

# A Guide to Extended Pascal and Objects

This document is intended for readers having some familiarity with earlier versions of Pascal. It contains a summary of the new features of Extended Pascal, and a survey of the provisions for Object Oriented programming. It also outlines the enhancements included in this implementation, in particular those for exception handling.

## Contents

Background

Simple enhancements

- Non-decimal numbers
- Constant expressions
- Relaxed ordering
- Enhancements to functions
- Implementation characteristics
- CASE enhancements
- Short-circuit operators
- Protected parameters
- Set extensions
- Complex numbers
- Exponentiation operators
- Inverse of `ord`
- Underscores

Character strings

- Capacity and length
- Operations
- Parameters
- Strings in the heap
- Implementation points

File extensions

- File binding
- Direct access

Date and time

Initial values

Array and record constructors

## Modules

- Export and import
- Module heading
- Module block
- Supply chain
- Predefined interfaces
- Implementation points

## Nonstatic types

- Schemata

## Object-Oriented programming

- Objects and Classes
- Inheritance
- References
- Classes and views
- Creating and destroying objects

## Exception handling

- Exception hierarchy
- The TRY statement
- User-defined exceptions

## Other additions

## Background

The programming language Pascal was originally designed by Professor Niklaus Wirth, and named after a French mathematician and philosopher who was widely admired for the clear and direct nature of his ideas. Wirth set out two principal aims for the design of Pascal: that it should be secure, systematic and coherent, so far as possible avoiding arbitrary restrictions, and that it should be suitable for efficient implementation on the currently available machines. That was in the early 1970's, and it was these aims that made Pascal one of the natural choices ten years later when the "currently available machines" came out of the computer room and onto the desktop. The personal computers of the 1980's had the capacity to support efficient implementations of a relatively simple language, and its systematic and coherent nature made Pascal attractive for the teaching of programming. Taken together, these influences gave a great boost to its popularity, but it also became apparent that there was a potential conflict between the twin aims of security and efficient implementation.

Probably the most important point was that the original definition had no provision for separate compilation. The machines on which it was first implemented were relatively large, but on the limited microcomputers of the early 1980's, that was a serious shortcoming, and at least two quite distinct kinds of solution were adopted. Then again, the original language was good at handling arrays of floating-point numbers, but the facilities for manipulating variable-length character information were distinctly rudimentary. Microcomputers were becoming widely used in areas where ease of processing character information was more important than number-crunching, and string-handling additions were devised. In this case, one definition was dominant in the micro field, but was not so widely accepted for larger machines.

With the aim of establishing a common base, a definition at essentially the level of Wirth's original was published as a Standard in 1983, and work continued – at first mainly in the US – on an "extended Pascal" that would provide secure separate compilation, character strings, and other important features. This project gave rise to a new Standard\* which after the formal approval stages was published in 1990/91. The standardisation committees then moved on to consider Object Oriented programming, and additions were devised and published as a Technical Report. The Prospero implementation is based largely on the Extended Pascal Standard and the Object Oriented additions, and this document sets out the main features.

\*ANSI/IEEE 770X3.160-1989 and ISO/IEC 10206:1991

## Simple enhancements

To begin with, there are a number of simple enhancements that make Pascal easier to use. Some have been widely implemented in the past, though not always in the same way, and it may surprise some to find that they were not part of the language from the beginning. In other cases, the enhancements improve consistency by removing restrictions that may have become so familiar that they seemed inevitable.

### Non-decimal numbers

Numbers (called **extended numbers**) can be introduced into source programs, using the notation `base#number`, for instance `16#FF` or `8#377` (hexadecimal `FF` = octal `377` = decimal `255`). Many Pascal implementations provide a means of writing hexadecimal constants, using an initial `$` character, or a trailing `H` as in the assembler convention. The freedom to specify the number base, though, allows a range of possibilities from 2 (binary bit-masks) to 32.

### Constant expressions

These are also found in a number of existing implementations, and can be used where in the original language only individual constants were allowed, such as in definitions. In Extended Pascal, constant expressions can employ almost all the predefined functions as well as operators.

```
CONST linefeed = chr(10);    {ASCII}
      buflen   = 200;
TYPE  buffinx  = 0..buflen-1;
      buffer   = ARRAY [buffinx] OF char;
```

As that short example shows, constant expressions allow a base value to be carried into other definitions, avoiding the need for additional modifications if a change is needed; changing `buflen` automatically updates `buffinx` and `buffer`. They are also not confined to integer type, for instance `linefeed` above is of type `char`, and string constants can be built up. The expressions are evaluated during the compilation process and impose no execution penalty.

Constant expressions can appear in various situations, both in declarations and definitions, and also within executable statements such as `CASE` statements, where a constant is required. Here are a few more simple examples.

```
CONST tabchar  = chr(9);    {ASCII}
      heading  = 'Name' + tabchar + 'Address';
      weekdays = [Monday..Friday];
```

## Relaxed ordering

Declarations and definitions (CONST, TYPE, VAR etc.) can be repeated and can occur in any order. Originally, Pascal imposed a specific order on these parts, and would not allow constructs such as the following.

```
TYPE  rainbow = (violet, indigo, blue, green,
                yellow, orange, red);
CONST firstcol = violet;
      lastcol  = red;
```

Having constants equated to enumeration values, you can write

```
FOR col := firstcol TO lastcol DO ...
```

and avoid naming the values themselves. The `rainbow` enumeration is unlikely to change, of course, but many others are more volatile.

The usefulness of some other extensions is much enhanced by relaxed ordering. You can for example define a constant of an array or record type by means of a constructor. When TYPE can precede CONST, such constants can be named.

There are two constraints on relaxed ordering. The first is a natural extension of a familiar rule: there can be no forward references from one part to another. The definition of a pointer type can still precede that of its domain type, but the reference must be resolved within the same definition part. The other is also familiar: an identifier can have only one meaning in one scope. An identifier introduced near the end of a block (by a VAR declaration following one or more nested procedures say) applies backwards to the start of the block as well as forwards. The consequences of this may be harder to foresee; take the following example.

```
PROGRAM xyz (output);
...
FUNCTION rms (a,b: real): real;
  BEGIN rms := sqrt(sqr(a)+sqr(b)) END;
...
VAR  sqr: ARRAY .. OF .. ; {illegal!}
...
END.
```

The declaration of the variable `sqr` applies to the whole program block, and the earlier reference to the built-in function with the same name is strictly the illegal construct, but in practice it is the variable declaration that will be flagged.

