On the Use of Pointers and the Teaching of Disciplined Programming

Sergio E. R. Carvalho and Miguel Angelo A. Nóvoa

Departamento de Informática, Pontifícia Universidade Católica
Rio de Janeiro, Brasil.

Abstract

In the past few years there has been considerable debate over the question of pointers in programming languages. Some maintain that pointers should not be allowed, while others try to restrict their use in a number of ways. In this paper we try to justify our view that pointers are a natural and useful way to teach beginners in Computer Science to manipulate list structures, provided a group of strong limitations is placed upon them. We define pointers in SPL, a language to teach beginners disciplined programming.

1. Introduction

The Department of Computer Science of the Catholic University in Rio de Janeiro offers undergraduate courses to students in Engineering. In particular, Eletrical Engineering students attend in large numbers to such courses, and in fact many of them, upon graduation, decide to pursue further education by enrolling in our M. Sc. program. Part of this "enthusiasm" towards Computer Science is due to the government policy of supporting teams of researchers in industries and in universities, to improve and even develop both software and hardware.

One of the most important courses offered to junior students is a course on Data Structures, where traditionally the book by Knuth [Knu 68] is used as text. In 1974, while preparing to teach this course, we decided to include as a major secondary objective the teaching of disciplined (top-down) programming [Wir 71a]. Immediately the choice of the programming language to be used presented itself as a problem. Among the programming languages available in the system, PL/1 was the one invariably used not only in the Data Structures courses, but also in most of the other undergraduate courses offered by the Department. To begin with we felt that PL/1 was not adequate to the teaching of disciplined programming, due to well-known problems as its size, its type conversions, the absence of a "case" construct, the undisciplined pointer facilities (see for instance [Hel 72, Zel 74]). Preliminary interviews with some students (in which they were asked to construct programs to manipulate a stack) confirmed what we further suspected: that due to the excellent knowledge of tricky programming in PL/1 the students would systematically delegate the task of program development to the background, in favour of the simpler task of coding.

We therefore decided to create yet another high level programming language, which (due to our absolute lack of inspiration) was called SPL (Simple Programming Language). SPL was to be used as an "explanations language"; algorithms shown in class coded in SPL would serve as models for the students' PL/1 programs. Should the experience be successful, SPL would then be implemented in our system. It turned out that, when asked to make comments on the course upon its completion, most students declared that:

(i) they felt that their programming style had considerably improved;

(ii) some characteristics of programming languages were now better understood, in the light of the simple facilities provided in SPL.

In this paper we present the SPL solution to the use of pointers. Recently, the use of pointers in high level programming languages has been under considerable debate. Hoare [Hoa 73a] maintains that pointers are low level constructs, and, as such, should not be present in high level programming languages ("Their introduction into high level languages has been a step backward from which we may never recover"). Although pointers are traditionally considered as a valuable aid in structuring data, Hoare has shown [Hoa 72, Hoa 75]that one can actually construct complex structured types without using pointers.

Records and pointers were introduced in SPL to allow users to more naturally represent linked lists. List elements are thus represented by record instances, having one or more fields of type pointer, which provide the linking mechanism, as we shall see. We felt that the use of well behaved pointers restricted to this situation would bring none of the well know problems connected with pointers in programming languages.

In section 2 we present a brief description of the SPL language. In section 3 the definition and the manipulation of records and pointers in SPL are described. An example is described in section 4.

2. Brief Description of SPL

SPL is a block structured language with standard scope rules, whose main features are:

(i) User defined data types: these were introduced to give the programmer more facilities to represent his way of thinking about a problem. This feature does not increase the power or flexi-

bility of the language, since only the renaming of standard types is allowed. Although abstract types [LZ 74] are not implemented, they may be included in a future version of SPL, but it is not clear yet that this may be a desirable feature in such a language.

(ii) Two types of storage allocation schemes: the first is controlled by the compiler, with allocations and deallocations taking place at block entry and block exit, respectively (used for all types, except RECORD); the second is under the control of the programmer, who, using the statements CREATE and FREE, can allocate and free storage for RECORD variables.

(iii) Full specification of procedures and functions in the prologue of a block: this helps debugging and documentation, since it becomes easy to see which routines may be called from inside a block. This feature also avoids the uneccessary duplication of declarations of routine names (an awkward feature of PL/1, for example).

