Pegs & blocks
A Concrete Model for Teaching Basic Programming Structures

Tim Wöhrle and Debora Weber-Wulff, Technische Fachhochschule Berlin

Abstract

To help students to get started with programming, a model of wooden bricks was developed which promotes understanding of basic terms such as "pointer", "type" or "binary tree". The model offers an intuitive understanding of coherences, and is suitable for simulating pointer-based algorithms such as the construction of a binary tree.

1. Introduction

In computer science as well as in any other field it is necessary for beginners to understand the basic structures on which further learning will be based. Many of these basic programming structures are quite abstract and seen to have no parallels in the everyday world. Students who are not used to operating with abstract models, e.g. student without a good mathematical background, often have extreme difficulties grasping these basic notions.

To alleviate this situation and to offer concrete examples with well-known objects, a project was initiated at the polytechnical college Technische Fachhochschule Berlin (TFH) called „Informatik Be-Greifen“, grasping /understanding computer science. As a part of this project, a model of building blocks was developed in order to explain basic programming structures such as data types, operators or pointers. The model is also suitable for simulating pointer-based algorithms step-by-step. Problems that occur while inserting nodes into a tree, for example, can be intuitivly understood. The model has been successfully used in the second author's "Programming I & II" lectures at the TFH Berlin since 1995.

2. Basic elements

The basic elements of the model are small wooden bricks which can be bought in a toy store or hobby shop. They will be used to represent variables, values and constants. A variable is represented by a brick with holes drilled on top in a specific pattern on its top and with space on front

to record its name. A value is represented as a block with pegs in the same positions as the holes on the corresponding variable type block. Because of these pegs it is not possible to place a value directly onto the table which represents storage for variables; it must be plugged into a variable in order to be stable. To represent different data-types (integer, character, string) the variable and value bricks are designed with different arrangements of pins. Since the pin arrangement is not visible to the students past the first rows of the classroom, each type of variable or constant is assigned a color. Because of the drill patterns, it is not possible to place a value on an variable of the wrong type. The only way to do this is to use a type conversion block as shown below.

Fig. 1 shows the different arrangements of the drilled holes on top of variables.

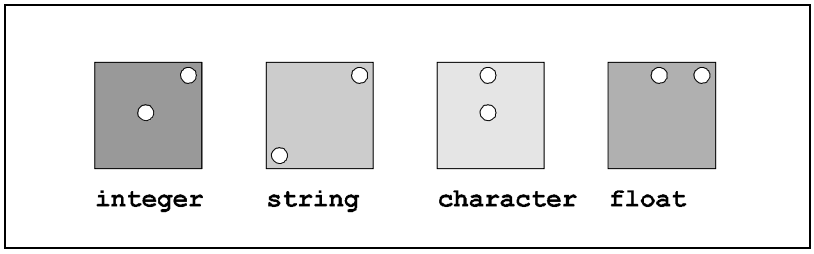

Fig. 1: Arrangements of holes for different types

The bricks which represent constants do vary at one point from the usual value-variable pair: While the value of a variable can be replaced by another value, the constant's value is glued to the variable and can not be modified by being takien apart.

Another basic term explained by the model is the value range of variables. Each value-brick has a limited space on the front to note the value. A larger value requires a *long* variable of the same type (the same color), but with double the width. A long variable has more place to store larger values, but needs as twice as much room on the table as short variables. If only long variables are used, memory decreases faster.

Fig. 2 shows example blocks from the model and the corresponding Pascal -statements.

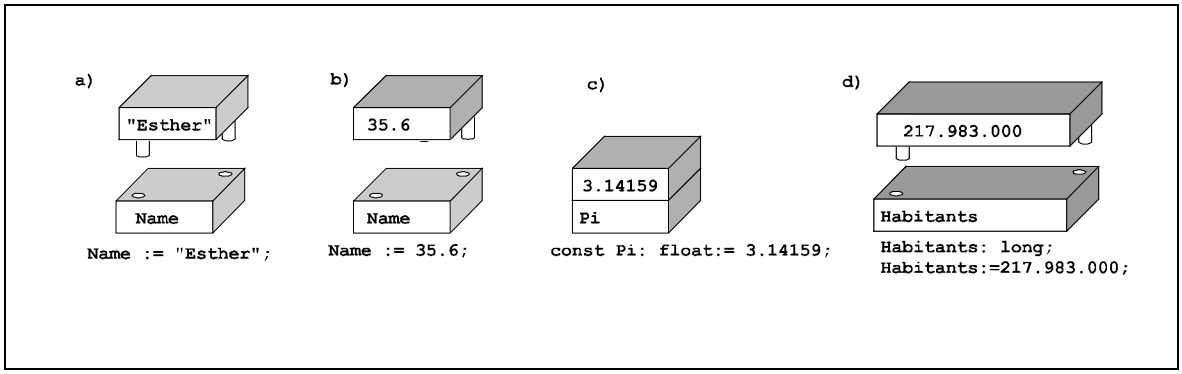

Fig. 2: Basic elements: a) a variable with a value of the correct type ,b) a variable with a value of the wrong type , c) a constant, d) a long variable with a fitting value

