Levels of Abstraction in CS2

James P. Kelsh
Department of Computer Science
Central Michigan University
Mt. Pleasant, MI 48859

The principle of an abstract data type accustoms us to thinking of a program at two levels: The application program and the underlying implementation of the data types involved. Convincing students to maintain this distinction can be difficult: The temptation to use details of the chosen implementation often overcomes their good intentions. One solution is to introduce a "second degree" of data abstraction, separating the "core" operations (which must use the details of an abstract data type's implementation) from higher-level operations on the abstract data type and then freely exchange data type implementations. This can also prepare students for later exercises which will require more levels of abstraction.

1.Data Abstraction and Strings.

We expect our CS2 course to teach students to use abstract data types [2]. However, many students learn more about coding one implementation of each data type than about abstraction. (See Kumar & Beidler[3] for an earlier discussion of this problem.) McCracken [4] provides an excellent example, showing the reader an implementation and encouraging the student to program at a higher level (ignoring the details of the implementation). However, when my students used these routines for character strings, they often found it easier to manipulate the characters in relatively high-level procedures than to learn how to use the supplied string procedures. Even when the multiple implementations provided by Dale & Lilly [1] were available, many students would ask "Which implementation do you want us to use?" and then use the details of that implementation rather than learn the calling syntax of the routines. They learned to code data types, but missed the abstraction we want them to learn.

To discourage this approach, I produced three different string implementations and an interface separating them from the programmer. (A fourth has recently been added: See section 4: Next Developments.) A program that works correctly with all three must not be using special features of the implementation. If the currently available implementation changes from day to day, programs are forced to be implementation-independent. The three implementations are:

The "standard" implementation: (More authors than I could possibly acknowledge have written such implementations.)

```
ChRay = packed array [1..MaxStrLength] of
char;
StrType = record
  Str    : ChRay;
  Length : integer
end;
```

MaxStrLength is a programmer-specified constant. Str holds the characters and Length specifies how many of the available characters are in use.

The Turbo Pascal implementation:

```
StrType = packedarray[0..MaxStrLength] of
char;
```

This emulates the type String[N] provided by the Turbo Pascal compiler. N specifies the maximum number of characters the string may hold. The compiler implements String[N] as a packed array [0..N] of char. Positions 1..N hold the characters; position 0 holds a character whose ordinal value is the length of the string (the analog of the "Length" field above).

The C implementation:

```
StrType = packed array [0..MaxStrLength] of
char;
```

This emulates the way the C language handles strings. It resembles the Turbo Pascal implementation, but only positions 0..(N-1) hold the characters of the string, where N is the number of characters in the string. The final character is followed by chr(0), a terminator in C.

2.Strings: Independent in the Second Degree

Choosing three implementations triples the task of writing the code. Believing strongly in reusing software components, I soon extracted a core of string routines that were intimately involved with the details of the string implementation:

procedure **MakeStringEmpty** (var S:StrType);
Sets its parameter to the empty ("null") string.

procedure **AppendChar** (var S: StrType; C: char);
Appends the character C to the end of the string S if the result would not exceed MaxStrLength. In that (error) case, AppendChar takes no action.

function **StrLen** (S: StrType): integer;
Returns the current length of the string S.

function **NthCh** (N: integer; S: StrType): char;
Returns the character in position N of string S. (If N exceeds StrLen(S), NthCh returns a blank.)

These core routines constitute an intermediate level of abstraction: All the other routines are defined in terms of them. (The fourth implementation -- described in Section 4 -- required adding a fifth core routine.) For example, to concatenate a "source" string to the end of a "target" string:

```
procedure ConcatStr (var Target: StrType;
Source: StrType);
var   I : integer;
begin
  for I := 1 to StrLen(Source) do
    AppendChar(Target, NthCh(I, Source))
end;
```

Note that any anomalous conditions are handled by AppendChar and NthCh.

The string package includes, among others:

```
function  StringsAreEqual (S1, S2:
StrType): boolean;
function  CompareStr (S1, S2: StrType):
char;
procedure ReadLnStr (var FileV: text; var
S: StrType);
procedure WriteStr (var FileV: text; S:
StrType);
procedure ReverseStr (var S: StrType);
procedure IntToString (I: integer; var S:
StrType);
procedure IntVal (S: StrType; var Int,
Err: integer);
procedure RealVal (S: StrType; var
RealNum: real;  var Err: integer);
procedure Shorten (var S: Strtype;,
HowMuch: integer);
procedure UpperCaseString (var S:
StrType);
```

This intermediate level distinguishes this approach from that of [1]: Although that text provides multiple implementations of each abstract data type, the book develops programs at only two levels of abstraction: The program and the implementation.

