Data abstraction for Pascal programmers

by VISWANATHAN SANTHANAM and JOHN R. POTOCHNIK
Wichita State University
Wichita, Kansas

ABSTRACT

Lack of data abstraction facilities in Pascal is a serious shortcoming for a language whose principal aims include “teaching programming as a systematic discipline.” In this paper, a scheme is presented to implement data abstraction in Pascal using a limited form of the class structure defined in Concurrent Pascal. The class structures are translated to equivalent (sequential) Pascal constructs with the help of a preprocessor whose design is also described.
INTRODUCTION

Use of abstraction techniques to improve program reliability and programmer productivity can be traced back to early computer languages such as FORTRAN. Subroutines and functions of most algorithmic languages represent abstraction of operations. However, it is only in the past decade or so that programming language constructs have been devised to extend the idea of abstraction to data as well as operations in a unified manner. The term data abstraction has been applied to implementation-independent characterization and usage of data. Among the languages that provide data abstraction capabilities are MODULA, Concurrent Pascal, and Ada.

Most notable among the recent programming languages that do not support data abstraction is Pascal—a language that is growing in use, especially among small computer users and educators. The reason for this omission is perhaps related to the desire on the part of its architect to keep the language simple. Whatever the reason, the absence of even a rudimentary form of data abstraction capability has to be termed a vacuum in the language. This paper describes a scheme that will fill this vacuum.

Interestingly, most languages that currently support data abstraction also support concurrent programming, leading a casual observer to believe that data abstraction somehow naturally belongs in the realm of concurrent programming. The coincidence is largely due to the fact that the need for orderly sharing of data between concurrent processes naturally leads to centralizing such data and related operations. Though there is no particular need for hiding the implementation details to make this sharing work, it turns out to be relatively easy to add this quality to the notion of centralization. The concurrent data- (or resource-) sharing facilities are termed monitors; the sequential abstraction facilities are termed classes.

This work presents a data abstraction facility for (sequential) Pascal users. The Pascal language is extended to include a form of class structure similar to that of Concurrent Pascal. However, unlike Concurrent Pascal and other extensions of Pascal that provide similar capabilities (see Steensgaard-Madsen, for example), there is no need in this approach for a special compiler or operating environment to compile or execute a program. We present the design of a “preprocessor” that will translate the extensions to suitable constructs in sequential Pascal. This approach has the advantage of being applicable to existing Pascal systems. The source-to-source translation approach has the additional pedagogical value of illustrating a moderately simple technique to realize data abstraction in languages that do not support such a facility.

EXTENSIONS

The central theme of our approach is to extend sequential Pascal to include classes. A preprocessor will then accept programs in this extended language and translate them to equivalent programs in the standard language. The first extension is the introduction of a new type called class. Its syntax and semantics are essentially the same as those in Concurrent Pascal, with minor variations:

type CLASSNAME = class [(<formal parameters>)];
<class block>

where the optional formal parameters serve to instantiate (expand) the class for a specific application. For example,

type STACK = class(ELEMTYPE, MAXSIZE);
would allow us to define a generic class type for a variety of different element types and stack sizes. The preprocessor will treat these parameters like macro parameters, replacing them with the texts of the corresponding actual parameters when the class block is expanded.

The class block is similar to an ordinary subprogram block in Pascal, with a few differences. Any function or procedure within a class block may be designated as an entry function or procedure. The syntax for this extension is defined by

procedure entry PROCNAME [(<formal parameters>)];
and

function entry FUNCNAME [(<formal parameters>)];

For example,

function entry POP: ELEMTYPE;

Entry subprograms must be local subprograms within the class and not be nested deeper. The visibility (scope) of such subprograms is the same as that of the class block itself.

The statement part of a class block consists of an initialization sequence with this syntax:

begin init
<statement sequence>
end;

Defining a class type allows the programmer to declare class variables of that type. This is done by a var statement, as is done for declaring any other variable:

var <variable name list> : CLASSNAME [(<actual parameters>)];

