The Pascal Programming Language
Bill Catambay, Pascal Developer

Chapter 4

The Pascal Programming Language
by Bill Catambay

Return to Table of Contents

IV. Myths Uncovered

The fast-moving pace of technology and the variety of platforms different compilers support makes it easy for multiple dialects of the same programming language to evolve. Although standards exist, compiler vendors have not chosen to comply fully with them. The result is a variety of Pascal dialects with significant differences in performance.

Most Pascal compilers on the market support the unextended Pascal standard. Most support an expanded subset of the Extended Pascal standard. Programmers often pass judgment on the Pascal language based on their experience with one particular Pascal dialect, without knowing whether that dialect complies with the standards or how it compares to other available implementations. In several of the dialogs I've been involved in over the past ten years on the USENET newsgroups, negative views of Pascal have been based upon pre-conceptions about the language or bad experiences with a specific compiler. In most cases, these notions were based upon myth rather than fact.

Myth 1: C and Pascal Are Basically the Same Language

This comparison has been made quite frequently over the years, yet nothing is further from the truth. From the compiler's perspective, the Pascal architecture is much more straightforward. It adheres to stronger type definitions, making optimizations easier to accomplish. From the programmer's perspective, the nature of Pascal is inherently different from C in convention, syntax, structure and mindset.

I have converted several C source code projects, and even a few C++ projects, to Pascal and Object Pascal, and I've seen a variety of coding styles. The process of converting a C program to a Pascal program provides first hand experience in the differences between the two languages. In Pascal, declarations must be moved to the top of a block, and often a lot of investigation is required to decipher many of the C data structures. During the conversion process, I reorganize the code, changing and indenting it to make it more readable, so that the actual translation process is much easier. It's a significant amount of work, however, and in some cases the difficulty was so extreme that I canceled the task.

The point is that while C code can be made readable, it provides loopholes that give programmers the ability to create chaos. Given the weak typing of the C language, inconsistency of data use becomes not only a possibility, but a probability. Pascal's strong typing makes inconsistency of data use far less likely to occur.

Another difference between C and Pascal is the underlying design of the language, and the intent of that design. C shares APL's penchant for being able to cram a lot of action into a single expression or line-of-code. This provides a certain "cool" factor that many programmers appreciate, but it also leads to a bloated and unreadable code base which can be difficult, at best, to maintain. The extensive operator overloading and weak-typing architecture increases the odds that a programmer will get caught up in the "passion" of the moment, indulge every shortcut, and end up with a maintenance nightmare.

Cultures have sprung up around both languages; cultures reflecting different attitudes toward getting work done. The culture of Pascal is oriented unapologetically toward readability in style, elegance of algorithm, and its expression as code. The culture of C is self-consciously ambitious, yet obfuscatory. This is illustrated quite well by C. A. R. Hoare in an introduction to the classic paper, *An Axiomatic Definition of the Programming Language Pascal*, for the book, Great Papers in Computer Science. In reference to his goal of designing a better programming language - one which makes it easier to write correct programs and harder to write incorrect ones - Hoare writes, "It is a matter of continuing regret that so few languages have ever been designed to meet that goal, or even to make significant concessions towards it. For example, the programming language C was designed to assist in writing a small single-user operating system (UNIX) for a real-time minicomputer (PDP 11), now thankfully obsolete. For this purpose, its low level of abstraction and plethora of machine-oriented features are entirely appropriate. For all other purposes, they are a nuisance. The successful propagation of the language can be explained by accidental, commercial, historical, and political factors; it is hardly due to any inherent quality as a tool for the reliable creation of sophisticated programs."

Myth 2: Pascal is Limited in Power

The biggest myth about Pascal is that it is a language without power. Nothing can be further from the truth. My personal experience attests to the fact that Pascal is not only a language designed to encourage well-written and manageable code, but that it has evolved into a powerful language that supports industrial needs (see About the Author on page 1) and commercial needs (see http://pascal-central.com/pascalware.html#Commercial).

One problem with perception is that many look no further than the original unextended Pascal standard. While a strong language in itself, it does fall short of supporting more industrial strength needs. Extended Pascal evolved from Pascal to support industrial, scientific and commercial needs. Extensions to Pascal were developed by Borland and Apple Computer that provide the language with object-oriented capabilities.

The so-called "limited power" of Pascal may also refer to the lack of low-level and machine-oriented features which are inherent in C. It's true that Pascal's design is not conducive to poking around the machine at a low level, but there are constructs available within the language that support performance of low-level operations.

Low-level hacking may not be encouraged by the language, but it is not prevented either. Programmers have access to every allowed memory location via the use of memory pointers. In addition, there are low-level libraries on each platform that support the access and control of memory and devices. Any language can access these libraries as long as the call is properly defined. These low-level libraries are inherently machine-dependent anyway, so direct language support is inappropriate.