(iv) Control of routine call side effects: this is done by the specification (which is mandatory) of the type of relationship between arguments and parameters.

(v) Access to global variables: also controlled by specifying in each block which variables have read only access in that block.

(vi) Control structures: a Pascal-like set [Wir 71b] which gives the programmer a powerful tool to structure his programs.

(vii) Absence of an unconditional goto statement: however, there exists a restricted type of goto, namely the EXIT statement, whose function is to leave a repetitive statement (while, repeat, and for loops).

(viii) Simple input/output statements, without format specification: this does not only simplify the task of reading and writing data, but also introduces a great deal of simplification in the compiler (the semantics of these i/o statements follows that of ALGOL W [WH 66]).

(ix) Record and pointer handling: this is the topic of the remander of the paper.

3. Records and Pointers in SPL

3.1. Generalities

The main characteristics of the SPL system, with respect to records and pointers, are:

(i) Compile-time binding of pointers to record classes, as in ALGOL W;

(ii) Pointer type fields of a record are allowed to point only to instances of variables declared of the same record type;

(iii) Possibility of user's allocation and deallocation.

Adoption of (i) above saves overhead in compiling: (i) and (ii) provide efficient type checking. The main consequence of restriction (ii) is that lists in SPL are homogeneous. We feel this is not a serious restriction, considering the purpose of SPL. SPL users have the ability to allocate and deallocate record instances; dynamic storage management in SPL is mainly programmer's responsibility. It is our opinion that, due to (i) and (ii) above, SPL is an adequate environment for the teaching of list manipulation.

3.2. The Definition of Records and Pointers

In the syntax rules that follow in this section, lower case strings of letters, with possibly one or more hyphens, are considered to be nonterminals; strings of capital letters represent reserved words in the system. Square brackets denote that the enclosed sequence of symbols may or may not be present, curly brackets followed by an asterisk denote the occurrence of zero or more instances of the enclosed sequence of symbols.

Records are defined as follows.

record-type → RECORD (field-list)

field-list → field-name-list: field-type
{; field-name-list: field-type}*

field-name-list → identifier-list

field-type → simple-type | LINK

A record is a list of fields each having a name (selector) and a specification of the type of values the field can hold. Such values can be either simple (INT, REAL, BOOL, CHAR(n)) or pointers to the record class being defined (LINK's). To illustrate, consider the following example:

```
TYPE person = RECORD (name: CHAR(12);
                      age, income: INT;
                      father, mother: LINK)T_END
```

TYPE and T_END are delimiters for type definitions. In the above a record class called "person" is defined, having "father" and "mother" as selectors for fields of the type LINK.

Pointer types are defined as follows.

pointer-type → POINTER (record-class-identifier)

record-class-identifier → identifier

Record class identifiers are names which are assigned to record types through a type definition ("person" in the example above). Note that in this way pointers are syntactically bound to record classes.

As an example, consider the pointer declaration below:

```
DECLARE P1, P2, P3: POINTER (person) D_END
```

P1, P2 and P3 are assigned to memory positions which can hold addresses of instances of the record type "person".

3.3. The Manipulation of Records and Pointers

A record definition does not cause memory to be allocated. This is done by the programmer, through the CREATE statement. Suppose A1 is the name of a variable declared of a certain record type R, and that P1 is the name of a pointer to instances of the same record R. Then the statement:

```
CREATE A1 SET(P1)
```

causes the allocation of an area in memory compatible with the description of the record class R and sets P1 to point to this area. The general form of a CREATE statement is a follows:

```
create-statement →
      CREATE record-identifier SET
      (pointer-identifier-list)
```

where record-identifier is the name of a variable of type record, and pointer-identifier-list is a list of names of pointers to that same record.

Record instances can be deleted explicitly by the programmer. The execution of a statement of the form

```
free-statement →
    FREE record-identifier REF
    (pointer-identifier-list)
```

causes the memory positions occupied by the instances of the record identifiers in the list to be returned to the list of available space. After this takes place, all pointers in the list have their values set to NIL.