## Enhancements to functions

Function results can be of any assignable type, including arrays and records, and the processes of indexing, field selection, and pointer-following (^) can be applied directly where appropriate. For example, there is a predefined record type called BindingType associated with binding of files, and a function `binding` which returns such a record. One field in the record is a Boolean called `bound`, and you can write

```
IF binding(f).bound THEN ...
```

Again, classic Pascal allowed functions that returned a pointer, but required the value to be stored before it could be dereferenced. In Extended Pascal, an immediate dereference is allowed.

In the heading of a function, the result variable may be given a name, different from the function name, which can be referenced without causing a recursive call of the function.

```
FUNCTION min3 (a,b,c: integer) = ans: integer;  
BEGIN  
  IF a < b THEN ans := a ELSE ans := b;  
  IF c < ans THEN ans := c;  
END;
```

The outcome is no different from declaring a local variable `ans` and assigning it to the result just before exit from the function, but it is clearer, and a little more efficient, to present it like this.

As will be seen, an initial value can be associated with a type, and when a function is defined with such a result type, the function result acquires a default value on entry.

## Implementation characteristics

Just as the predefined constant `maxint` gives you information about the integer type in a particular implementation, there is a constant `maxchar` (the character with largest ordinal value), and constants `maxreal`, `minreal` and

## CASE enhancements

Both in variant record declarations and CASE statements, constant expressions and ranges are permitted in constant lists, and an **OTHERWISE** clause may be introduced to complete the list.

```
CASE ch OF
  '0'..'9':          digit;
  'a','e','i','o','u': vowel;
  OTHERWISE          other;
END {case};
```

The concept of an OTHERWISE clause (called a **completer** in the Standard) is frequently found as an extension to the CASE statement in Pascal implementations, but in some the word ELSE is used instead of OTHERWISE.

The introduction of ranges into variant record declarations allows a clearer and more compact representation, but the need for a completer may be less obvious. The reason is that in the interest of security the original Pascal standard required the cases in a variant record to match all the values of the tag-type. With the completer notation, this can conveniently be done, without losing the original security.

## Short-circuit operators

There are variants of the Boolean operators AND and OR which guarantee that an expression is evaluated no further than is necessary to determine the result. The new variants are called **AND\_THEN** and **OR\_ELSE**, and they can be used as in the next examples to simplify conditional code or save unnecessary operations.

```
WHILE (p <> NIL) AND_THEN (p^.fld2 = 2) DO ...
IF BoolFunc1 OR_ELSE BoolFunc2 THEN ...
```

In the first, if *p* is NIL control immediately leaves the loop, and the illegal reference *p*<sup>^</sup> is never executed. (It may appear harmless, but if it were, an exception would be raised.) In the second example, the call of BoolFunc2 is avoided when the result has already been established by BoolFunc1 returning true.

Some implementations, including this one, in fact treat simple AND and OR in the same way, but the distinct notation emphasises points which depend upon short-circuit for correct behaviour, particularly if the code is moved elsewhere.

## Protected parameters

A value or VAR formal parameter may be declared to be `PROTECTED`; the code within the procedure or function must then not contain any statements that might modify the parameter. (The technical term is `threaten` the parameter.) A caller passing a variable to a protected VAR parameter, in particular, knows that there is no risk of the contents being overwritten, and does not need to make a copy first; in the case of a large structure this might represent a significant saving. Declaring a protected value parameter indicates to the reader of the program that it retains its entry value throughout the execution of the procedure, and to a compiler that certain optimisations may be possible.

These headings show the two forms.

```
PROCEDURE orange (PROTECTED pip: real);
```

```
FUNCTION squeaky (PROTECTED VAR fat: bigarray): real;
```

The code within the procedure or function must not threaten such parameters, either by direct means like assigning to them, by passing them on as VAR parameters unless the latter is also `PROTECTED` or by reading into them, or by subtler means such as changing the active variant in a record.

The concepts of protection and guarding against threats are used in other contexts also. The control variable of a FOR loop is protected against threats from code in nested blocks, and the `PROTECTED` attribute can be applied to other variables (see [EXPORT](#).) The checks against threats are all ones that can be done at compiler time, so the security has no run-time cost.

## Set extensions

Sets have always been a distinctive feature of Pascal. Three gaps in the facilities for working with sets have been filled.

A new operator `><` evaluates the symmetric difference (the exclusive or) of two set values.

There is a new predefined function `card` which returns the cardinality of a set (the number of members present).

The FOR statement includes a new form in which the control variable is given in turn the values defined by a set, as here.

```
FOR n IN [5,7,10..17,21] DO ...
```

(You cannot specify the sequence in which the values are to be taken. In this implementation it is ascending order, but this is not a requirement in the Standard. In particular, the written order is not significant.)

## Complex numbers

A new scalar data type `complex` is provided. The internal representation is not specified, though it seems likely that the form employed in Fortran will generally be adopted. There are functions to obtain the real and imaginary parts (Cartesian view), the magnitude and argument (polar view), and to construct a complex value from either pair of inputs. The mathematical operators and functions available for type real can also take complex arguments and return complex results, apart from comparisons, where only equal and unequal are defined. Complex values cannot be directly read from or written to textfiles, but the parts can be handled separately quite easily.

```
Z1 := cmplx(x,0.5y);    {real and imaginary parts}
Z2 := cos(z1 * 5.5);
writeln(re(z2),im(z2));    {Cartesian view}
writeln(abs(z2),arg(z2));  {polar view}
```

## Exponentiation operators

Two exponentiation operators are included. `POW` raises a value to an integer power, and `**` accepts a real exponent. In either case, the left-hand operand can be integer, real or complex. An integer operand of `**` (as with the `/` operator) is cast to real before the operation. Exponentiation has higher priority than the multiplying operators.

```
k := (-1) POW j;
f := x POW 3 + y POW 3;
z := y ** x;
```

The operation `POW 2` gives the same result as the `sqr` function that has always been part of the Pascal definition.

