Instructional Frameworks: Toolkits and Abstractions In Introductory Computer Science

Cynthia Brown, Harriet Fell, Viera Proulx, Richard Rasala
College of Computer Science, Northeastern University, Boston MA 02115

1. Introduction

Computer science education has been changing over the past few years. The Denning Report [4] and the ACM-IEEE Curriculum 91 [6] helped trigger a number of new initiatives aimed at improving computer science education, especially at the introductory level. One proposal is the use of closed laboratories to improve programming instruction. A second idea is to add breadth to the introductory curriculum. A third suggestion is to introduce more formal instruction in theoretical computer science.

These proposals have merit but their implementation is problematic. Often the projects suggested for closed laboratories are too simple and uninteresting. The breadth component is frequently poorly integrated with the programming component. The theoretical material is often presented before students have sufficient practical and scientific experience to follow what is presented and understand its significance. Furthermore, software engineering is preached rather than practiced since neither demonstration programs nor student projects are large enough for genuine software engineering methods to be illustrated.

At Northeastern, we have developed a teaching paradigm which integrates a number of ideas in current science curriculum reform with some approaches that are unique to our institution. Our teaching emphasizes visualization and interaction both in animated demonstrations that we provide and in laboratories and assignments that we ask students to complete. We believe that software design and development is an incremental process so we provide students with substantial bodies of code to read, expand, and modify. In effect, our model is an apprentice based approach in which students make meaningful contributions to interesting software products but are not required to program every detail. Theoretical concepts are taught in the context of practical algorithm and data structure design problems. Software engineering is emphasized throughout as the combination of theoretical ideas, design techniques such as abstraction and the use of tools, and technical knowledge such as programming languages and system expertise.

Partial support for this work was provided by NSF grants USE-9152211 and USE-9155929.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

SIGCSE'97.
Copyright 1997 ACM 1-58113-499-1/97/0006…$5.00.

In recent papers [2,3], we described the use of visualization in our curriculum and the experimental approach to problem solving we expect of our students. In this article, we discuss the software infrastructure which makes this approach possible. We present selected examples of basic toolkits and more sophisticated abstractions which allow students to create high quality projects with robust code. We also explain how these modules can be used to illustrate important theoretical and software design principles.

2. Basic Toolkits for Freshman Programming

A fundamental strategy of software engineering is the use of separately defined and compiled software toolkits. In the introductory computer science curriculum at Northeastern, toolkits are an essential component of the instructional framework. THINK PASCAL on the Macintosh makes the use of toolkits very easy since the process of compiling and linking the separate files in a project is entirely automated. In this article, we will discuss only three of several toolkits: SimpleWindows, StringTools, and IOTools. These toolkits are the ones most frequently used by the students.

SimpleWindows

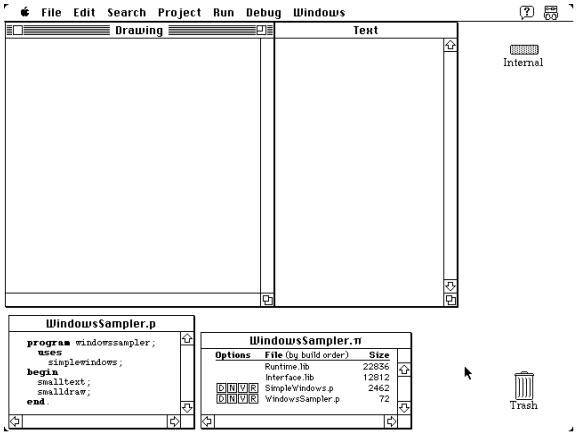


Figure 1: Typical Placement of the Text and Drawing Windows Using the SimpleWindows Toolkit

The SimpleWindows toolkit is used in the first few weeks of class in the freshman computer science courses. SimpleWindows permits the student to specify the position and size of the text and drawing windows in such a way that the windows utilize screen space efficiently. Since

75% of the Macintosh computers in our open laboratory are small screen Macintosh SE's, utilization of screen real estate is especially important. In Figure 1, the tiling obtained by the pair of calls `smalltext` and `smalldraw` is shown. SimpleWindows provides three other tiling pairs which share space in various ways. Some tiling pairs leave significant vacant space so that debugging windows can be open and visible as the program is executing. Each window is put into place by calling the appropriate procedure in SimpleWindows. Thus, if the student wishes to reduce the window sizes during debugging, she simply needs to replace the calls `smalltext` and `smalldraw` with the calls `minitext` and `minidraw`.