3. Compound types

To enable a user to compose structured data types with the model, the variable bricks have holes on their sides as well. Pegs can be used to plug a number of bricks together. Variables of the same type can be combined to one- or two-dimensional arrays, those of different types to records. On the front of the array components, instead of the name an index is noted. Array and record names are placed on one side. The components of records are named, but not numbered. The values of arrays and records are accessable component-wise; it is not possible plug together a number of values.

Fig. 3 shows the representation of arrays and records in the model.

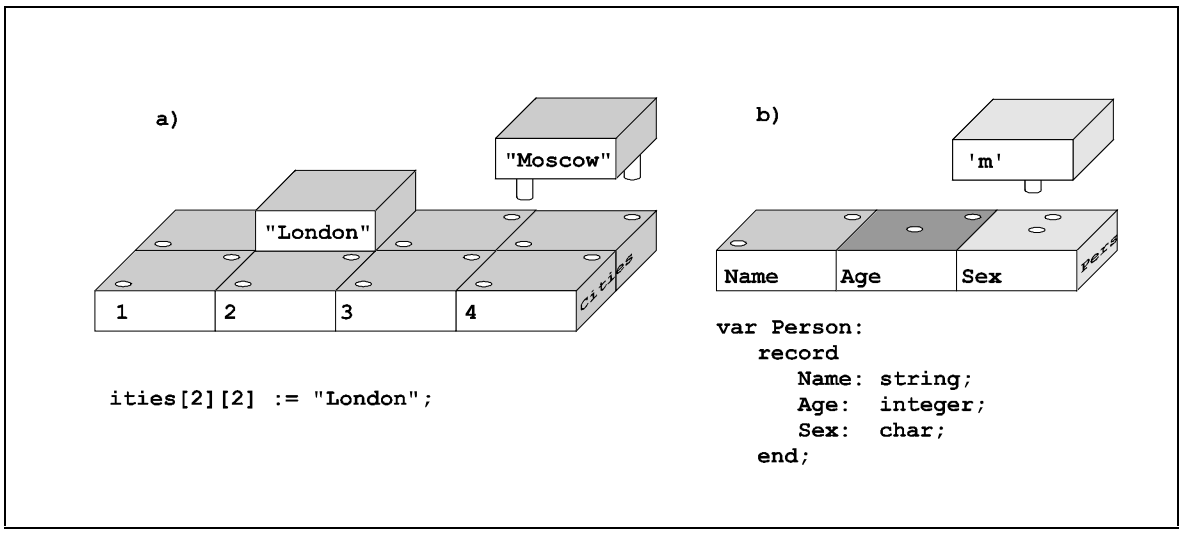

Fig. 3: Compound types: a) a two-dimensional array, b) a record

4. Pointers

To represent pointers in the model, it is necessary to introduce a new kind of brick. It is provided with a string glued onto its top which is connected to a peg. The peg fits into the side-holes of other variables (the side-holes are smaller then a variable's top holes; therefore the peg can't be plugged into a wrong hole). A pointer brick can't carry a value itself, but it can point to any other kind of variable. The pointer-bricks are colored black; they can be typed with a colored sticker, if the programming language to be taught uses typed pointers, as in Ada. The pointers can point to variables, structured variables and of course to other pointers. Since pointer bricks are provided with holes on their sides, it is possible to use them as components in structures. A good example for this is in nodes of linked lists or binary trees.

Fig. 4 shows the usage of pointers.

