The Locker Metaphor to Teach Dynamic Memory

Ricardo Jiménez-Peris, Marta Patiño-Martínez, J. Ángel Velázquez-Iturbide
Universidad Politécnica de Madrid
Depto. de Lenguajes y Sistemas Informáticos, Facultad de Informática
Campus de Montegancedo s/n, 28660 Boadilla del Monte, Madrid, Spain
{rjimenez,mpatino,avelazquez}@fi.upm.es

Cristóbal Pareja-Flores
Universidad Complutense de Madrid
Depto. de Informática y Automática, Escuela Universitaria de Estadística
Avda. Puerta de Hierro s/n, 28040 Madrid, Spain
cpareja@dia.ucm.es

Abstract

Some students experience difficulties when first introduced to dynamic memory. The goal of this paper is to present an analogy between dynamic memory programming and a real-world example that will help students in understanding the underlying concepts behind dynamic memory: a left-luggage room with lockers.

1 Introduction

Dynamic memory (i.e. pointer variables and their use) is a crucial topic in imperative languages and is covered in many courses, typically in CS2. It also constitutes one of the hardest topics in programming courses. The main problem seems to be that dynamic memory concepts are too abstract. Paradoxically, these low level concepts are embodied in languages such as Pascal, by means of nonintuitive, abstract mechanisms.

There are many ways of introducing programming topics that enhance comprehension, e.g. program animation. We use here a discourse that is commonly used in computer science to deal with abstract and arbitrarily defined concepts: metaphors. There are many examples of successful metaphors in this field: the desktop, the memory of computers, running programs, etc. Notice that the word "metaphor" here is loosely used to refer to any image that represents one thing and that is used instead of something else to explain it better [3].

SIGCSE '97 CA, USA
© 1997 ACM 0-89791-889-4/97/0002...$3.50

The use of analogies in teaching abstract concepts is often a key element in its pedagogy, as advocated in [2]. For instance, a well-known metaphor for introducing the critical section problem in teaching concurrent programming is that of Ben-Ari about igloos and eskimos [1].

In our work, we begin by introducing dynamic memory concepts intuitively by means of a metaphor. Once these concepts are well understood, they are taught in a general setting, as found in programming languages. We have chosen a concrete and well-known scenario, a left-luggage room, to ease the comprehension of the abstract concepts.

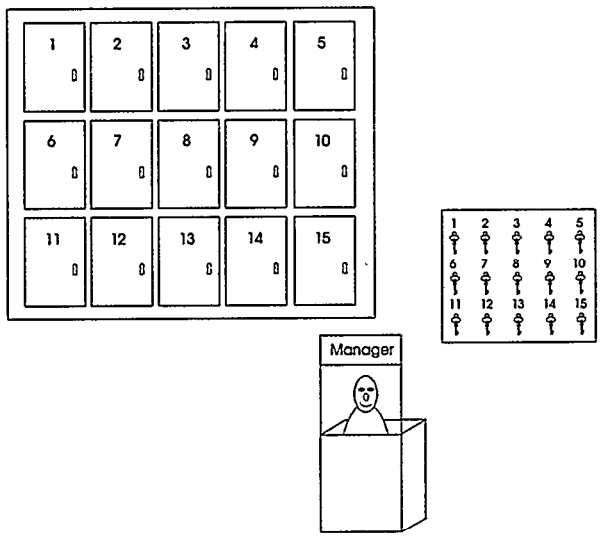
The structure of the paper is as follows. First, we present our metaphor and its analogy with dynamic memory concepts. In section 3, we include several common programming pitfalls, showing how the metaphor helps in revealing them. The two last sections include a discussion and our conclusions.

2 The Locker Metaphor

In this section, we present the elements of the metaphor. Each subsection presents new concepts and explains their correspondence in Pascal.