The use of SimpleWindows early in the freshman course emphasizes to students the importance of *software toolkits which are encapsulated in separate files*. SimpleWindows also provides a significant example of the use of procedures without parameters.

StringTools

THINK PASCAL comes with a good collection of routines for string manipulation. The goal of the StringTools package is to provide a few additional highly useful tools. The first set of tools test if a character is a letter, is uppercase, is lowercase, or is a digit and handle conversion of characters between uppercase and lowercase. We will not discuss these further.

The next set of tools concern string standardization. These tools deal with a major annoyance in student programs which involve strings. To explain the difficulty, consider a program with a data file which contains a list of cities such as `Boston, New York, Los Angeles, …` together with associated data. A typical problem might ask the student to read in the file and then interactively type the name of a city to retrieve the data associated with that city. Invariably, many students will type `new york` instead of `New York` and be unable to match the name of the city in the data base properly. The string standardization tools address this difficulty.

StringTools provides three utilities `upcasestring`, `locasestring`, and `standardize` which will put a string into a standard form either all uppercase, all lowercase, or mixed case with leading capital letters. For example, `standardize('nEW yoRk')` returns `New York`. Students can use these tools in one of two ways. One way is to homogenize input as it comes in so that all strings can be guaranteed to have a desired standard form. The other way is to leave input alone but make comparisons of the form: `if standardize(s) = standardize(t) then ….`

The last set of tools in StringTools provides string splitting. In classical PASCAL, if a line of text contains a mixture of data items, one must read the line character by character to parse the line into the separate items. This is very tedious to program. A much better way is to read the entire line into a string and then split off appropriate substrings corresponding to the various items. StringTools provides two tools to split a string, one which looks for a particular character to define the break point and the other which looks for a character belonging to set of separator characters.

StringTools is a frankly utilitarian package which sends an important software engineering message to students. *A programming language may be awkward to use in the form provided by its developer. By building relatively simple tools, the ease of use of the language can be significantly increased*. This message is reinforced by the IOTools package.

IOTools

Examination of current introductory computer science textbooks shows that a substantial fraction of the source code presented as examples does not focus on the issue being taught at the moment but on sequences of prompts, `readln`'s, and `writeln`'s. This explicit in-line code for input-output is boring and distracts students from the real issues. Moreover, such code teaches very poor software engineering habits. There is *no conceptual organization* applied to the input-output code and, furthermore, *robust error checking is impossible* since an invalid input to a numeric variable will crash the program before the `read` call ever returns.

The IOTools package provides a well-engineered set of input routines. The features include:

- robust input for strings of fixed or arbitrary length
- robust input for other data types, especially, numeric
- built-in prompt strings and optional default values
- utilities to force numeric values within range
- input from internal strings as well as the keyboard

There are two functions for string input:

```
function request_string (prompt, default:
  string): string;
function limited_string (prompt, default:
  string; limit: integer): string;
```

The function `request_string` is the central input routine called by all other input routines. This function displays the `prompt` string if it is non-empty, displays the `default` reply string within brackets if it is non-empty, and then reads the user input line. After reading, the line is squashed to remove leading and trailing blanks. If the squashed line is non-empty then it is returned as the function value otherwise the `default` string is squashed and returned. The function `request_string` traps numerous errors, politely signals these with the message IO Error, and then permits the user to re-enter the line of text. There is only one fault that this function cannot trap. If the user presses the Enter key then this is interpreted as end-of-file on the input stream and there is no clean way to recover. The idea that an interactive device can ever signal end-of-file is an obsolete holdover from the days when batch programs were run without modification on video terminals.