Finally, some have the misguided idea that C's inherent low-level nature makes executables more efficient than ones generated in Pascal. That, too, is a fallacy. The efficiency of the executable is only as good as the compiler. The truth is that Pascal's architecture lends itself to easy optimization by a compiler. In a recent interview, John Reagan, architect of the Compaq Pascal compiler and member of the X3J9 Pascal Standards Committee, writes, "The strong-typing of Pascal makes it easier for an optimizer to understand the program and provide better generated code. That isn't to say it can't be done for C, but it just takes more work."

"The point is that Pascal can produce efficient code and provide the additional benefit of strong-typing and optional run-time checks. You need not switch from Pascal to C just to get performance (at least on Compaq's OpenVMS or Tru64 UNIX platforms where our compiler runs)."

This last statement of is an important point regarding an efficient code compiler: the performance of a compiler is highly dependent upon the compiler architect. Unfortunately, while Pascal's design makes optimizations easier to implement in a compiler, that does not mean that all compiler architects take advantage of it.

To summarize: while the original unextended Pascal may have lacked certain functionality, fully supported Extended Pascal provides industrial strength power, and Object Pascal advances the language to support of object-oriented programming. Furthermore, the nature of Pascal lends itself to easy optimization by a compiler to produce efficient runtime executables. Finally, as Ingemar Ragnemalm, noted shareware author and co-author of the book Tricks of the Mac Game Programming Gurus, once wrote, "I can do everything in Pascal that can be done in C, but in a more elegant manner."

Myth 3: Pascal Has Weak String Handling Capabilities

The nature of myths makes it ironic that Pascal should be saddled with the same criticism made of C regarding poor string-handling capabilities. Nothing could be easier than working with strings in Pascal. String handling capabilities are built into the language, using the predefined STRING schema type for variable length strings, and PACKED ARRAY[1..n] OF CHAR for fixed length strings.

The elegance of Pascal strings is that they are so simple to use that there is really no need to understand how they are handled internally. A string is assigned using single-quoted text, such as:

myString := 'Strings are simple in Pascal';

Checking for null strings is also simple, with both of the following examples working equally as well on variable length strings:

```
1) if myString = '' then
       myString := 'Go 49ers!';
```

```
2) if length(myString) = 0 then
      myString := 'Go 49ers!';
```

For fixed-length strings, the length is always the fixed length regardless of the value, so the first option above would be the way to perform the comparison. The smaller string is padded with blanks to match the length of the fixed string it is being compared with.

Locating a string within a string is done simply with an INDEX function:

```
theIndex := index(myString,'9');
```

Using the value set for *myString* above, the results of this INDEX call would result in a value of 5 for theIndex. Pascal's position of characters in a string starts at one rather than zero (i.e., the third character is position 3, the fourth character is position 4, etc.). In some other languages, the first character is position 0, the second is position 1, etc.. In this regard, Pascal is more intuitive.

Extended Pascal also supports the "+" operator for string concatenation. Therefore, a string may be constructed as follows:

```
output_String := 'Employee #' + emp_no + '(' + emp_name +
   ') prefers to program in ' + emp_favorite_language;
```

Extended Pascal also provides support for reading and writing from strings just like one would read and write from a file. For example:

```
Var
   myAge:   string(10);
   age:     integer;

myAge := '29';
readStr(myAge, age);
```

The above READSTR call would result with a value of 29 in the integer variable Age. Likewise:

```
Var
   outputString:   string(100);
   empName:        string(30);
   boxes:          integer;

empName := 'Jane Doe';
boxes := 298;
writeStr(outputString,'Employee ',empName,' sold ',boxes:1,
   ' boxes of cookies.');
```

The above would result with the following value for *outputString*:

```
Employee Jane Doe sold 298 boxes of cookies.
```

There are also the predefined string functions of SUBSTR and TRIM, and other useful string support, all described in the Extended Pascal standard.

Myth 4: Pascal Does Not Support Object Oriented Programming

Initially designed and released by Apple Computer in 1986, Object Pascal was developed as an extension to Pascal to support object-oriented programming. Object support is incorporated in THINK Pascal, CodeWarrior Pascal, Borland Pascal and various open source Pascals. As an example, the following code illustrates Object Pascal under the Codewarrior Pascal dialect (see Figure 8 below).

```
Program OOP_Sample;

Type
   Employee = object
      firstName:   string;
      lastName:    string;
      hourlyWage:  real;
      Function     Pay(hoursWorked: integer): real;
      end;
   ExemptEmployee = object(Employee)
      Function     Pay(hoursWorked: integer): real; override;
      end;

Var
   anyEmp:   Employee;
   exEmp:    ExemptEmployee;

Function Employee.Pay(hoursWorked: integer): real;

   begin (* Pay with deduction for benefits *)
   Pay := hourlyWage * hoursWorked - 100;
   end;

Function ExemptEmployee.Pay(hoursWorked: integer): real;

   begin (* Pay with no deductions *)
   Pay := hourlyWage * hoursWorked;
   end;

begin
new(exEmp);
exEmp.Hire('John','Smith',15);
writeln('As exempt, pay ',exEmp.lastName,' $',exEmp.pay(40):8:2);
new(anyEmp);
anyEmp.Hire('Jane','Doe',15);
writeln('As a regular employee, pay ', anyEmp.lastName,
   ' $', anyEmp.pay(40):8);
end.
```