## Inverse of ord

While `chr` has been available as an inverse of `ord` for characters, there has been no equivalent for other types such as enumerations. The `succ` and `pred` functions in Extended Pascal take an optional second parameter of integer type, which defaults to 1 to produce the familiar one-parameter form. You can use the second parameter to combine multiple steps, so that `succ(Monday, 3)` for example is `Thursday`, which is neater than `succ(succ(succ(Monday)))`. More generally, because `ord(Thursday)` is 3, it will be seen that the two-parameter form of `succ` can be used as an inverse of `ord` for enumerations.

## Underscores

An underscore is allowed as a significant character within identifiers, as in many existing implementations. The Standard prohibits them as the first or last character, and leading underscore is generally reserved by ANSI for private names needed by language suppliers; in default mode, this implementation does not reject leading or trailing underscore, but does so at the strict Extended Pascal level. Leading underscores in particular should be avoided unless there is some good reason.

## Character strings

The definition of classic Pascal had facilities for string literals, and for variables with types based on packed arrays of characters, but lacked the concept of variable-length strings. Many microcomputer implementations incorporated the UCSD definition of strings, recognisable by the `concat`, `copy`, `pos` and other functions; on larger machines, however, other forms were adopted. Extended Pascal includes provision for dynamic string types, and unifies them with individual characters and the packed arrays of characters inherited from the original definition.

### Capacity and length

Just as with numeric types, you can declare string-type variables and manipulate string-type values. A string-type value can be obtained from a variable, from a literal, or as the result of a string operation. String variables are declared with a maximum *capacity*, for instance:

```
VAR    s1,s2: string(200);
        fname: PACKED ARRAY [1..20] OF char;
```

Variables `s1` and `s2` are dynamic strings with a capacity of 200; `fname` is a “fixed string” as found in classic Pascal, and has a capacity of 20.

String values have a *length* (number of characters). A dynamic string variable such as `s1` can hold a value of any length from zero up to its capacity, and provided it is correctly used the Pascal system keeps track of the current length. With a fixed-string such as `fname`, the length of the contents is equal to the capacity; when a shorter value is assigned to it, it is padded on the right with space characters. A reference to a dynamic string variable produces a general string value whose length and contents are those stored in the variable; a reference to a fixed-string variable produces the padded-out contents. A variable of type `char` has a capacity of 1, and always produces a value of length 1. Character and string literals produce values of their apparent length, which may be zero.

You can index individual characters, for example `s1[i]`, or substrings such as `fname[1..k]`. Such references can be assigned to as well as read. An index applied to a dynamic string variable, however, must be within the length of the current contents.

It will be seen that in the example above, the dynamic string variables were declared as `string(200)` rather than `string[200]`, which might be more familiar. The type `string` is seen as a template, and the 200 as a parameter. The parameter (or more correctly *discriminant*) may be, as here, a compile time constant, or in suitable cases may specify a run-time value; the capacity, and hence the size, are then decided at run time. In your program, you can always refer to the capacity of a dynamic string variable `s` (say) as `s.capacity`.

You can supply an initial state as part of a string type. The following examples show a dynamic-string and a fixed-string declaration, each with an initial state.

```
VAR str200: string(200) VALUE '';
    pac20: PACKED ARRAY [1..20] OF char VALUE '';
```

The effect in the two cases is rather different. The dynamic string `str200` is initialised to empty (zero length), whereas the fixed string `pac20` is filled with spaces, as it would be if a zero-length value were assigned to it.

## Operations

String values can be concatenated using the `+` operator, which can also be included in the definitions of constants such as `'ABC'+chr(13)`. There are built-in functions to perform the commonly-needed string operations:

```
length(s)      returns the current length of s
index(s1,s2)   search for s2 in s1
substr(s,i,c)  return part of s defined by i and c
substr(s,i)    return tail of s beginning at i
trim(s)       remove trailing spaces
```

Those familiar with the UCSD string definitions will see that the `+` operator does the job of the `concat` function, the `length` operation is essentially unchanged, and `index(s1,s2)` is equivalent to `pos(s2,s1)`. The first form of `substr` is equivalent to `copy`. The second form and the `s[i..j]` notation are new ways of manipulating substrings. A difference from some UCSD implementations is that the arguments to `length` and other functions can be general string values, so for example you can write

```
len := length(s1+s2);
inx := index(substr(name,j),' ');
```

String values can be compared in two ways. You can use relational operators such as `<` or `<=`, or you can use some new functions called `eq`, `ne`, `lt`, `le`, `gt` and `ge`. When the values have different lengths, the operators work as though the shorter was padded with spaces, whereas the functions take the length into account. In the ASCII character set the only practical differences arise with equal and unequal, and then only when the operands match as far as the length of the shorter.

Strings can be read from or written to textfiles, and there are forms of the textfile `read` and `write` which take a string variable in place of the file, making all the conversion and editing processes available internally.

```
readstr(s,..) treat s as a line read from a textfile
writestr(s,..) similarly, "write" to string s
```

## Parameters

String values can be passed to value formal parameters, and string variables can be passed as VAR parameters provided the types agree. To this extent they are similar to other types. You can also declare formal parameters that adjust to the actual parameter at run time. They take the general type name `string`, for example:

```
PROCEDURE p (sval: string; VAR svar: string);
```

Here, any string expression, `substr(s1+s2, j, k)` for example, can be passed as the parameter `sval`, and at each call the formal parameter is made just big enough to hold it. Similarly, string variables of different sizes can be passed as actual parameters against the formal `svar`, and the latter effectively adjusts to match. The code in the procedure can refer to `svar.capacity` to discover the size of the current actual.

## Strings in the heap

As with parameters, a pointer can be declared with a domain type that is a string type of a specific size, or with one that is determined at run time. The following example shows the two possibilities.

```
TYPE str20 = string(20);  
VAR p1: ^str20; p2: ^string;
```

The statement `new(p1)` creates a variable of type `str20` with capacity 20 in the heap. With `p2`, a capacity is supplied with the call of `new`, thus `new(p2, cap)`, and the created variable has the specified capacity. The matching `dispose` has the single argument `p2`, after which another `new` can create a string of a different size.

## Implementation points

The preceding description has been of the provisions for strings in the Extended Pascal standard. One general remark about this implementation is that string operations are in most cases guarded against overflowing and overwriting other items in memory. An assignment to a string variable, for example, will be truncated if the capacity is insufficient, and if assignment range checks have been requested an exception will be raised. The exception to the general rule is when an indexed reference is made, and checking of indexes has not been requested.

The format of string variables allows for strings up to 32760 characters in length, and makes them suitable for passing immediately in API calls. You can also declare `widestring` variables to hold Unicode characters.