`Limited_string` is similar to `request_string` except that it guarantees by truncation that its return value will have length at most the `limit` parameter. This function is necessary since a program will crash if a line is read which is longer than the string provided to receive it.

For each of the other basic types, IOTools provides a pair of input routines designed somewhat differently. The routines for type `integer` are typical:

```
function read_integer(prompt: string):
  integer;
function request_integer(prompt: string;
  default: integer): integer;
```

The `read_integer` routine insists on a non-empty reply and will reinitiate the read if the user hits Return on an empty line. In contrast, `request_integer` will return the `default` in that case. Both routines add to the standard error checking by catching numeric errors in the input line and permitting the user to re-enter the input if needed.

There are several utility routines which request single character responses from a user. These routines automatically convert the response to uppercase so that it is easy to feed the result to an **`if`** or **`case`** statement. In addition, there is an interesting function called `confirm`:

```
function confirm(prompt: string; default:
  boolean): boolean
```

Function `confirm` displays its `prompt` and expects `Y` or `N` as the response (where `Y` = `Yes` corresponds to `True` and `N` = `No` to `False`). The function is useful for asking questions whose answer may control decisions and loop termination.

From this brief description of IOTools, it is clear that *input-output programming is both more robust and more compact when these tools are used*. In contrast, although many textbooks give lip service to robust input-output, none really do much about it. The reason is that *it is out of the question to carry out a high level of error checking if such checks have to be programmed manually for every single IO operation*. It is only by encapsulating the error checks in an organized and complete toolkit that the programmer is empowered to access and use them on a regular basis. This fact is a vital software engineering lesson for the freshman computer science student.

3. Sophisticated Abstractions in the Freshman Curriculum

Every scientist knows that abstraction is essential for organized, efficient thinking. To a naive freshman, however, it may appear that abstraction is an unnecessary complication which hides the concrete issues behind a dark veil. To show a freshman that abstraction has substantial positive benefits, examples of abstraction must be presented and *utilized* which provide compelling evidence that abstraction is not merely useful but that it is in fact the *only way* to deal with complexity in computer science, mathematics, and the other sciences. In particular, the concept of *levels of abstraction* must be taught so that the student learns that the concrete issues are not hidden forever but simply organized into various layers which can be handled more effectively one by one.

The current computer science textbooks for freshmen praise abstraction highly but the examples given of abstraction are bland and uninspiring. The student is left with the feeling that abstraction accomplishes little except to make the programming process longer. We believe that *the student must experience sophisticated instances of abstraction which demonstrate that an abstract approach to thinking and designing is vital*. In this section, we will

describe several abstractions we introduce at Northeastern and explain what principles each helps to elucidate.

Loops, Decisions, and the Swimming Fish Lab

The Swimming Fish laboratory exercise is designed to require students to program a loop with decisions in which the progress of the loop cannot be predicted prior to runtime. The situation of the exercise is a large underwater maze-like cave in which a large fish searches for food consisting of a school of small fish { see Figure 2 }.

Figure 2: Typical Initial State of the Swimming Fish Laboratory

The large fish is initially positioned at the left side of the cave and the school of small fish at the right. The cave is randomly generated but is designed so that the large fish can find the food using only moves up or down or to the right. The large fish never needs to backtrack to the left.

The Swimming Fish laboratory is introduced to the students about seven weeks into the first course *before* array data structures have been discussed. The students are able to solve the exercise because the critical tools are presented as abstractions. The solution is based on a shell program which the students must complete, on four of the basic tools modules, and on a file which contains the picture resources for the large fish and the school of fish. The four key abstractions which the students use to program the search of the large fish for the food are:

```
type directions = (up, down, right);
function freetomove (d: directions):
  boolean;
procedure movefish (d: directions);
function foodfound: boolean;
```

The type `directions` abstracts the three directions in which the large fish may need to move. Function `freetomove` tests whether movement in a particular direction is possible, that is, is there open water rather than cave walls in that direction. Procedure `movefish` will move the large fish in the desired direction. Finally, function `foundfood` tests whether the large fish has landed on the cell containing the school of small fish.