Pointer assignment statements are allowed in SPL. Their form is as follows:

```
pointer-assignment-statement →
      pointer-variable := pointer-value
```

Pointer variables are defined as follows:

```
pointer-variable →
      simple-pointer-variable
      {→record-identifier.link-identifier}*

simple-pointer-variable → simple-variable

simple-variable → identifier [(subscript-list)]
```

Pointer variables can be either simple or qualified. Simple pointer variables are either identifiers (declared of type POINTER), components of arrays of pointers or function calls returning a pointer value. Qualified pointer variables allow programmers to access instances of records through the link type field of records in the class.

Pointer values are either the special value NIL or a pointer variable.

```
pointer-value → NIL | pointer-variable
```

Pointers can help access record fields of types other than LINK whose values can then be used in expressions. The only other operations allowed on pointers are the comparison operations (= or ¬ =).

4. On the Teaching of Pointers and Records

Pointers and records were introduced with the study of linked allocation. An element (node) of a list was first presented, and the idea of a field containing an address of a node as its value was introduced. As expected, this was well accepted by students with a prior knowledge of assemblers. Full acceptance, however, came only with an explanation of node insertion and deletion. In this initial explanation, no programming language notation was used: instead, rectangular boxes and arrows represented nodes and links, respectively. Using this same representation, we mentioned the "dangling reference" and the "inaccessible and still needed data item" problems.

Our next step was to show how algorithms in linked allocation were represented in the SPL system. First we illustrated how records could be used as node templates. Then algorithms for insertion and deletion were developed using SPL. Finally, a SPL program to add polynomials was presented.

As seem above, our approach was what we believe to be a "natural" one. It is important to note, however, that the notion of pointers was introduced in a "smooth" way: during the time linked allocation was presented, the students were really concerned with problems like insertion and deletion; it was not apparent at all that any students were having problems with the notion of pointers and records. We believe this was due at least in part to the way SPL was designed, encouraging an adequate use of pointers.

More complicated linking mechanisms were presented along the course, involving the creation and deletion of instances of records with multiple LINK fields. We were able to observe that:

(i) in a short period of time the sudents were skillfully writing SPL programs dealing with records and pointers;

(ii) hand simulating a sample, few errors were delected due to dangling references or inaccessible data items.

5. Conclusions

SPL is a programming system designed for beginners in Computer Science. As such, some of its features were given special attention. In particular, language structures which in other systems may give rise to undisciplined programming were carefully adapted to meet the purposes of SPL. In this paper we showed how pointers and records are defined and manipulated in the SPL system. We stated that the only reason for the presence of pointers and records in our language is to give programmers the ability to model and manipulate list structures in a more natural way. Some well known problems associated with the use of pointers in programming languages (type violations and dangling references) were avoided in our system.

From the teaching experience point of view, we conclude that if adequate languages for teaching programming (as Pascal) are not available in the system, then it is better to develop and implement a new language than to use a language which, by its complexity, would obscure the main issues at hand.

6. Bibliography

[Hoa 72] - Hoare, C.A.R., "Notes on Data Structuring", in "Structured Programming", Academic Press. 1972, 83-174.

[Hoa 73a] - Hoare, C.A.R., "Hints on Programming Language Design", Proc. ACM SIGACT/ SIGPLAN Symp. on Principles of Programming Languages, Boston, Oct 73.

[Hoa 75] - Hoare, C.A.R., "Data Reliability", Proc. 1975 International Conference on Reliable Software, SIGPLAN Notices 10:6, Jun 75, 528-533.

[Hol 72] - Holt, R.C., "Teaching the Fatal Disease", Report RCH-1, Dept. of Computer Science, Univ. of Toronto, Dec 1972.

[Knu 68] - Knuth, D.E., "The Art of Computer Programming", volume 1, Addison-Wesley, 1968.

[LZ 74] - Liskov, B. & Zilles, S., "Programming with Abstract Data Types", SIGPLAN Notices 9:4, Apr 74, 50-59.

[WH 66] - Wirth, N. & Hoare, C.A.R., "A Contribution to the Development of Algol", Comm. ACM. 9:6, Jun 66, 413-432.

[Wir 71a] - Wirth, N., "Program Development by Stepwise Refinement", Comm. ACM 14:4, Apr 1971, 221-227.

[Wir 71b] - Wirth, N., "The Programming Language Pascal", Acta Informatica 1, 1971, 35-63.

[Zel 74] - Zelkowitz, M.V., "Pointer Variables within a Diagnostic Compiler", Tech. Ref. TR-343, Dept. of Computer Science, Univ. of Maryland, Dec. 74.