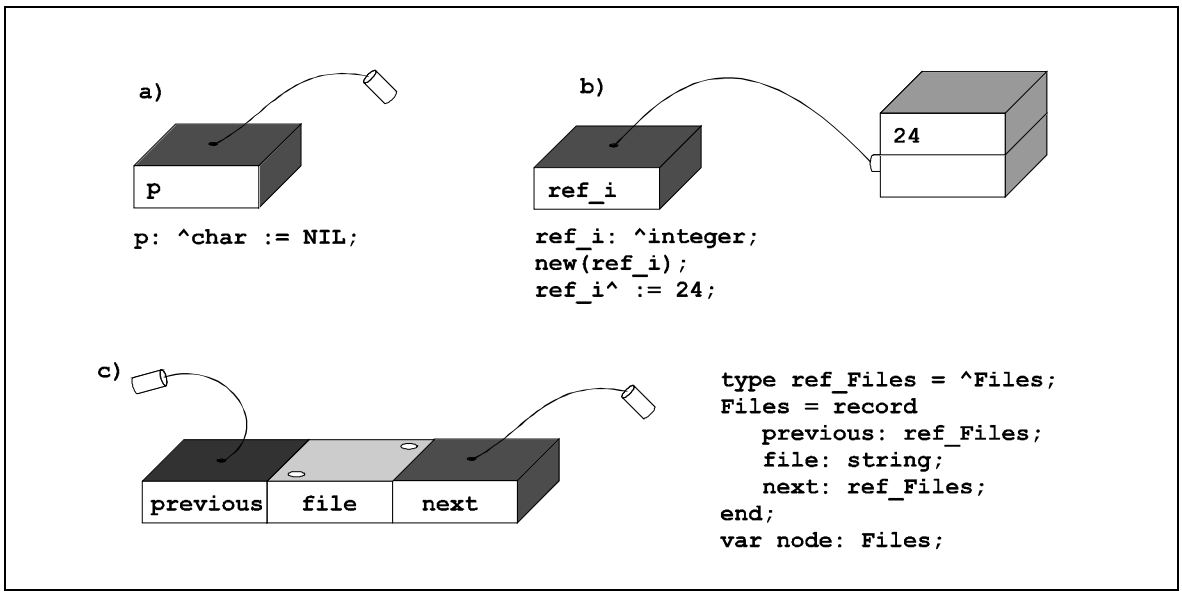

Fig. 4: Pointers: a) a pointer to NIL, b) a pointer to an integer-variable, c) the node of a forward-backward linked list

With the introduction of pointers, some basic ideas of memory management can also be shown. Every memory device (e.g. an overhead projector) has limited space to place variables on. The memory consists of two parts: a fixed part, on which the conventional variables can be placed (the *data segment*), and the remaining part (the *heap*). The data segment can be represented by a box on a slide which has a fixed size. To use the remaining storage, a dynamic data structure is necessary. A root pointer standing in the data segment points to a structure in the heap that has been dynamically allocated. This is the first node of a linked list or a tree. The structure can be expandend until the heap is full, but it is important that the connection between the data segment and the dynamic

structure is never interrupted. The shadows of the storage projected by the overhead projector give a very good picture of what goes on during dynamic storage allocation. In the lab, a table with a sheet of paper as the data segment can be used.

This concept it allows one to play through pointer based algorithms such as inserting, deleting or searching nodes in a list. When a bucket of blocks and pegs is placed in the lab, students will pick them up and play through the algorithms as they have been demonstrated before programming. In the first semester to use this technique there were no errors in using dynamic data structures in the exercises - this is a concept that has been truely grasped.

5. Operators

To process and convert values in the model, a representation for operators can be introduced. An operator is fitted with holes on top *and* with pins at the bottom. This makes it possible to plug one or more values on the operator and to plug the whole construction onto a variable of the resulting type. Input and output types do not have to be the same kind. Additionally, it is possible to build unary, binary or higher operators. As in the variable blocks, the operator's name can be written on the front.

A special kind of operator is a type-casting operator. It has holes for the source type on the top and pins for the resulting type at the bottom. Type-casting operators are the only way to plug values on variables of another type. Fig. 5 shows examples for some operators.