The students must program their solution to the exercise entirely in terms of these abstractions. They are not permitted to peek at the underlying array data structure for the cave. After some thought, they realize that `foundfood` can be used to control the termination of a **`while`** loop and that `freetomove` is the critical tool needed for deciding where to move the fish next. Of course, they must plan the order in which various directions are tested and maintain state information to prevent an indefinite oscillation of the fish up and down.

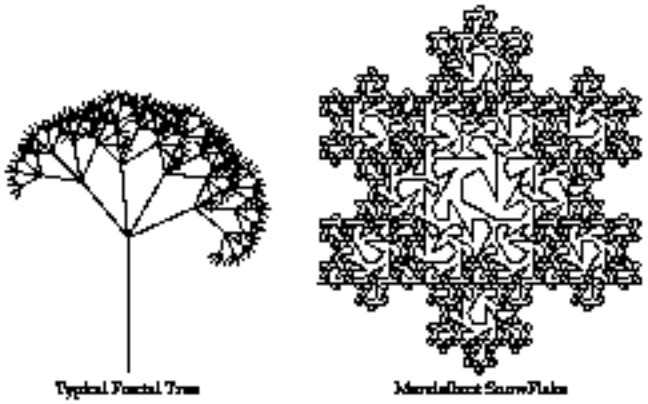
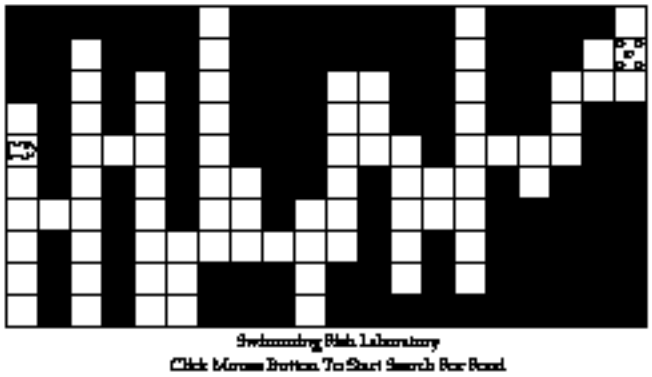


Figure 3: Typical Results of the Recursive Fractals Assignment

In the Swimming Fish laboratory handout, we are open with the students that the use of abstraction is a key lesson of the assignment. In a section entitled "Educational Goals and Additional Comments", we explain to the students:



> "An interesting aspect of this exercise is that you obtain a global solution (food is found) simply using local information (what directions are open to the fish at the current position). In computing, it is pleasing when you can find an efficient global solution using only local information.
>
> The fact that local information is sufficient for the solution makes it easy to set up the abstractions freetomove, foundfood, and movefish which help to hide the internal data structures and thereby permit a clean program design."

The shell program hides a great deal of information in addition to the abstractions directly used in solving the problem. The cave must be randomly initialized in such a way that there is always a path from the large fish to the food and such that backtracking is not required. Also, the cave cells, the fish, and the food must be drawn. *All of these nuggets of code form a rich domain for exploration by the better students but the design of the laboratory permits the weaker students to simply work on the main problem without distractions.*

Recursion, Fractals, and the Turtle Abstraction

Fractal curves are one of the most beautiful illustrations of recursive definitions and we therefore use fractals to help explain recursion. Students see live demonstrations of the Koch snowflake, the Hilbert curve, and variations of the dragon curves. The homework assignment asks the students to program a tree fractal and a Mandelbrot snowflake as shown in Figure 3.

To permit all of the fractals to be treated in a similar manner, we introduce a LOGO-style turtle abstraction to assist in tracing out these curves [1]. The design issues of the turtle abstraction offer a great opportunity to discuss

with freshman the concept of an abstract data type. The idea behind the turtle abstraction is geometrically simple yet the implementation of the data type and its operations requires some sophistication.