There are also some additional provisions to ease the porting of older programs: a `shortstring` type in the format often used, and acceptance of calls to functions such as `concat` with automatic conversion to equivalent Extended Pascal code.

## File extensions

Classic Pascal included the concept of file types, and variables of file types, but did not provide any uniform method of associating a program variable with an external file. Also, it was only possible to read and write files sequentially. Individual implementations provided their own methods of file binding and random access.

### File binding

Extended Pascal treats this matter in a very general way; it extends the concept of type by adding a new attribute of **bindability**. A variable of a bindable type is capable of being **bound** to an entity outside the program. The most obvious and widely-used example is in binding a variable of a file type to an external file, but the concept is intended to apply more widely than this. For example, an implementation may provide the means of binding a program variable to a real-time clock, to a screen buffer, or to a memory-mapped i/o port. Such applications of binding would of course be specific to the particular hardware and/or operating system. You specify bindability by including the word `BINDABLE` in the declaration, for example

```
VAR   f: BINDABLE FILE OF integer;
```

There is a predefined record type, called `BindingType` which holds binding information, new procedures `bind` and `unbind`, and a function `binding` to report the current status. The standard definition of `BindingType` requires only two fields, a name and a Boolean `bound` to indicate the status. In this implementation there are additional fields allowing you to specify that a file must already exist, that it will only be read by this program, or that exclusive access is required.

This implementation in addition provides functions `OpenRead` and `OpenWrite`, which employ the basic routines to give commonly-used file binding operations. They are Boolean functions that return `true` if successful:

```
IF OpenRead (f, 'January data.Dat') THEN ...  
ELSE {error};
```

Notice in passing that in this environment filenames can be longer than 8 characters, and can include spaces.

### Direct access

The provisions in Extended Pascal for direct access (random access) files are based on an addition to the original file type of an index, similar to an array index, by which the elements can be referenced. The external file can be made up of elements of any uniform size; they are often records, but other types are possible. The same external file can be opened for sequential access and direct access on different occasions.

The file variable is associated with the external file by means of `bind`, the mode of opening (attributes such as `readonly`) depending upon how it will subsequently be used. You then call one of the procedures `SeekRead`, `SeekWrite` or `SeekUpdate` to position it and prepare for a subsequent operation. After `SeekRead` or `SeekUpdate`, the specified element is available in the file buffer. This example displays a string taken from a file made up of strings of capacity 30 characters.

```
VAR    f: FILE [0..9999] OF string(30);  
      ...  
      SeekRead(f, i);  writeln(f^);
```

The element to be displayed is specified by the index value `i`. You can modify the contents of this element like this.

```
SeekUpdate(f, i);  
f^ := 'New value';  update(f);
```

## Date and time

A predeclared record type `TimeStamp` is provided, which contains fields for year, month, day, hour, minute and second. There are also two Boolean fields called `DateValid` and `TimeValid`. It is envisaged that implementations may add further details, and in this one there are fields for milliseconds and day of the week. A procedure `GetTimeStamp` sets the current values in a `TimeStamp` record, and also sets the Boolean fields to `true`. Functions `date` and `time` take a `TimeStamp` and returns a string containing the date or time in display format.

The most common application is likely to be obtaining and displaying the current date and time, but the record can be used in other ways, for example you can set a specific date like this.

```
VAR    ts: TimeStamp;
      . . .
      ts.year := 1993;
      ts.month := 1;      {January}
      ts.day := 22;
      ts.DateValid := true;
      writeln(date(ts)); {display the date}
```

The format for display of dates and times is defined by an implementation, but for the example above might well be '1993 Jan 22 '.

Dates and (more particularly) times will usually be obtained and displayed in terms of “local” time, which takes account of time zones. In the Windows environment there is also behind the scenes a “system” time which is UTC or Universal Coordinated Time, which as the name implies removes any local variations. When working across networks, in particular, it is important to have such a global means of identifying times.

This implementation also provides some library routines for obtaining the elapsed time between two `TimeStamp` records and other time-related operations.

## Initial values

In the classic Pascal definition, all variables are created in an undefined state (that is to say, they contain an undefined value). In practice, outer-level variables would often contain zero, but any local variables of procedures or heap variables would have random contents. In Extended Pascal, you can specify an **initial state**, which is automatically given to any variable when it is created. At the outer level, the implementation may preset the value in the program image; for local or heap variables an initialisation sequence is generated.

The initial state can be introduced in two ways. It can be associated with the declarations of particular variables or record fields, or it can form part of the definition of a type, in which case it is given to every variable or field of that type unless an individual declaration overrides it. The following example shows how this can be used.

```
TYPE  intz = integer VALUE 0;
      trec = RECORD
          a,b: intz;
          c: char VALUE '*';
      END;
VAR   recp: ^trec VALUE NIL;
```

The type `intz` is a version of type `integer` that has an associated initial state of zero. A variable of type `intz` behaves just like one of type `integer` except that it will initially contain zero. Fields such as `a` and `b` can be declared with type `intz` and will initially contain zero; in particular, if you call `new(recp)` the record created in the heap will have zero in fields `a` and `b`, and asterisk in field `c`.

In the same program, you might declare a procedure with local variables:

```
PROCEDURE pq (fp: integer);
  VAR   lint: integer VALUE 16#5C5C;
        lrec: trec;
  BEGIN ..
```

Whenever this procedure is activated, `lint` is automatically initialised to `16#5C5C` and the fields of `lrec` to `0`, `0` and `'*`.

An initial state can be defined by any constant expression that could be assigned to the variable by means of a statement, so for instance a real variable might have an initial state specified as `5.0`, `0.5E1` or `5`, which all have the same effect, or alternatively as say `sin(0.5)`. You can give an initial state to a string variable, or initialise a pointer to `NIL`. The elements of an array can be set to a uniform initial value by defining an element type with an initial state, for example:

```
TYPE  real1 = real VALUE 1.0;
VAR   vector: ARRAY [1..1000] OF real1;
```

In the case of variant records, you can use the initial state of a tagtype to define which variant is active when the record is created, as in the following elaboration of the earlier example.