2.1 Left-luggage room and locker keys

A left-luggage room is a store of lockers, where each one can be opened by means of its corresponding key, as shown in the figure below:



Each key has a number which corresponds to the number of the locker it opens. The dynamic memory area is very similar to a left-luggage room, where memory cells are lockers and memory addresses are locker keys; as each locker can be opened with its key, a cell can be accessed by its associated memory address. Notice that neither keys nor memory addresses can be modified (both are literal values).

2.2 Keyrings

In our metaphor, keyrings are used to hold keys. Each keyring can hold at most one key (i.e. zero or one). Keyrings and keys play the role of pointer variables and address values, respectively, in imperative programming languages. Keyrings represent pointer variables that can hold a locker key (a memory address). A keyring holding no key represents the `NIL` value of a pointer variable. Two keyrings can hold a copy of the same key; in this case, they can open the same locker. This situation corresponds to the fact that two pointer variables can point to the same memory cell, that is, they can contain the same memory address.

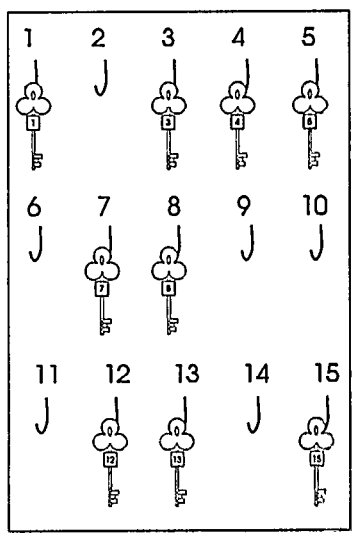
Simple pointer operations can also be explained graphically with the locker metaphor. Comparison of pointers (keyrings) is straightforward: it will be successful only if both keyrings are empty or the keys contained in both keyrings open the same locker.

Assignment of pointer variables corresponds to the following procedure: if the source keyring is empty, the target keyring is left empty; otherwise, a copy of the key in the source keyring is made, the old key in the destination keyring (if any) is thrown away, and finally the copy is introduced into the target keyring. An important property can be observed in the metaphor: when the contents of a keyring is copied into another, the key in the destination keyring is lost, with no chance of recovering it later.

2.3 Asking for a locker

When someone needs a locker, he or she goes to the locker manager office, gives the manager a keyring, and asks him for a locker. The manager will take any available key and put it in the keyring (the previous key, if any, is thrown away). Although the way the manager bookkeeps keys could be considered irrelevant to the user, it can help to fully understand certain kinds of errors. We can imagine that at the manager office there is a board with numbered hooks (as can be seen in the figure below), one per locker. If a key is hanging on a hook, then its locker is free, otherwise it is assigned.

In a programming language, the dynamic memory manager is in charge of cell bookkeeping. When a cell is needed, the `New` primitive is invoked, returning the address of a free cell. We can imagine some kind of bookkeeping similar to that of the manager, but its details are irrelevant.



2.4 Liberating a locker

When a locker is no longer needed, the user notifies the locker manager, and gives him the keyring with the key of the locker. The manager makes a copy of the key, hangs it up on its hook, indicating that the locker is available again, and returns to the user the keyring with the old key. Notice that surprisingly, the user keeps his copy of the key. This part of the metaphor corresponds to this situation: when a memory cell is not needed anymore, it must be freed by the

`Dispose` operation, but this operation could leave the pointer with its previous value.

2.5 Equivalent concepts

To summarize, a table with the equivalence between the metaphor and dynamic memory concepts is shown.

memory pool	left-luggage room
memory manager	locker manager
pointer	keyring
memory cell	locker
address	key of a particular locker
list of available cells	board hooks with a key
`k1, k2: ^Type`	*k1* and *k2* are keyrings
`New(k1)`	ask for a locker, get its key and put it in keyring *k1*
`k1^`	open the locker whose key is in *k1* to access its contents
`k1 = NIL`	check whether *k1* is empty
`k1 = k2`	check whether *k1* and *k2* are both empty or both contain keys that open the same locker
`k1:= NIL`	leave empty keyring *k1*
`k1:= k2`	if *k2* is empty then leave *k1* empty, else make a copy of the key in *k2* and put it in *k1*
`Dispose(k1)`	free the locker whose key is in keyring *k1* hanging a copy of the key on its board hook

3 Common Pitfalls

There are many examples of situations (i.e. uses of pointer variables) that the metaphor helps to illustrate graphically. We include in this section several common pitfalls. First, we give a description of each one in terms of the metaphor. An advantage is that the underlying concepts of dynamic memory can be illustrated independently from the programming language being used. For the sake of completeness, we include the corresponding Pascal code.