The first issue is: How should the turtle store its numerical data? Since the underlying coordinates of the computer screen are integers, it is tempting to use integers to store the current state of the turtle. This is a major mistake when drawing complex fractals. After drawing hundreds of small line segments using only integer data, the accumulated round off error can make the computed position of the turtle quite different than what its true position should be. Therefore, the proper decision is to store turtle data as reals.

The next question is: How should angles be measured? Degrees are used in the real world but mathematicians and programming languages tend to be partial to radians. Which measure is better? For fractal problems, degree measure is superior since the angles which commonly occur are easily expressed: 90˚, 60˚, 45˚, 30˚. Radian measure is really most useful in calculus situations because the formulas for derivatives of trigonometric functions work out elegantly in radian measure. Radians can be a nuisance however in computer graphics.

The next question is: What are the appropriate turtle operations? The LOGO turtle can move forward in its current direction by its current step size and, by lifting the pen, can jump forward as well. The turtle can also change its step size and its current direction. This leads to six basic operations:

```
procedure forward     (var tt: turtle);
procedure jumpforward (var tt: turtle);
procedure changestep  (var tt: turtle;
  factor: real);
procedure right  (var tt: turtle);
procedure left   (var tt: turtle);
procedure rotate (var tt: turtle;
  degrees: real);
```

The remaining operations to round out the turtle abstract data type are initializations which set the initial turtle position, step size, direction, and default turning angle. It is also convenient to have a utility procedure `getsincos` which returns the sine and cosine of an angle given in degrees.

There is a subtle implementation issue in the turtle abstract data type which illuminates critical questions about recursion and the use of **`var`** versus non-**`var`** parameters in PASCAL. In using a recursive procedure `tree` to compute the tree fractal, you would like the `tree` procedure to forget what it has done when drawing a sub-tree and restore the prior state of the turtle when it returns to each node. This suggests non-**`var`** semantics for the recursive procedure:

```
procedure tree(tt: turtle; … )
```

In contrast, in using a recursive procedure `mandelbrot` to draw the Mandelbrot snowflake, you would like the `mandelbrot` procedure to retain the turtle state since each link of the snowflake attaches to the previous link. This suggests **`var`** semantics for the recursive procedure:

```
procedure mandelbrot(var tt: turtle; … )
```

The subtle implementation issue is that a naive approach to programming procedure `forward` can cause the `tree` procedure to fail even with the proper non-**`var`** turtle parameter. Why is this so?

In Apple Macintosh graphics, the current location of the drawing pen is maintained as a system parameter by the QuickDraw toolbox. If the implementation of procedure `forward` relies on this fact then the turtle will always draw from the last position visited independent of the semantics used in the procedure calls. For the abstraction to work properly, the turtle must always ensure that it is drawing from the correct starting point. This is a general principle. *An abstract data type must ensure its implicit preconditions and not rely on system parameters.*

The preceding discussion shows that the introduction of the abstract turtle data type permits a wide-ranging exploration of recursion, round off error, procedure call semantics, and the proper way to implement abstract data types in general. Students obtain a better handle on each of these topics through understanding their interconnections.

The Game of Life and the Fat Bits Abstraction

The Game of Life is a simulation of cellular life invented by John Conway in the late 1960s and introduced to the world in Martin Gardner's Mathematical Games column in the Scientific American. See Gardner [5] for an updated account of Life research. Implementing the Game of Life is an fascinating assignment for students which exercises both algorithmic concepts and 2-dimensional array data structures. In order for this assignment to be effective, however, three conditions must be met:

- the life simulation must be animated on the computer screen in real time;
- the student must be able to create life forms interactively;
- the student must be able to save interesting life forms on a file and then read them back at a later time for further examination.

We have seen textbooks which suggest that the Game of Life can be programmed with the output streamed in teletype mode to a text window but such a method makes exploration and understanding of the Game of Life almost impossible. The output must be graphical and the student must be able to interact with it.