TYPE

## Array and record constructors

Users of Pascal will be familiar with set constructors such as this:

```
[ '0' .. '9' , '+' , '-' ]
```

In Extended Pascal, **constructors** for arrays and records can be defined with related notations, which build structured values. A constructor is one form of expression. When all the ingredients are constant, the constructed value can be named as a constant, used to define an initial state, assigned, or passed as a value parameter. A constructor that includes any variable ingredients can be assigned or passed as a parameter.

An array constructor can list indexes of individual elements and the values to be given to those elements, and/or can provide a default value for any that are not individually listed. The array must be completely specified, and if not all the elements are given their own values, a “completer” must be supplied to fill any gaps.

```
TYPE  a10 = ARRAY [1..10] OF real;
CONST car = a10 [3..6,9: 5.5; 10: 10.5;
                OTHERWISE 0];
```

Here, `car` is a constant of type `a10`, in which elements 3 to 6 and 9 have the value 5.5, element 10 has the value 10.5, and the other elements are zero. The type name (`a10`) is needed when the type is not clear from the context.

A record constructor names record fields together with the values to be given; all the fields must be defined.

```
VAR    rec = PACKED RECORD
        f1: 0..99;  f2: char;
        f3: PACKED ARRAY [1..8] OF char;
        f4: integer;
    END
    VALUE [f1,f4: 10; f2: '$'; f3: 'Message'];
```

The record declaration includes an initial state defined by a constructor. In this usage the type of the constructor is clear from the context. The description of strings explains that an assignment to a fixed-string (a packed array of `char`) fills any remaining positions with spaces, and the same will happen when `f3` is initialised to `'Message'`.

If a record has a variant part, the constructor specifies the variant and supplies values for all the fields in that variant. The outline form is:

```
Typename [ fixed fields CASE selector OF
           [ variant fields ] ]
```

where `selector` is a constant of the tagtype.

Previous examples have shown constructors with simple constant components. The values can be any expressions, and when they are constant expressions the constructor can still be used as a constant. They can also be run-time values, but the constructor then can only appear within a statement. The notation is good at expressing multiple assignments of the same value, so for example the statement

```
ax := a10 [ 3..6,9: sqrt(sqr(x)+sqr(y));
           10: x*y;
           OTHERWISE x+y+5.6 ];
```

is compact and clear compared with any longhand equivalent. The compiler allocates temporaries to hold the values of computed expressions.

A further advantage of constructors is the security they offer arising from the requirement to define all the components of a structure (array or record, as the case may be). Even in a simple case, you can write something like this

```
matrix := mat [ OTHERWISE [ OTHERWISE sqrt(x) ] ];
```

and be sure that the code will adapt if the dimensions of the matrix change. This is particularly important if the dimensions are determined at run time, as described in nonstatic types.

The processes of indexing and field selection can be applied to structured constants just as they can to variables, and within statements an index can be a run-time value. You can pick out different elements of an array as they are needed, for example

```
TYPE  rainbow = (violet, indigo, blue,
                green, yellow, orange, red);
       colvec  = ARRAY [rainbow] OF real;
CONST wavelen = colvec [ violet: 0.0;
                        indigo: 0.0; blue: 0.0;
                        green: 0.0; yellow: 0.0;
                        orange: 0.0; red: 0.0 ];
```

## Modules

The standard definition of “classic” Pascal (first published in 1982) did not contain any provisions for separate compilation, which is an essential feature for construction of large programs and as the basis of shared libraries. For these reasons most implementations of Pascal have introduced means such as *units*, or *external* directives, to permit separate compilation of program components. It was recognised that Extended Pascal must provide for separate compilation, but do so in a way that maintains the security which Pascal provides within single programs.

Modularity in Extended Pascal is a structural device which may also be a vehicle for separate compilation. An executable program may contain, besides the main program component, one or more modules. A *module* is a program component at the same level as a main program, and has two parts, a module-heading and a module-block, which may be amalgamated or separate.

### Export and import

A module heading can contain definitions of constants and types, declarations of variables, and headings of procedures and functions. Any of these may be exported, by naming them in an export list, and they can then be imported and used by other modules or by the main program. Export and import are achieved through named *interfaces*, and the module heading is therefore sometimes referred to as the “interface part”. A module heading may export one or more interfaces, each containing a list of named components.

An interface so produced may be imported into the heading or the block of another module, into a main program, or (more rarely) into a procedure or function. The constituents of the interface, that is, the various items named in the export list, are then available for use anywhere within the module, main program, or procedure or function containing the import statement. Constituents imported into a module heading can also be re-exported, allowing for example amalgamation of interfaces to form a library.

A component may be renamed at the point of export if desired, the new name being the one by which it will be known on import by other program components. The concept of “protected” was described in connection with parameters. A variable can also be protected on export from a module, making it in effect read-only when imported. By default, all the components named in the interface become available in the importing environment; when needed, there are options to deal with name conflicts, and to selectively import from large interfaces.

## Module heading

Here is a small example of a module with combined heading and block. Module one exports the interface `i1`, containing two constants `upper` and `lower`. It has a minimal module block.

```
MODULE one;
EXPORT i1 = (lower, upper);
CONST lower = 0; upper := 11;
END {of module heading};
END {of MODULE one}.
```

Every module heading must have an export part and export at least one interface. (Except in rare cases, a module only contributes to the program through exporting, so the requirement is not imposing anything extra.) You can export more than one interface from a module, for instance to give access to special test information separately from the normal user details.

When a record type is exported, the names and types of the fields, together with any variant structure, are automatically included in the interface. Export of a procedure or function similarly carries with it the information about formal parameters that is needed to compile calls in importing modules. It is in the export list that you can specify the `PROTECTED` attribute, or rename constituents.

A module heading can import from another module, and may then re-export some or all of the imported items, allowing composite interfaces such as library definitions to be constructed. Here is another small module that imports the constants `lower` and `upper` from module one. It exports a subrange type and in another interface re-exports the two constants.

```
MODULE two;
EXPORT i2 = (subr);
      j2 = (lower, upper);
IMPORT i1;      {the original constants}
TYPE subr = lower..upper;
END {of module heading};
END {of MODULE two}.
```

This implementation provides a shorthand notation for export of all constituents defined in the module heading, when this is appropriate. It avoids the need to update the export list when new items are introduced.

## Module block

A module block, or “implementation part”, contains the actual declarations of any procedures and functions whose headings were given in the module heading. When there are no exported procedures or functions, as in the two earlier examples, the module block is generally trivial, and the heading and block are best combined. In a module which exports procedures or functions, on the other hand, the most convenient arrangement is usually to separate the two parts. This is because the heading contains all that is needed by other components of the program, and provided that it does not change, any modifications to the block are effectively isolated.