3.1 Copying pointers and cells

A frequent error occurs when students, intending to duplicate the contents of a cell, resort to assigning pointers. The source of this error is that they think that copying pointers implies the duplication of the pointed cell. Following our metaphor, the copying of a key simply results in having two keys of the same locker. Obviously, neither a new locker was given, nor were the contents of the locker duplicated.

Let us consider this fact in our metaphor by means of the following sequence of operations.

> *We have two keyrings* k1 *and* k2. *We ask the locker manager for a key and he puts it in keyring* k1. *The manager assigns us a locker (say, number 25). Now, we can open the locker with the key in keyring* k1 *and put $1,000 in. Afterwards, we duplicate the key in keyring* k1 *and put the copy in keyring* k2. *Then we use the key in keyring* k2 *to open the locker and take out the money. When we open again the locker with the key in* k1, *there is no money.*

In the example it is clear that copying keyrings (pointers) is not a way to make more money, because the keys are copied, but not the locker contents.

This example is equivalent to the following Pascal code segment:

```
VAR
  k1, k2 : ^INTEGER;

New(k1);
k1^:= 1000;
k2:= k1;
k2^:= 0;
Write(k1^)
```

that will display number `0` on the screen.

3.2 Memory leakage

A common programming mistake consists in forgetting to free unused cells. In the metaphor, it is evident that a locker can only be reused if its key is returned to the manager. On the other hand, if the locker key is lost, neither the locker will be used again nor will it be assigned to someone else, since no one has its key.

This mistake can be illustrated by the following example:

> *We have the keyring* k1. *We ask for a locker to the locker manager, whose key will be put in keyring* k1. *(Although it is irrelevant, we can suppose that we are assigned locker 47.) We open the locker with the key in keyring* k1 *and put $1,000 in. Finally, we throw the key in the river, leaving the keyring empty. When we try to open the locker, we find out that we do not have its key.*

The assigned locker (locker 47) will never be reused, as the locker manager should have received its key back. Even worse, we will not be able to open the locker in order to get our money.

The corresponding Pascal code is:

```
VAR
  k1 : ^INTEGER;

New (k1);
k1^ := 1000;
k1 := NIL;
Write (k1^)
```

The program will yield a run-time error.

3.3 Misuse of liberated cells

Another common mistake is trying to use cells that were previously liberated. In the metaphor, the situation is as follows. Once a locker has been freed, it can be assigned to someone else. However, the previous owner still keeps a copy of the key, so two different people can access the same locker, only one being authorized. As a consequence, the correct working of the locker system depends on the goodwill of its users. This situation can be illustrated by the following example:

> *We have keyring* k1 *and Peter has keyring* k2. *We ask the locker manager for a key to be placed in keyring* k1. *(Suppose we are assigned locker number 18.) Now, we can use the locker. Later, we free the locker (number 18), showing the key in* k1 *to the locker manager. Recall that we still keep the key.*
>
> *Later, Peter asks for a locker with keyring* k2. *Suppose that he is assigned the same locker (in this case, locker 18). Peter then puts $1,000 into the locker. If we open the locker now (remember we still keep the key of locker 18), we can take the money from it. When Peter opens the locker again, he will find out that there is no money.*

In Pascal:

```
VAR
   k1, k2 : ^INTEGER;

New (k1);
... use of k1^ ...
Dispose (k1);
...
New (k2); {it happens to point to
           the same cell as k1}
k2^ := 1000;
k1^ := 0; {the money is stolen}
Write (k2^)
```

Notice that this program is useful for illustrating a bad use of pointers, but reading the program is more didactic than running it. In fact, only when both variables are assigned the same address will it print the expected value of 0.