At Northeastern, we have created a graphical shell program which permits the student to program the algorithmic portions of the Game of Life and then receive the interactive support almost for free. The Game of Life program is controlled entirely with the mouse using action buttons, check boxes, and a suite of radio buttons. In Figure 4, we show a typical screen from the Game of Life. A particular life form with a period 3 oscillation has been entered as an illustration. New life forms can be created by interactively editing the individual cells using a "fatbits" method modeled after MacPaint.

In the version of the Game of Life given to the students, the "Start" button activates a stub procedure which is empty and does nothing. Also, the "Create Random Data" button is missing and must be created and made active. The student is expected to make the "Start" button operate by implementing the transformations specified by Conway in the rules for the Game of Life. The student is expected to create the new button by exploring the existing code and then imitating what has been done. *We believe that imitation is a very effective form of learning both for toddlers learning to speak and for college students learning to program.*

Although the Game of Life assignment has many aspects which deserve discussion, we want to focus on the `fatbits` abstract data type which is provided to the students and which forms one of the most sophisticated examples they will encounter in the freshman year. The data structure for the `fatbits` abstraction is:

```
fatbits = record
  data: packed array[0..63,0..63] of 0..1;
  rows: integer;
  cols: integer;
  cellsize: integer;
  blocksize: integer;
  drawgrid: boolean;
  bounds: rect;
end;
```

The `fatbits` type includes a `data` array with the bitmap information and a sequence of other parameters which help make it easy to work with the data type. This design illustrates a general principle: *Include all relevant data of an abstract data type in a single encompassing data structure.*

The most interesting operation in the `fatbits` abstraction is the procedure `fatbits_edit` which handles interactive editing of a `fatbits` structure. The students are surprised at how easily this procedure can be encoded. A careful examination of the code shows them that the simplicity results from using a well designed set of auxiliary routines and from taking good advantage of the main event loop. Thus, `fatbits_edit` illustrates how *a complex task can be made easy by an appropriate division of labor.*

All of the essential information about a particular life form is contained in the `fatbits` data structure. Therefore, to save such data to a file in PASCAL, it suffices to create a binary file of record type `fatbits` to which the entire `fatbits` data structure may be sent with one `write` call. This technique is quite general. *As long as the state of a program can be encapsulated in a single static data structure, this state can be saved to a binary file which will contain exactly that one data item.* To our knowledge, no elementary textbook discusses this method.

The Game of Life program introduces upon some fairly advanced programming and software engineering techniques. The material is manageable for the students because what they have to work on themselves is carefully confined to particular aspects of the whole program. *They are therefore like apprentices in a workshop who get to observe an entire project but who are expected to actively work on only appropriate components of the project.*

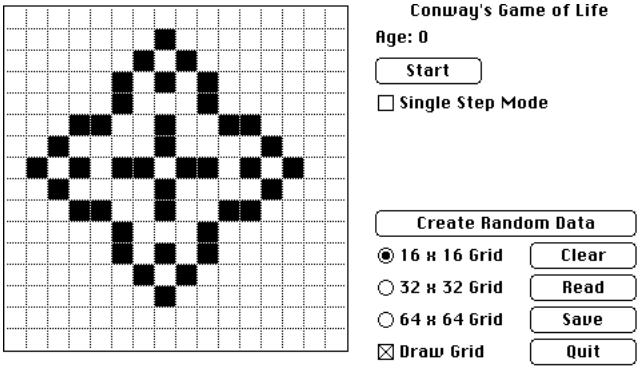


Figure 4: Sample Screen from the Game of Life

File Abstractions

The PASCAL concept of file is too restricted. PASCAL offers two kinds of files: text and binary. Text files mimic the line-by-line input-output of a keyboard and a screen and provide the same facilities with the same limitations as in interactive text IO. Binary files offer a more direct bit-to-bit method of data storage: what is in memory may be sent "as is" onto the corresponding locations on disk. Unfortunately, when Niklaus Wirth designed PASCAL [7], he viewed sequences as the paradigm for file data and so decreed that a binary file must be a *sequence of data items of the same type*. Although one can sometimes use such binary files to advantage, as in the `fatbits` example above, in general the restriction to items of a single type cripples the use of PASCAL binary files.