When they are separate, the module heading and module block are associated by the module name. The module heading begins

```
MODULE modulename interface;
```

and the module block begins

```
MODULE modulename implementation;
```

All definitions and declarations imported into or specified in the interface part are available for use in the block, and each procedure or function heading must have a matching definition in the implementation part.

A module block can import other interfaces, and it may also contain constants, types and variables, and definitions of other procedures and functions, which are not exported but remain local to the module.

The module block may also contain an *initialization part* and/or a *finalization part*. The former takes the form

```
TO BEGIN DO <statements>;
```

where <statements> are executed prior to the first statement of the main program, and the latter takes the form

```
TO END DO <statements>;
```

where <statements> will be executed after completion of the main program.

## Supply chain

It must be possible to arrange the program components in an order such that the export of an interface name precedes all imports of that name. The exporting module is said to “supply” the importing component, and the direction of supply must be such that the export precedes the import. The main program is at the end of the chain of supply, since it has no export part and cannot supply any other component.

The separation of module heading and module block allows an ordering in which the heading of module A supplies module B, which in turn supplies the block of module A. (Similar use can be made of forward declaration of procedures.) Separation also permits alternative implementations of one set of specifications, for example with and without diagnostics.

A rather different kind of advantage arises from the effect on automatic “make” processing. Any changes to a module block, which holds the implementation details, are self-contained; only changes to the heading will lead to recompilation of the importing components. Again, separation can be helpful when a large program is divided among a number of implementors; only the headings are critical to other modules and need to be shared.

### Predefined interfaces

The standard definition of Extended Pascal includes two predefined interfaces called `StandardInput` and `StandardOutput`, which make the standard files `input` and `output` accessible in a block that imports them. A particular case is the module block, where it may be convenient to write to `output`, for example, during testing. This implementation includes a further predefined interface `ExceptionHandling` containing the set of names and definitions needed for exception handling.

### Implementation points

The implementation of interfaces involves the compiler generating a file at the point of export which contains the information it will need when the interface is imported and the constituents are used. For record types, this includes the names and types of the fields, together with any variant structure; for procedures or functions it includes the types of any parameters. The interface file is read at the point of import, and the compiler’s internal definitions of the constituents are reconstructed. A similar method is employed to reproduce the definitions in a module heading and make them available when processing the module block.

When the compilation of a module gives rise to an interface and also to generated code, there is a possibility that the two may get out of step. Automated “make” processes will generally ensure that this does not happen, but the danger is that code generated from components that import the interface may be linked with code from an earlier or later version of the module. (It is not a new situation; an equivalent problem can arise with an Include facility.) To avoid the possibility, a consistency check is incorporated into the code and activated on program startup.

## Nonstatic types

The concept of data types is central to Pascal. Every variable and every value possesses a type, which governs how it may be used. Proper use of the type facilities increases the clarity and security of programs, by allowing the text to describe more exactly the programmer's intention. In the construction of any non-trivial program, consideration of the data structures that will be needed should accompany the design of any algorithms.

Types which can be fully defined at compile time are known as *static* types. All the types available in unextended Pascal, apart from the optional conformant array parameters, are static types. An array type such as

```
ARRAY [1..10] OF integer
```

for example is of fixed size, and the positions of its elements are also fixed, so that space can be allocated for variables and code can be generated to access them. The same applies to records with fields of static types.

Pascal has strong type rules which provide security, but within the simple framework where all types are defined at compile time they can be restrictive. Without losing security, Extended Pascal includes *nonstatic* types, that is, types that are only fully defined at execution time. One kind of nonstatic type is the *conformant array* feature, which was an option available also in unextended Pascal; another kind is the *schema*. Conformant array parameters allow procedures and functions to be written to which arrays of different sizes may be passed in different invocations. The bounds of the array index type are available to the code within the procedure or function so that it may adapt to the actual parameters. This provides a degree of flexibility that suits, for example, mathematical procedures that manipulate arrays, since the only requirement is that an actual parameter "conforms" to the formal in the sense of having the same number of dimensions, index types, and final element type.

## Schemata

Schema types have some similarities with conformant arrays, but are a more general facility. A *schema* defines a family of related types from which specific members may be selected, and a type so produced (a form of nonstatic type) can be used in most respects like a conventional static type. You can for example declare variables of such a type. The family may be arrays with index bounds determined at execution time, rather like conformant array parameters, but may also for example be a family of sets, or a family of records defined by variants. The selection is made by substituting either compile-time or run-time values for "parameters", or more correctly *discriminants*, in the type definition. As an illustration, dynamic character strings are formally defined in terms of the predeclared schema type `string`, from which an individual type is derived by specifying a `capacity`.

A local variable of a procedure can have a type produced from a schema, involving one or more parameters of the procedure in the selection. You might for instance pass a parameter which is then used to specify the index bounds of a local array.

Just as the schema name `string` can be quoted as the type of a formal parameter of a procedure, which then adapts to the actual parameter passed with each call, so can the name of a schema you have defined. The actual parameters must have been produced from your schema, and the code within the procedure can use the names of discriminants to discover information about them, such as array bounds.

Again like the `string` schema, the name of a schema you have defined can be given as the domain type of a pointer, and an individual type selected by calling the procedure `new` with values for the discriminants. A variable of the appropriate size is allocated in the heap. The discriminant values are recorded together with the variable, and to dispose the variable you simply call `dispose` with a reference to the pointer. This is an altogether more secure arrangement than the local extensions that have previously been introduced for variable-size heap allocations which require the size to be repeated on `dispose`.

Whereas static types allow sizes and offsets to be fully determined at compile time, schemata introduce the possibility of some of these parameters being known only at execution time. The size of an array with schematic index bounds is one example, and when such an array is made a field of a record the size and structure of the record are not fixed. A type which is produced from a schema, or which incorporates such a type, is called nonstatic. These types require some operations which are performed at compile time for static types to be postponed, but this is transparent to the programmer; the implementation maintains enough information to ensure that their use can be made as secure (and convenient) as that of conventional static types.

## Object-Oriented programming

The Extended Pascal Standard did not include any facilities for object-oriented programming, but a Technical Report was published subsequently that defined a set of additional features. The description that follows is based on the provisions in this implementation, which follow the Technical Report in all but a few details. A range of examples is provided with the software.