3.4 Uninitialized pointer variables

Another frequent error consists in using pointer variables that have not been initialized, as if they were pointing to a dynamic variable. In the metaphor this situation can be modelled as follows. At the beginning, keyrings can hold old keys (i.e. keyrings with previously used keys). For this reason, the keys they contain should not be used, since they open lockers we are not authorized to operate. To use a keyring for the first time, we have to empty it or to ask for a new locker with it. Thus, the old key is not used.

It is straightforward to think of a more concrete scenario and to show its correspondence in Pascal.

4 Discussion

In the previous sections, we have intermingled the metaphor and dynamic memory concepts. We have proceeded in this way in order to explain it to the reader, but the metaphor is usually used in a different way during lectures. We prefer to start first by introducing the left-luggage metaphor, and by highlighting the misunderstandings and errors described in section 3. Then, we introduce proper dynamic memory programming, relating its concepts to the metaphor when necessary, thus providing students with a concrete basis to refer to whenever a concept is too abstract to comprehend.

Our metaphor can be adapted to cope with several data types, by providing several kinds of lockers. However, as the metaphor gets more complicated, we prefer to restrict our examples to use a simple data type (e.g. INTEGER).

The metaphor can also be extended to deal with linked lists, by considering that the contents of the locker consists of two shelves, one to store an object (e.g. money), and the other for storing a keyring. The keyring in the second shelf can be used to hold a key of yet another locker; thus, lockers can be linked. This extension of the metaphor is useful to introduce linear data structures in a concrete way. However, we find it more adequate to use the metaphor in

situations similar to the ones expressed in this paper. Students can easily understand the corresponding Pascal code and can always refer to the metaphor when in doubt.

Metaphors used for learning in computer science fall into three overlapping streams [4]. Operational approaches focus on their measurable effect on learning. Structural approaches develop formal representation of relations between the source and the target domain. Pragmatic approaches acknowledge that metaphors are incomplete, but claim that their power may be attributed to such disparities. We designed our metaphor following the second approach because we wanted a bijection between the elements of the metaphor and those of dynamic memory.

Our metaphor has a number of important features, some of them because it is structural. First, it is simple since it refers to a familiar situation. Second, it has a rich structure, which enables a clear illustration of all the concepts and activities involved in handling dynamic memory. We can easily show correct and incorrect situations as illustrated by the pitfalls described above. Third, its expressivity enables us to explain dynamic memory in terms of this unique metaphor. This is an important advantage because mixing several metaphors can confuse beginners [3].

5 Conclusions

A pedagogic metaphor for introducing dynamic memory programming has been presented. The metaphor helps students to understand concepts of dynamic memory in a concrete way. It also allows them to work with dynamic memory before it is presented as a programming language mechanism, in a more abstract way. As a consequence, we can highlight errors in a natural way, thus paving the way for a better understanding of sometimes hard to grasp abstract concepts.

We have no formal statistics measuring the impact of the metaphor on our students. Nevertheless, our experience has shown that using the metaphor to illustrate abstract concepts has lead to an overall better understanding of the abstract concepts underlying dynamic memory management.

Acknowlegments

We want to thank Sami Khuri for his comments and suggestions which greatly helped to improve the paper.

References

1. Ben-Ari, M. *Principles of Concurrent Programming.* Prentice-Hall, 1982.
2. Cole, J. D. While loops and the analogy of the single stroke engine. *SIGCSE Bulletin 23*, 3 (Sept. 1991), 20-22.
3. Johnson, G. J. Of metaphor and the difficulty of computer discourse. *Communications of the ACM 37*, 12 (Dec. 1994), 97-102.
4. Madsen, K. H. A guide to metaphorical design. *Communications of the ACM 37*, 12 (Dec. 1994), 57-62.