The actual parameters, if any, textually replace the corresponding formal parameters in the class block when the latter
program EXAMPLE( INPUT, OUTPUT );

type STACK = class( ELEMTYPE, MAXSIZE );
type STKTABLE = array[1..MAXSIZE] of ELEMTYPE;
var PTR: INTEGER;
  STK: STKTABLE;
function entry EMPTY: BOOLEAN;
begin
  EMPTY := PTR<1
end;
function entry FULL: BOOLEAN;
begin
  FULL := PTR[MAXSIZE
end;
procedure entry PUSH( OBJ: ELEMTYPE );
begin
  if not FULL then begin
    PTR := SUCC(PTR);
    STK[PTR] := OBJ
  end
end;
function entry POP: ELEMTYPE;
begin
  if not EMPTY then begin
    POP := STK[PTR];
    PTR := PRED(PTR)
  end
end;
begin init
  PTR := 0
end;

var S1, S2: STACK( INTEGER, 100 );
I: INTEGER;

begin (* MAIN PROGRAM *)
  while (not EOF) and (not S1.FULL) do begin
    READLN( I );
    S1.PUSH( I )
  end;
  while not S1.EMPTY do
    S2.PUSH( S1.POP )
end.

Figure 1—An example of data abstraction using class

is instantiated. Thus it is possible to declare several distinct
classes from a single type definition. For example,

var S1: STACK(INTEGER,100);
SR: STACK-REAL,200);

sets up an integer stack and a real stack of different sizes from
the same class type STACK.

Operations on class variables are restricted to those defined
by entry subprograms in their class types. Invocations of these
operations appear in the form of procedure and function calls
qualified by the class variable name. For example, S1.PUSH
(-401) invokes the PUSH operation (an entry procedure)
within the class type of S1.

To illustrate these extensions and their role in data abstrac-
tion, an example of a complete program is shown in Figure 1.
The program contains the definition of a class type called
STACK. Two parameters, ELEMTYPE and MAXSIZE, will
allow different instantiations to be derived from this single
generic definition. The class block includes several definitions
and declarations that are private entities within the class
block. They represent the implementation details of the ab-
stract data type STACK. There are four entry subprograms
within the class block. They represent the predefined opera-
tions on the STACK that are available to the creators of
STACK-type variables. They enclose the implementation de-
tails of the respective operations, as any subprogram does.
The initialization sequence for the class consists of setting the
stack pointer PTR to zero.

The main program includes a declaration for two integer
STACKs, each capable of holding up to 100 elements. The
statement part of the program reads integers from the input
and pushes them on S1 stack until it gets full or the input is
exhausted. Then, the elements are transferred from S1 to S2
one at a time.

Though the class construct described in this section is simi-
lar in most respects to that of Concurrent Pascal (see Hansen
for an extensive discussion on the nature and use of classes),
there are some important differences. Both systems allow
parameters to be specified in a class definition, but for differ-
ent purposes. Also, in Concurrent Pascal, class variables have
no special restrictions on where they may be defined or how
they may be used. In our system, they may not be part of a
larger data structure, such as an array or a record, nor may
they be passed as parameters to subprograms. These latter
restrictions are necessary for circumventing the limitations of
Pascal and difficulties of translation.

PREPROCESSOR

The definition of a class type is treated by the preprocessor as
a macro-definition. The macro itself is expanded (instan-
tiated) only when a variable of that class type is declared.
During the expansion the actual parameters replace the cor-
responding formal parameters in the class definition. This
substitution alone will not yield an acceptable Pascal program.
Several other transformations are needed, and those are de-
scribed in this section.

A class block has a dual function. In most respects it is like
any subprogram block enclosing private definitions of con-
stants and types and declarations of variables and sub-
programs. However, there are some key exceptions to this
similarity rule. Entry subprograms are not strictly private to
the class block; they are callable from blocks that declare
variables of that class type. (These latter blocks are called host
blocks.) This is in contrast to the usual visibility rules of
Pascal. Additionally, variables declared within the class block
cannot be treated like ordinary local variables, because they
are to be allocated when the host block is entered and retained
throughout the lifetime of the host block. In other words,
their creation and destruction must coincide with the creation
and destruction of variables in the host block. But their visi-
tibility is to follow the usual block structure rules.

There seems to be no simple way to accommodate the dual
characteristics of a class block in Pascal. Our approach is to
strip the class block of its enclosing property but retain the
privacy of identifiers defined within it by transforming them to
unique identifiers that do not conflict with those in the host block. The transformed definitions (constants, types) and declarations (variables, subprograms) are added to the respective groups in the host block. Of course, there is no need to transform names that are locally defined within the subprograms in a class block, because their uniqueness is guaranteed by the usual scope rules of block structure.

The above scheme nicely solves the dichotomy of entry subprograms. Type and constant definitions are also handled adequately by this transformation. However, variables and labels of the class block must be handled somewhat differently.

