is about programming concepts, not about their implementation. This distinction is crucial, because the (bad) habit of trying to understand a programming concept solely in terms of its implementation is responsible for at least some of the confusion. Here I shall explain the pointer mechanism by drawing a comparison with arrays: in a way the two concepts are the same and arrays are well-understood.

1 Variables and their values

A (program) variable is a *name* and, during execution of the program, it has a *value*. Distinct variables always have different names, but distinct variables may have the same value. Occurrences of a variable in an expression always refer to the variable's value. For example, the (boolean) expression $x = y$ has the value true if and only if x and y have the same value. Leibniz's principle of *substitution of equals for equals* applies; for example: $[\, x = y \;\Rightarrow\; x^2{+}3 = y^2{+}3 \,]$. This also explains why the C-construct $\&x$ is so horrid: $\&x$ is not an ordinary expression depending on the value of x; for instance, we do not have: $[\, x = y \;\Rightarrow\; \&x = \&y \,]$.

An assignment like $x := y$ does not make x the same variable as y, but gives x the same value as y has. Notice that in this assignment x does *not* refer to the value of x: here we do *not* have that when $x = 5$ the assignment is equivalent to $5 := y$ (which would be meaningless).

aside: In the weakest-precondition semantics, the distinction between variables and their values is void: a variable is just a place-holder in expressions, for which other expressions can be substituted. This is reflected by the *axiom of assignment*, which defines (the semantics of) the assignment statement as a

substitution:

$$[\ wp(x{:=}E, Q) \ \equiv \ Q(x{:=}E)\ ] \quad .$$

In other words, the values of variables only play a role during program execution.

□

It must be clear that the only purpose of a variable's name is to distinguish it from the other variables in the program. The name of a variable has no intrinsic meaning beyond this purpose. Programmers can employ this freedom by choosing "meaningful identifiers" for their variables, but this is not without danger: one is easily seduced to attribute to a variable properties that are only *suggested* by its name. It is, for instance, perfectly possible to write down assignments like:

$$sum := x - y \quad .$$

Here is a more subtle example of the kind of confusion that may arise in this way. Assume that variables of a datatype (or: object class) *person* are used to represent information about people. To represent a person named "John" we might use a variable *John* of type *person*, but how are we now going to interpret an assignment like:

$$John := Mary \quad ?$$

Of course, nothing is wrong with this assignment: it does not make John the same person as Mary, it just establishes that (apparently) John has the same properties Mary has. The reader who feels an emotional resistance against this assignment probably has identified, without being aware, the variable with the person it is supposed to represent.

The above is very elementary, but it occurs to me that in particular (some of the) object-oriented programmers tend to forget the importance of not confusing program variables with the "real-life" objects they represent. To me such confusion is surprising, because the program's *specification* should provide the logical firewall between the formal program and reality. I hope that this is not a symptom that object oriented programmers tend to skip the construction of formal specifications.

2 Arrays as stored functions

We consider a variable x of type:

$$\mathsf{array}\ [\,0\,..\,N{-}1\,]\ \mathsf{of}\ \mathsf{Int} \quad .$$

One way to look at this is to consider x as a collection of N distinct integer variables that only happen to be named in a systematic way. The prevailing (and preferred) interpretation, however, is to consider x as a single variable whose value is a function of type $[0\,..\,N) \to \mathsf{Int}$. This does justice to the observation that a function (on a finite domain) can be represented in two ways, namely by an algorithm for computing its values and by a table containing all its values. Procedures provide the means for implementing *computed functions*, whereas arrays provide the means for implementing *stored functions*, in the case that their domains are finite intervals of the integers.

In Pascal-like languages the difference between computed and stored functions is visible in the notation used for function application; for computed function x its application to an argument p is usually written as $x(p)$, whereas for array x its application to argument p is written as $x[p]$.

aside: One might well argue that the notation should be the same for computed and stored functions alike, because the underlying mathematical concept — function— is the same in both cases.
□

In this view an assignment to a single element of x modifies the whole function stored in x, albeit in a single point of its domain only. An assignment $x[p] := E$ is just an abbreviation of $x := x(p \to E)$, where, for every function f, the expression $f(p \to E)$ denotes the function that equals f in all points of its domain, except in point p where its value is E. The advantage of this is that the axiom of assignment remains valid.

Nobody who understands arrays becomes confused by an example like:

$$\{\ x[p] = 3 \wedge p = q\ \}\ \ x[q] := 5\ \ \{\ x[p] = 5\ \}\quad .$$

Because x is a function Leibniz's principle applies and whatever the value of x is we have: $[\ p = q\ \Rightarrow\ x[p] = x[q]\ ]$. Hence, assignments to $x[q]$ change $x[p]$ as well whenever $p = q$.