### Objects and Classes

An *object* is an instance of a data structure with which are associated a set of services. It belongs to a *class*, which defines the structure of objects and the repertory of services that are shared by those objects. The objects are *members* of the class.

A class generally *inherits* from one or more parent classes. The class data structure contains the components of the structures of each parent, together with optional new components. Definitions of services are inherited from each parent class; the detailed algorithms may be inherited unchanged, or may be *overridden* by new versions in the new class. New services can optionally be added. An object is defined to be a member of its parent classes as well as of its own class.

There are three kinds of class: *concrete*, *abstract* and *property*. Concrete classes define complete objects. Property classes define data and related services which can be inherited and form parts of objects. Abstract classes include services which are incompletely specified and must be overridden when inherited.

The services for objects are provided by *constructors*, which are used to create and prepare new objects, and by *methods*, which are specialized procedures and functions that manipulate objects of the associated class. A method is invoked to operate on a particular object, and if the method has at some stage been overridden the version appropriate to the type of the object is automatically selected. The Technical Report also defines *destructors* which remove objects that are no longer needed; a default destructor is provided in the implementation, and adding further destructors is not recommended.

Objects are created dynamically, and accessed through *references*, similar to pointers. A variable or field declared with a class type holds a reference, but the up-arrow notation is not used when addressing the object; rather, code is written “as if” the object itself were located at the reference.

When a class type has been defined, *views* of it can also be defined that specify a set of components (data fields and procedural items) which are “visible”. Objects themselves are not affected, but if a reference is declared with a view type then access is restricted to the components that are visible in the view. Exporting a view rather than the original base type therefore retains privacy over selected parts of objects or selected services.

In the context of an Extended Pascal module, any class types or views that are to be exported are defined in the module heading. Full declarations of the methods, constructors and destructors are placed in the module block. If the type is not to be exported then the definition as well as the declarations of procedural items can be situated in the module block.

Extended Pascal modules, with definitions exported through interfaces, and the possibility of amalgamating interfaces through import and re-export, make an excellent vehicle for libraries. There are features such as schema types and procedural parameters which allow the definitions of libraries to be much more flexible than is possible in some other languages. Dynamic linking allows the implementation code of a library to be replaced without the need to modify or even to relink programs that employ it. Allied to this, the object-oriented approach provides a framework that allows libraries to be developed which can be inherited and specialized by users, who can “override” individual actions while still making maximum use of the original library code.

## Inheritance

An analogy of the difference between classes and objects can be found in a cookery recipe that gives a list of ingredients, followed by directions (the “method”) for turning the ingredients into a finished product. Each time the recipe is employed, a new set of ingredients is turned into a new product by following the same method. From the same area, we can find an analogy for inheritance. The recipe for a white (Béchamel) sauce contains ingredients and a method. It is seldom produced as an individual item, but is a common starting point for other more interesting dishes. In object oriented terms, the latter “inherit” the white sauce recipe.

A class type is not always based on inheritance from parents, but in practice many are. When a new class type inherits from a parent class, an object of the new class includes the data from an object of the parent class, together with the services (methods and constructors) which operate on the data.

```
TYPE newclass = CLASS (parentclass)
    f1,f2: integer VALUE 0;
    FUNCTION ready: Boolean;
END;
```

The definition of `newclass` inherits from `parentclass`, and adds two data fields and a function. An object of the new type will contain all the data of a parent object, together with the new fields, and any methods from the parent class can be invoked as well as the new function.

The definition of the new class can specify that an inherited method will be replaced, or *overridden*, to take account of new features. For example, if the parent class included a procedure `Prepare`, the definition might be extended like this.

```
TYPE  newclass = CLASS (parentclass)
      PROCEDURE Prepare; override;
      f1,f2: integer VALUE 0;
      FUNCTION ready: Boolean;
      END;
```

Later, in the program block or module block, there must be the full definitions of `Prepare` and `ready`. They might look like this.

```
PROCEDURE newclass.Prepare; override;
BEGIN
  INHERITED Prepare;
  f2 := trunc(10*rand);
END;

FUNCTION newclass.ready: Boolean;
BEGIN
  ready := (f1 > 0) AND (f2 > 0);
END;
```

The definition of a parent class includes the names of data fields, and the headings of any methods, but the author of the new class may or may not know the full definition of `Prepare` in the parent class, or for that matter whether it may be modified over time. The new version therefore starts by invoking the parent version. Some other points raised by this example are discussed later.

## References

An object is created in a space which is not accessible to normal Pascal code, but at the time it is created a **reference value** is returned. This reference is the key to the object, and resembles in some respects a Pascal pointer. To perform any action on an object, you must start with a reference. A variable or field that is declared to be of a class type holds a reference, not the actual object. A reference can be assigned or passed as a parameter, and two references can be compared. However, there are important differences between pointer values and reference values.

A variable can hold a reference value that identifies an object of the class with which the variable was declared, or one of any descendant class. This is unlike a pointer, which is associated with just one domain type. A reference value can be assigned to a reference variable of its own type, or one of an ancestor type. In the second case, the value assumes the ancestor type, and only the fields and methods that were inherited from the ancestor are accessible. (If any of the inherited methods have been overridden, however, it is the overridden version that is invoked.)

Suppose you have a variable `cv` of type `newclass`, containing a reference. You can access fields in the object like this

```
cv.f1 := cv.f2;
```

or invoke methods like this

```
IF cv.ready THEN ..
```

The coding to implement the method can access the object through which it was called, and refer directly to fields such as `f1` and `f2`, as in the example function `ready` above.

## Classes and views

The definition includes three kinds of class. The preceding description has been of *concrete* classes, from which new objects can be created. Each concrete class inherits from exactly one concrete parent class; if no such parent is explicitly named, then the new class inherits from the predefined `Root` class, which is the common ancestor of all concrete classes. The `Root` class contains a constructor called `Create` and a destructor called `Destroy`, so every concrete class includes these by inheritance, and can override them to provide additional functionality.

Because a concrete class inherits from just one concrete parent class, a second kind of class forms the basis of multiple inheritance. They are the *property* classes, which define not whole objects, but rather aspects or “properties” to be inherited. Any class, including a new property class, can inherit from one or more property classes, provided they have no common ancestor. A property class with no named parent starts a new ancestry. The following example shows the declaration of a property class.

```
TYPE person = PROPERTY CLASS
    lastname, given: string(20);
    PROCEDURE register;
END;
```

It is a new property class, with no ancestors, ready to be inherited by other classes.