Variables in the class block represent actual data structures required to implement the abstract data type. Each class variable declared in the host block would require its own set of these data structures. For example,

```pascal
var S1, S2: STACK(INTEGER,20);
```

declares two distinct variables of the abstract type STACK. If STACK is implemented by using a linear array in the class (as in the example of Figure 1), the expansion of the class definition must result in two distinct arrays. This is achieved in our scheme by introducing into the host block one set of class block variables for each class declared in the host block (see Figures 1 and 3 for an example). The name transformation technique will ensure that there are no conflicts in the host block.

The duplication of class block variables could lead to other duplications during the instantiation of a class. For example, a subprogram in the class block may reference a class block variable. This could mean that we need to duplicate such subprograms, making one copy for each class variable declared in the host block. This could be costly if there are several such subprograms and/or they are lengthy. Our approach avoids such duplication by eliminating any direct reference to class block variables within a subprogram. Instead, all data structures of the class block are passed as reference (var) parameters to each subprogram. Though it may not be necessary to pass all data structures to all subprograms, this simple scheme avoids the need for scanning the subprograms to determine which data structures are needed. In addition to modifying the subprogram definitions, instances of calls to these subprograms are also modified by adding a corresponding set of actual parameters. In the case of calls originating in the host block, the additions consist of class block variables associated with the specific class variable qualifying the call. For calls within the class block, the additions consist of the same names as those added to all subprogram headers. (See Figures 1 and 3 for examples.)

The initialization sequence of the class block is transformed into a special procedure. This procedure is invoked once for each class variable in the host block by including suitable calls at the top of the statement part of the host block (see Figures 1 and 3 for example). The name for this special procedure is derived from the class name so as to avoid conflicts, and all class block variables are passed by reference to this procedure as well. If there are any labels declared in the original class block, they must refer to statement labels in the initialization sequence. Therefore, the class block label declarations, if any,
program EXAMPLE( INPUT, OUTPUT );

```
type STACK#STKTABLE = array[1..100] of INTEGER;
var S1#PTR, S2#PTR: INTEGER;
S1#STK, S1#STK: STACK#STKTABLE;
I: INTEGER;
```

```
function STACK#EMPTY( var STACK#PTR: INTEGER;
var STACK#STK: STACK#STKTABLE ): BOOLEAN;
begin
STACK#EMPTY := STACK#PTR<1
end;
function STACK#FULL( var STACK#PTR: INTEGER;
var STACK#STK: STACK#STKTABLE ): BOOLEAN;
begin
STACK#FULL := STACK#PTR>100
end;
procedure STACK#PUSH( var STACK#PTR: INTEGER;
var STACK#STK: STACK#STKTABLE; OBJ: INTEGER );
begin
if not STACK#FULL(STACK#PTR, STACK#STK) then begin
STACK#PTR := SUCC(STACK#PTR);
STACK#STK[STACK#PTR] := OBJ
end
end;
function STACK#POP( var STACK#PTR: INTEGER;
var STACK#STK: STACK#STKTABLE ): INTEGER;
begin
if not STACK#EMPTY(STACK#PTR, STACK#STK) then begin
STACK#POP := STACK#STK[STACK#PTR];
STACK#PTR := PRED(STACK#PTR)
end
end;
procedure STACK#INIT( var STACK#PTR: INTEGER;
var STACK#STK: STACK#STKTABLE );
begin
STACK#PTR := 0
end;
```