Now suppose that we have a procedure P with a value parameter h, defined by:

$$\mathsf{procedure}\ P(h : \mathsf{Int})\ =\ |[\ \ x[h] := 5\ \ ]|\quad .$$

Then we also have:

$$\{\ x[p] = 3 \wedge p = q\ \}\ \ P(q)\ \ \{\ x[p] = 5\ \}\quad .$$

In a call $P(q)$ the parameter transfer amounts to an assignment $h := q$; this does not involve x at all. The procedurebody contains x as a global variable, and in some way q may be considered to "refer to" $x[q]$, but so what? The only value

involved in the parameter transfer is q, independently of what it means or what it "refers to".

Of course, the same can be said about ordinary assignments like $p := q$: although, one way or the other, p and q may refer to (elements of) x, the assignment $p := q$ has nothing to do with x, even though the function value $x[p]$ before the assignment may differ from the function value $x[p]$ after the assignment; this difference is but a result of the change in p. There is no difference with the observation that $sin(p)$ after an assignment to p probably will have a different value than before.

3 Towards pointers

An array is a stored function whose domain is a finite interval of the integers. Ignoring that it is an integer interval, we now consider a finite set V as the domain for a stored function x. That is, variable x now has type:

$$V \rightarrow \mathsf{Int} \quad .$$

We make no assumptions about the internal structure of V, but we do allow variables of type V and, for p of type V, we denote the value of function x in point p by $x[p]$, as before.

The discussion in the previous section is completely independent of the fact that an array's domain is an interval of the integers. So, this discussion remains valid and, for instance, we still have:

$$\{\ x[p] = 3 \wedge p = q\ \}\ \ x[q] := 5\ \ \{\ x[p] = 5\ \} \quad .$$

As before, because x is a function Leibniz's principle applies and whatever the value of x is we have: $[\ p = q\ \Rightarrow\ x[p] = x[q]\ ]$. Hence, assignments to $x[q]$ change $x[p]$ as well whenever $p = q$.

Now suppose again that we have a procedure P, now defined by:

$$\mathsf{procedure}\ P(h : V)\ =\ |[\ \ x[h] := 5\ \ ]| \quad .$$

Then we still have:

$$\{\ x[p] = 3 \wedge p = q\ \}\ \ P(q)\ \ \{\ x[p] = 5\ \} \quad .$$

This shows that the concept of a stored function with an arbitrary finite domain is equally viable as the concept of an array. Now we add something new, by assuming that V itself is a variable, of type:

$$\mathsf{subset\ of}\ \ \Omega \quad ,$$

where Ω is an *infinite* set the internal structure of which is left unspecified. The value of variable V is a finite subset of Ω. It is finite because we assume the initial value of V to be empty (which certainly is finite) and because we introduce only one operation to modify V; this operation extends V with a new element (this maintains finiteness). We denote this operation by:

$$new(V,p) \quad ,$$

where p is an output parameter of the operation. Its effect can be specified operationally by the program fragment:

$$p \leftarrow \Omega \backslash V \;\; ; \;\; V := V \cup \{p\} \quad ,$$

where $p \leftarrow \Omega \backslash V$ must be read as "select a value in $\Omega \backslash V$ and assign it to p". (Notice that, because Ω is infinite whereas V is finite, such selection is always possible.) So, $new(V,p)$ extends V with a new value and assigns this value to p.

We wish to maintain that x is a function with domain V; therefore, every extension of V with a new element p requires a corresponding extension of x with a function value $x[p]$. Hence, we may expect every extension of V to be followed by an initialisation of $x[p]$, in the following way —for some expression E—:

$$new(V,p) \;\; ; \;\; x[p] := E \quad .$$

remarks: Alternatively, we may incorporate the initialisation of $x[p]$ into the operation new; this requires an additional parameter but otherwise this poses no problems. Here I shall not pursue this possibility.

The implementation of this may require allocation of some storage space for the value of $x[p]$. This allocation can take place at the very beginning of program execution, which is simple but not very efficient, or during the new-operation, or as part of the initialisation of $x[p]$. Conceptually though, this is irrelevant and all what matters about the new-operation is that it extends V by one (new and anonymous) value. (Compare this with the explanation of the standard procedure new in your favourite Pascal text.)

□

Now we have invented pointers! The problem with the Pascal-like languages, though, is that both the set V and the stored function x are left *anonymous*. In Pascal, for example, we have that:

Ω	is written as:	pointer to Int
$new(V,p)$	is written as:	$new(p)$
$x[p]$	is written as:	$p\uparrow$

So, my Ω is the set of all possible values of type pointer to Int and my variable V is the (anonymous) set of all such pointer values "in use". I am convinced that

much of the confusion about pointers is caused by this leaving anonymous of V and x; as a result it becomes virtually impossible to explain the mechanism in other terms than its low-level implementation. In my own experience, it helps very much to be aware of the fact that $p\uparrow$ is the application of a stored function to p; the only difference between this mechanism and arrays is that array indices are consecutive integers whereas the set of pointers in use (that is, V) has no visible structure. With arrays, for instance, expressions like $x[p{+}1]$ are meaningful, whereas $(p{+}1)\uparrow$ is absolute gibberish[0].