Figure 8: Object Pascal Sample

Myth 5: Pascal is Only an Instructional Language

There is no argument with the fact that Pascal is an excellent teaching language. Its design encourages good programming habits and supports the creation of complex data structures from simple, well-defined types. However, it is far more than just an instructional language. Pascal is used in commercial applications as well as in industrial and scientific environments. Most if not all of what a programmer would want to do in C and C++ can be done in Extended Pascal and Object Pascal. There

are entire systems built upon Compaq Pascal in industrial manufacturing shops, such as the company I work for, and there are also several commercial desktop applications which have been written in Pascal. Delphi, a Pascal development system available on Windows, is extended and object oriented, and is one of the most popular RAD (Rapid Application Development) systems on the platform. On the Macintosh platform, CodeWarrior Pascal and THINK Pascal are the most popular. Figure 9 below is an example of some of the commercial Macintosh applications written in Pascal.

1. InterArchy - *Full featured and award winning FTP client*
2. Ingemar's Skiing Game - *Action game*
3. Klondike - *One of the most popular Solitaire games*
4. Autoshare - *A full feature listserver and auto-responder*
5. Scripter - *Top selling development environment for Applescripting*
6. SuperLock - *File security program*
7. FlightMath - *Flight analysis program*
8. JacqCAD Master - *CAD program for Jacquard textile design*
9. NIH Image - *Image Analysis program for Biomedicine*

Figure 9: Commercial Quality Pascal Applications

Myth 6: Pascal is Not For Serious Programmers

In 1981, Brian W. Kernighan posted an article on the web, Why Pascal Is Not My Favorite Programming Language, that criticized Pascal as not being suitable for real programming tasks and referred to it as a "toy language". One of Kernighan's introductory comments regarding Pascal said, "Because the language is so impotent, it must be extended." The paper then exposes the shortcomings of the original unextended Pascal without mentioning the extensions.

It's relevant to note that the original intent of Pascal was to provide a solid language suitable for teaching programming - one whose implementations could be both reliable and efficient on then-available computers. Unextended Pascal was exactly that language. Early adaptors of Pascal, however, recognized the strengths and promise of the language and began to use it far beyond its original intent. The Extended Pascal standard was created to refine the language, to better support these commercial needs, and to establish Pascal as a language suitable for serious programmers.

The criticisms in Kernighan's paper have become outdated and mostly irrelevant with the implementation of Extended Pascal. The paper would not have been mentioned at all, except that the criticisms contained within the paper are used by many in the field today against current implementations of Pascal.

The following is a summary of those points, as well as the Pascal extensions which invalidate them.

> 1) The size of all arrays are part of its type; therefore, it is not possible to write general-purpose routines to deal with strings of different sizes.
>
> **With the introduction of Extended Pascal, variable length strings**

were implemented, along with several other powerful string capabilities (see item 3. String Capabilities).

2) The lack of static variables and variable initialization destroy the locality of a program.

Variable initialization is included in the Extended Pascal standard (see item 10. Initial Variable State).

3) The lack of separate compilation impedes the development of large programs and makes the use of libraries impossible.

Modularity and separate compilation is apart of the Extended Pascal standard (see item 1. Modularity and Separate Compilation).

4) The order of logical expression evaluation cannot be controlled, which leads to convoluted code and extraneous variables.

Short circuit boolean operators are apart of the Extended Pascal standards (see item 24. Short Circuit Boolean Evaluation).

5) There is no flow control due to the lack of RETURN and BREAK statements.

Although not in the standards, most current Pascals support function and control loop exits. In both Compaq and CodeWarrior Pascals, RETURN will exit a function. In Compaq Pascal, CONTINUE and BREAK are supported in control loops, and in CodeWarrior, CYCLE and LEAVE are supported.

6) The CASE statement is emasculated because there is no default clause.

The Extended Pascal standard now includes an OTHERWISE clause in CASE statements (see item 14. Case-Statement and Variant Record Enhancements).

7) The language lacks most of the tools needed for assembling large programs, most notably file inclusion.

Extended Pascal's modules provide much more sophisticated support for managing and assembling programs than primitive file inclusion methods (see item 1. Modularity and Separate Compilation). With modules, file inclusion is superfluous.

8) There is no escape from Pascal's strong typing controls.

Although not stated in the standards, type casting is supported in modern Pascal implementations. In Compaq Pascal, "::" is the casting operator (e.g., myVar::myType). In CodeWarrior Pascal, casting is implemented using parenthesis (e.g,. myType(myVar)). The language has evolved without sacrificing the tremendous benefits of strong typing (see Chapter II. The

Pascal Architecture).

Return to Table of Contents Next Chapter