The third and last kind of class is an *abstract* class. Essentially, an abstract class is a concrete class that is not fully defined, but must be inherited and refined. It contains one or more method headings that have no implementation at the level of the abstract class; the full definition must be provided at a descendant level before an object can be created. Such a class may be useful in a class library when the author provides a place-holder for later definition; for example, a sorting process may require that a client supplies the definition of a comparison function.

When a class type has been defined, giving a complete picture of fields and methods, one or more *views* of the class can be specified, which provide limited visibility to the contents of the class. Defining a view does not change the layout of an object, or modify the behaviour of any methods, but removes the names of some items from the full definition. By means of views you can partition the class into public and private, or indeed produce a range of access levels.

```
TYPE mysort = VIEW OF sorting
    PutItem, GetItem, Precedes
END;
```

This view assumes that a class `sorting` has been defined, and that it contains the items `PutItem`,

## Creating and destroying objects

To create a new object, you call a constructor in the class; after creating the object it returns a reference. In this usage, it behaves like a function. Any concrete class includes at least the constructor `Create` inherited from `Root`, which can be overridden to perform initialising operations that may be needed to correctly prepare the object. If overridden, the new definition should as a rule invoke the inherited version, in case it too performed extra tasks.

```
CONSTRUCTOR newclass.Create; override;
BEGIN
    INHERITED Create;
    <extra actions>
END;
```

In this usage, a constructor behaves like a procedure. There are examples of overriding `Create` in the issued software.

To remove the object, you invoke `Destroy`, which may similarly be overridden if there are actions to be performed. In this case the new definition should invoke the inherited version after (rather than before) any immediate finalisation tasks.

## Exception handling

*Exception handling has been the subject of proposals in the Standards committees, but no formal document has been issued. This implementation includes facilities that are broadly based on the proposals put forward, but do not represent an agreed standard definition.*

The term *exception* is used to describe an abnormal condition arising during execution of a program. Some conditions are the result of incorrect programming (attempting to read past the end of a file, for instance), other may arise from situations that are less readily guarded against. They are in most cases detected by routines within the Pascal run-time system, as a result of checks made while performing some service on behalf of the program, but may also be errors trapped by the hardware or operating system.

By default, exceptions are notified to the user of the program by means of a message, which indicates the cause and location. You can intercept exceptions, examine the cause, and if possible take remedial action to avoid the user notification.

### Exception hierarchy

The exception causes are organised into a hierarchy, somewhat resembling a class structure. There are three main subdivisions, one of which is further subdivided. Here is the outline classification.

```
Language
  Access
  Assignment
  BindUnbind
  Control
  File
  Format
  Nonstatic
  Numeric
  Variants
  Other
System
User
```

Within each subclass, such as Access, there are a number of more specific conditions. Access, for instance, includes ArrayIndex, StringIndex, Pointer, Field, Undefined and Violation categories. The cause of an exception is specified at the most specific level available, such as StringIndex or Undefined, but may be intercepted either at that level or at a containing one; a test for Access would pick up StringIndex, Undefined, or other errors in the category.

The exceptions hierarchy, and other definitions needed when exceptions are to be intercepted and handled within the program, are contained in an interface called `ExcepHandling` which is supplied with the package.

## The TRY statement

You intercept exceptions by inserting **TRY** statements into your program. The general form is

```
TRY <region>
  ON exception-name-1 DO handler-1;
  ON exception-name-2, exception-name-3 DO handler-2;
  ..
END;
```

The *region* of the try statement is a group of one or more statements, which can include procedure calls, within which exceptions will be trapped. If an exception occurs, the ON clauses are examined in turn. If the cause is within the category or categories named, the matching *handler* statement is executed.

For example, when numeric data is being entered, errors in the format of numbers might be trapped like this:

```
10: TRY  readln (intvar, realvar);
      ON ExcepFormat DO
      BEGIN
        writeln ('Incorrect format; repeat input');
        GOTO 10;
      END;
END;
```

If no handler is found, and the try statement is within a procedure, the call or sequence of calls is followed backwards. If any call was part of another TRY region, and the cause now matches one of the ON clauses, that handler is executed.

When control is passed to a handler, it is as though a GOTO had been performed from the exception point to the handler. The details of the exception are recorded in a structure, and the handler can obtain access to them.

## User-defined exceptions

The User category in the exception classification allows for user-defined exceptions to be intercepted and handled by the same TRY mechanism as those raised by the runtime support routines. This technique can be used for example to replace the model of escape from an error situation by means of a GOTO to a label in the main program block. There is a routine `RaiseUser` that accepts parameters which a handler can examine to distinguish different errors.

## Other additions

This implementation is largely based on the definitions in the Extended Pascal Standard and the Technical Report on object-oriented programming. However, besides the Exception Handling provisions, there are some local additions to the language to improve access to the Windows environment, and to facilitate porting of older programs. A few of these have been touched on in the descriptions of strings, file extensions, and elsewhere, and they are summarised below. In addition, there is a range of library routines whose definitions are not built into the compiler, but are provided in the form of interface files.

Identifiers can be used as labels, as well as the conventional Pascal numeric labels.

The “signature” of a forward-declared procedure or function can be repeated in the definition, as can that of a class method.

Notations are provided for “total export” of all items defined in a module heading, and for re-export of the contents of a specified interface.

There is an additional Unicode 16-bit character type, and an associated `widestring` type. The UCSD string functions such as `concat` and `copy` are available, and there is a `shortstring` type for data conversion.

`OpenRead` and `OpenWrite` functions encapsulate commonly-used forms of file binding.

Enumeration values can be written to or read from textfiles by name. Integer values can be written in non-decimal bases such as hex.

Alternatives to the `interface` directive in a module heading allow for the definitions of assembler-coded routines or API functions to be expressed in Pascal and imported into other modules.

There are additional numerical functions `max` and `min`, taking two arguments of compatible ordinal or real types; a function `pi` that provides a very accurate value of the constant; functions `arcsin`, `arccos`, and hyperbolic forms of `sin`, `cos` and `tan`.

Operators `AND` and `OR` can be used between integer operands to perform bitwise operations. Shift and remainder operators are provided.