In the above I have confined myself to integer-valued functions, but this is not essential. In Pascal-like languages pointer to T is a valid type for (almost) any type T. The above discussion is applicable to every such type in isolation, that is, for every type T there is a type pointer to T and there are (anonymous) variables V_T and x_T associated with this type.

4 Multiple stored functions

A further disadvantage of leaving a variable anonymous is that the trick can be used only once. Suppose that we have need of *two* (or more) stored functions s and t, say, on the same domain V but with different types:

$$s : V \to S$$
$$t : V \to T$$

Now it is impossible to use a form like $p\uparrow$: should it denote $s[p]$ or $t[p]$? Fortunately, there is a way out: instead of using a pair (or tuple) of functions, we can also use a single function whose value is a pair (or tuple). That is, we use a single variable of type:

$$V \to S{\times}T \quad .$$

In a Pascal-like notation, a pair of values is represented by a record type, and in such a notation we would write:

pointer to record $s : S$; $t : T$ end .

Applications of functions s and t to an argument p then are encoded as follows:

$s[p]$	is written as:	$p\uparrow{\cdot}s$
$t[p]$	is written as:	$p\uparrow{\cdot}t$

It does not matter very much whether we write $s[p]$ or $p\uparrow{\cdot}s$: the two forms have the same meaning and the difference is only syntactical. The fact that a variable is left anonymous is even slightly remedied: only the combined function mapping V

[0] Mind you, we are discussing programming concepts here, not low-level languages like C.

to $S \times T$ is anonymous, but the two functions s and t are still explicitly named. This is a convenient, albeit somewhat unconventional, way to interpret the fields of a record referenced by a pointer p: the fields contain the values of the functions in point p of their common domain. That these fields are part of the *same* record just expresses that they are stored functions on the *same* domain.

5 Towards object oriented programming

For the sake of explicitness, we may decide to use a record type, as in the previous section, even in the case of a *single* function. So, instead of:

$$\mathsf{pointer\ to}\ T \quad ,$$

we use:

$$\mathsf{pointer\ to\ record}\ x : T\ \mathsf{end} \quad ,$$

and instead of:

$$p\uparrow \quad ,$$

we use:

$$p\uparrow \cdot x \quad .$$

Thus, even in the case of a single stored function, that function retains an explicit name.

Now it is only a small step to change the notation slightly: if we always adhere to the above convention, we can safely drop the symbol $\uparrow$ and write:

$$p \cdot x \quad \text{instead of} \quad p\uparrow \cdot x \quad .$$

But now we are back at our starting point again: it does not matter very much whether we write the application of (stored) function x to argument p as $x[p]$ or as $p \cdot x$. The advantage is that we can now forget about pointers and pointer dereferencing altogether; instead, we can maintain the mathematically simpler interpretation that x is a variable whose value is a stored function on an anonymous domain. Initially, this domain is empty and by means of an operation $new(p)$ the domain can be extended by one value, which value is also assigned to p. As before, the function value $p \cdot x$ can be initialised by means of an assignment.

In order to reflect this raise in the level of abstraction, we should adapt the syntax accordingly. Instead of:

$$\mathsf{pointer\ to\ record}\ s : S\ ;\ t : T\ \mathsf{end} \quad ,$$

we now had better write something like:

class $s : S$; $t : T$ end .

Now we have laid the foundation for object oriented programming! In the object oriented jargon, the values in the anonymous domain are called *objects* and the stored functions, like s and t, are called *attributes* of the objects. An operation like $new(p)$ now amounts to the *creation of a new object*. (Recall that, according to Section 1, the object thus created is not the variable p but the value of variable p.)

6 Epilogue

The previous section shows that we obtain an object oriented view by abstraction from certain aspects of the pointer mechanism. Conversely, we may say that the pointer mechanism provides a means for the implementation of objects and their attributes. Many a discussion about object oriented programming is troubled by not clearly separating these two issues, giving rise to questions about "value semantics" or "reference semantics" and much additional confusion; the mere possibility that an object oriented language may also provide pointers, to integers as well as to objects, adds much to this confusion. In my view, however, the basic concepts of object oriented programming can be explained in a simple way, without reference to pointers; pointers are no more than a means to implement these concepts, very useful as such, but they should play no part in a definition of the semantics. (Notice that I have arrived at my goal in a roundabout way: I have introduced pointers and then eliminated them again by abstraction. Knowing this, however, we can make the step towards objects right after the first half of Section 3.)

Many programming concepts have arisen as abstractions from what is possible in the underlying implementations. Pointers can be thought of as abstractions from the machine concept of storage addresses. Objects can be thought of as a further abstraction from pointers. This explains, but does not justify, why the explanation of such concepts often suffers from an insufficient separation of concerns, namely the separation between the concept as a mathematical entity and its implementation. It goes without saying that this separation is indispensible for the development of effective techniques for programming with these concepts.

Eindhoven, 21 october 1994

Rob R. Hoogerwoord
department of mathematics and computing science
Eindhoven University of Technology
postbus 513
5600 MB Eindhoven