A binary file should be a free format object to which one can send any data of any size at any time. The program creating the file should be responsible for knowing how the data was sent to the file and should add extra information to the file when necessary to clarify how much data of a particular type has been sent and how it is arranged.

Is it possible to demonstrate this more general concept of file to students learning PASCAL? The answer is yes and the secret is abstraction! Why should the built-in concepts of file be taken as binding? To create a new notion of file, we need only create a file abstraction which is implemented on top of the lower level file tools provided directly by the Macintosh operating system. This abstraction is presented and used as a genuine "black box" since we do not encourage freshmen to get into the tangled lore of the File Manager chapters within Inside Macintosh.

The file tools are actually built on two abstractions, `fileinfo` for files and `folderinfo` for folders. Each uses the absolutely minimal amount of information needed to interface with the Macintosh File Manager. The file tools package then provides the following kinds of tools:

- file selection via standard dialog boxes
- file access tools: create, open, close
- file input-output: readfile, writefile
- file traversal tools: file size and current position
- file information tools: names, paths, parents
- an error message tool

In the file access tools, a delete-file tool is intentionally omitted since the potential for student error is great. We

will discuss in detail only the input-output tools. The interfaces are:

```
procedure readfile (var info: fileinfo;
  address: ptr; count: longint);
procedure writefile (var info: fileinfo;
  address: ptr; count: longint);
```

Here `address` is a pointer or memory address and `count` is the number of bytes to read or write. Typically, `address` is replaced by `@variable` where `@` is the "address of" operator built into THINK PASCAL and `count` is replaced by `sizeof(variable)` where `sizeof` is a built-in compile-time function. The definitions of `readfile` and `writefile` therefore guarantee that any data of any size may be read or written.

The fundamental message of the file tools package is that a programmer need not be limited by the deficiencies of the built-in tools. *Abstraction can be the key to the development of tools more suited to the task at hand and the way to hide enormous amounts of technical detail.*

4. Conclusions

In this article, we have described software tools and abstractions of much greater interest and depth than those normally introduced in the freshman curriculum. We have indicated how these tools can teach students genuine software engineering principles and enable them to build interesting and creative products. We believe that the use of sophisticated tools together with a radical change in the kind of software that students are asked to create helps make the freshman curriculum substantially more exciting and intellectually rewarding. Our curriculum integrates theory, design, and technique and it provides a model that other colleges and universities may desire to emulate.

Bibliography

[1] H. Abelson & A. diSessa, *Turtle Geometry*, MIT Press, Cambridge, MA, 1980.

[2] C. Brown, H. Fell, V. Proulx, & R. Rasala, *Programming by Experimentation and Example*, in *Computer Assisted Learning (ICCAL 1992)*, I. Tomak (ed.), Springer-Verlag, Berlin, 1992, pp. 136-147.

[3] C. Brown, H. Fell, V. Proulx, & R. Rasala, *Using Visual Feedback and Model Programs in Introductory Computer Science*, J. Comp. Higher Ed., 4(1), Fall 1992, pp. 3-26.

[4] P. Denning, D. Comer, D. Gries, M. Mulder, A. Tucker, A. J. Turner, & P. Young, *Computing as a Discipline*, Comm. ACM, 32(1), January 1989, pp. 9-23.

[5] M. Gardner, *Wheels, Life, and Other Mathematical Amusements*, W. H. Freeman, New York, 1983, Chapters 20-22.

[6] A. Tucker, et. al. (ed.), *Computing Curricula 1991*, Report of the ACM/IEEE-CS Joint Curriculum Task Force, ACM Press, New York, 1991.

[7] N. Wirth, *Algorithms + Data Structures = Programs*, Prentice-Hall, Englewood Cliffs, NJ, 1976.