begin

```
STACK#INIT( S1#PTR, S1#STK );
STACK#INIT( S2#PTR, S2#STK );
```

while (not EOF) and (not STACK#FULL(S1#PTR, S1#STK)) do begin
READLN( I );
STACK#PUSH( S1#PTR, S1#STK, I )
end;
while not STACK#EMPTY(S1#PTR, S1#STK) do
STACK#PUSH( S2#PTR, S2#STK, STACK#POP(S1#PTR, S1#STK) )
end.
```

Figure 3—Translation of program in Figure 1

their invocations) are in place. The update file consists of texts to be inserted at selected points in the host block. Each update is keyed for random access by a special marker. A copy of the marker is placed in the modified source file where the corresponding text is to be inserted. The points within the host block where potential insertions may take place are only a few:

1. Constant definitions
2. Type definitions
3. Variable declarations
4. Subprogram declarations
5. Top of statement part

The design of the output files from the first phase of the preprocessor minimizes the complexity of the second phase by reducing its function to one of scanning the modified source file, sequentially looking for markers, and merging in the corresponding text from the update file. The advantage, of course, is that the merging phase needs to have no knowledge of the syntax of Pascal. The overall design of the preprocessor is depicted in Figure 2.
EXEMPLARYs

A set of three graduated examples is presented in this section to illustrate the use of class structures, their translation, and the translation process itself.

In the first example, the abstract data type STACK of Figure 1 is revisited. The translation of the EXAMPLE program appears in Figure 3. For the sake of better readability, the name transformations are shown as concatenations of component names from which they are derived. For example, STACK#PTR stands for a unique name to be derived from STACK and PTR. In practice, such concatenation may be too simplistic to avoid conflicts. Nevertheless, by placing some restrictions on the use of special characters (such as #), it should be possible to retain a level of readability in the “production” translation comparable to that of the translations shown here.

The new additions to the host block EXAMPLE due to the instantiation of the class STACK by the declaration of Sl and S2 are enclosed in boxes for easy identification. The type name STACK#STKTABLE, for example, comes from the type definition for STKTABLE in STACK class. It is noted that in the type definition for STACK#STKTABLE and throughout the remainder of the class the formal parameters ELEMTYPE and MAXSIZE have been replaced by the actual parameters INTEGER and 100 respectively.

Each local subprogram of the class block has been extended to include the data structures originally defined as variables in the class block. The actual parameters list of external calls to these subprograms has been extended to include the data structures originally defined as variables in the class block.

Figure 4 shows a second example involving two distinct instantiations of the same class. Two important features of the preprocessing step are borne out by this example. Each instantiation (one for QI and another for QR) generates its own set of additions to the host block. It is not sufficient, therefore, to employ just the class name QUEUE to transform class block names. For example, if we generated QUEUE#QTABLE from QTABLE for the first instantia-
tion, it would conflict with the name to be generated during the second instantiation. In our preprocessor, this situation is handled by using a sequence number (in conjunction with the class name) to generate unique names for each instantiation. Thus, the first instantiation would yield the name QUEUE1#QTABLE (abbreviated as Q1#QTABLE in Figure 4), and the second instantiation would yield QUEUE2#QTABLE (Q2#QTABLE in the figure) for the original name QTABLE.

The second point illustrated by this example is that amount of translated code increases rapidly when several instantiations are invoked from a single class definition. Not all of this increase can be avoided. For example, the procedures Q1#ENTER and Q2#ENTER, though similar at the surface, deal with distinct types of data elements (INTEGER and REAL). In some cases (e.g., Q1#EMPTY and Q2#EMPTY), the procedures are invoked from a single class definition. Not all of this increase can be avoided. For example, the procedures

```pascal
begin
    (* other entries of ISAMFILE class go here *)
end;
```

variable F#KEYS and F#RECS to be added to DRIVER block. But F#KEYS is itself a class type (ASSOCMEMORY) variable. Thus, F#KEYS generates F#KEYS#ASSOC, F#KEYS#EMPTY, and F#KEYS#DELETED in its place. This kind of secondary instantiation is not handled any differently in our preprocessor than the single level case. Conceptually, the outermost class variables are instantiated first, giving rise to an intermediate program. This program is then processed as before to eliminate class variables from the new variables, and so on. This conceptual view will help explain the double transformation of certain original names in the translation, e.g., LOCATION to ISAMFILE#ASSOCMEMORY#LOCATION (abbreviated as IS#AS#LOCATION). In practice, the secondary instantiations can be achieved without a second pass over the text. This is done by processing all declarations (variables and subprograms) stemming from the instantiation of a class at the time that they are being added to the host block as if they were originally part of the host block.

CONCLUSION

We have presented here a technique for implementing data abstraction in Pascal using a limited form of the class structure.
provided in Concurrent Pascal. The class structures are translated to equivalent (sequential) Pascal constructs using a preprocessor whose design is also presented. The abstract data types supported by this source-to-source translation scheme are somewhat limited, chiefly because of the limitations of Pascal and a desire on our part to keep the translation simple and straightforward. Abstract data items may not be passed as parameters to subprograms, because this would have involved passing a variety of resources—data structures, procedures, and functions—as parameters in the translation, not all of which are supported in common implementations of Pascal. We have limited abstract data items from being part of a larger data structure, such as an array or a record, in order to simplify the task of translation as well as maintain a closer correspondence between the original and the translated programs. Even with these limitations, the scheme presented will permit a Pascal programmer to define and use sophisticated abstract data types, such as the index file type illustrated here, with only a small cost for preprocessing.

REFERENCES