To change from one string implementation to another requires merely changing the type definitions and exchanging the first 4 "core" subprograms. Students react favorably to this, possibly because the string routines themselves illustrate how to produce implementation-independent code. Although these "second-degree" independent operations have been useful in several programs, their main purpose is still to illustrate how to use an abstract data type without using the details of its implementations.

3.Results for Student and Instructor.

I've released the "second-degree" abstract string routines to four classes (CS2, Compiler Construction, Documentation, and Software Engineering). CS2 students were required to use them, while the students in the more advanced courses were merely given the option. In many cases, the more advanced students had more problems adjusting: Although many were moderately familiar with string extensions in the Pascal they had used, few knew the details of the implementation well enough to avoid surprises. Like many other language extensions, string extensions can surprise students who use code like:

```
S := 'ABC';
T := S;          {Both have Length = 3}
S[4] := 'X';
if S = T then  {Does the 4th char matter?}
  writeln('Yes')
else
  writeln('No');
```

Students who encounter surprises with the "second-degree" routines can read the code to find out what happened. But at the same time, the fact that the routines are usually "included" rather than being part of the source file seems to encourage students to use them as "black boxes" while designing their programs.

Students writing a compiler felt challenged by the programming assignment and were happy to use the string routines. Those who took their programming projects lightly were more likely to feel that learning to use the "second-degree" routines would take too long. One devised

his own pool of "string space" complete with garbage collection. None of the students who chose to develop their own string routines completed a working program by the due date.

4.Next Developments in strings.

Some users of these string implementations are troubled by the "MaxStrLength" limitation: This constant must be known at compile-time and every string will use that much space. This is the standard questions of static vs. dynamic storage allocation. It led to a fourth implementation: StrType.Inf.

In this case, StrType is a pointer to a "block" which resembles the earlier StrType.Std. A block can contain up to "CharsPerBlock" (a new programmer-defined constant) characters and possibly a pointer to another block. Thus, the length of a string is limited only by available memory and short strings can be as short as one block.

Although a working implementation is possible with only the four core routines described in section 2, efficient use of dynamic data structures usually requires some form on initialization. The fifth "core" routine is InitStr (S).

InitStr guarantees that storage is available for S. Every program should call InitStr before using a string. Although the Std, TP, and C versions of InitStr do nothing, the Inf version uses NEW to allocate space for the first block, sets the length field to 0, and sets the first block's "Next" pointer to nil. Separating core operations from higher-level operations allowed creating a reasonably full set of operations on a dynamic data structure with surprisingly little coding.

Certainly other data structures are amenable to this level of separation. I recently shared a layered toolkit for character-based windowing (using these string routines) with a software engineering class (They surprised themselves with the quality of their user interfaces!) and am now re-implementing its "core" to port it to two other compilers running on different operating systems. Maybe "second-degree" data abstraction will help the cause of software portability, too.

I can supply a copy of the "second degree" string routines and some sample programs to anyone who sends one formatted MS-DOS diskette in a self-addressed stamped mailer.

References:

1. Dale, and Lilly. *Pascal Plus Data Structures, Third Ed.* D. C. Heath & Company, Lexington, MA, 1991.

2. Koffman, Eliot B., Stemple, David, and Wardle, Caroline E. "Recommended Curriculum for CS2, 1984," *Communications of the ACM*, 28, 8 (Aug. 1985), pp. 815-818.

3. Kumar, Ashok, and Beidler, John. "Using Generics Modules to Enhance the CS2 Course," *Proceedings of the Twentieth* SIGCSE Technical Symposium on Computer Science Education (Feb. 1989), pp. 61-65).

4. McCracken, Daniel D. *A Second Course in Computer Science With Pascal*. John Wiley & Sons, New York, 1987.

5. Collins, William, and McMillan, Thomas. "Implementing Abstract Data Types in Turbo Pascal," *Proceedings of the Twenty-First SIGCSE Technical Symposium on Computer Science Education* (Feb. 1990), pp. 134-138.

****

TQ REFERENCES-- continued from page 34

Tripp, Leonard L. "Software Engineering Standards: Today and Tommorrow, Part 2. ", Software Quality, September, 1992.

Walton, Mary. The Deming Management Method. New York: The Putnam Publishing Group, 1986.

Training

American Society for Quality Control, P.O. Box 3005, Milwaukee, WI 53201-3005

Specific courses in TQM, Quality Control, Software Quality Assurance.

George Washington University, Continuing Engineering Education Program, 801 22n Street NW, Washington, DC 20052

Offers a wide variety of quality oriented courses.

Juran Institute, Inc. 11 River Road, Wilton, CT.

Special training in quality management as related to software engineering, as well as, courses in quality control and management.

Quality Assurance Institute, Suite 350, 7575 Dr. Phillips Blvd., Orlando, FL. 32819

Provides quality training in a wide variety of information system and software engineering fields. Specific courses offered in testing and measurement.