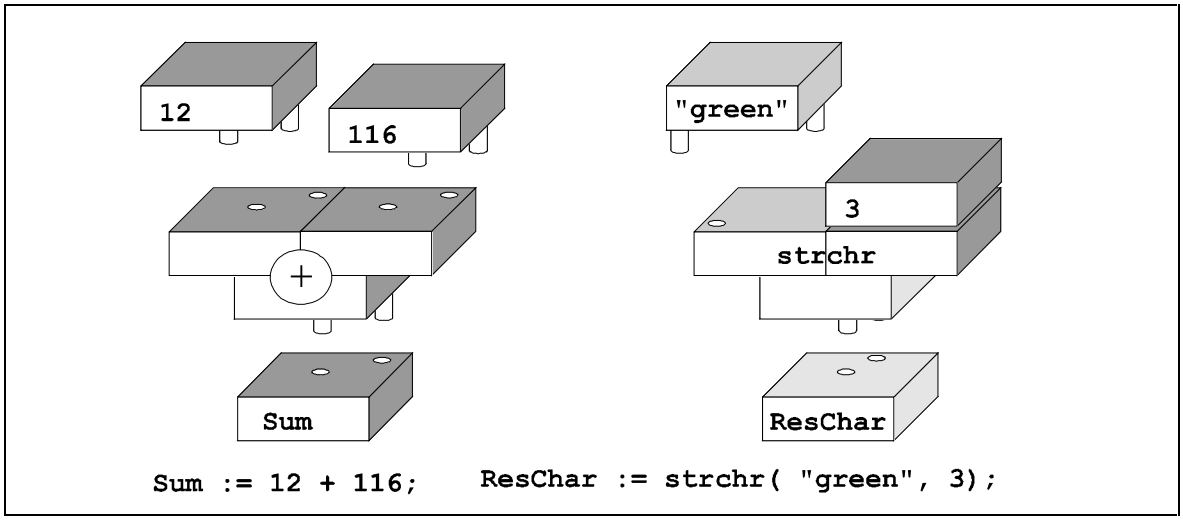

Fig. 5: Operators: a) a binary arithmetical operator , b) an operator with different input and output types (the n-th character of a string)

6. Type declarations

In the previous sections, variables of a special type were declared, which needed space on the table. Type declarations were just implicitly used. Declaring types explicitly in a program causes no memory allocation, and therefore it would be misleading to use bricks for the representation of type declarations. A better solution is the usage of file cards which illustrade the stucture of the type (e.g. the components of a record). That concept indicates what type declarations are: building constructions for variables. With one type declaration-card, a number of variables of this type can be built. Fig. 6. shows the type declaration and a variable of this type.

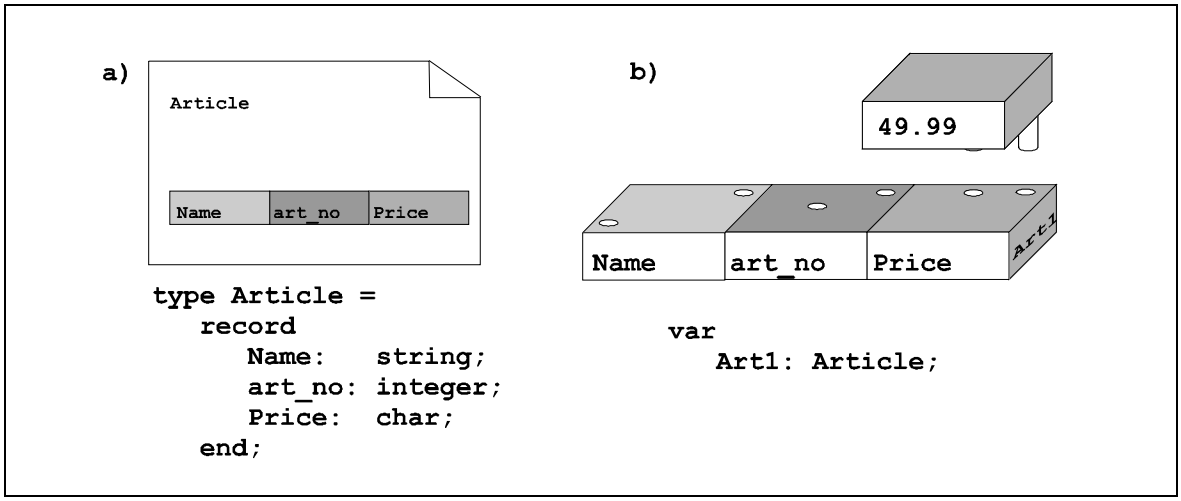

Fig. 6: Type declaration. a) the file card, b) a named varable of this type

7. Cost and used materials

For the project, a set of 100 stones was built, that includes all the descriped elements. To build this set, the following materials with a total price of DM 200,-- (about $ 150,--) are needed:

- colorless wooden bricks
- different suited colors (plus black color for the pointers)
- glue
- wooden pegs in two sizes (for the top and the side holes)
- string for the pointers

8. Literature

Pilz, Eva and Weber-Wulff, Debora: Der Einsatz von Bildern beim Lernen von Programmieren, Leuchtturm-Schriftenreihe Ingenieurpädagogik, Band 35 S. 393-398, Albach 1995