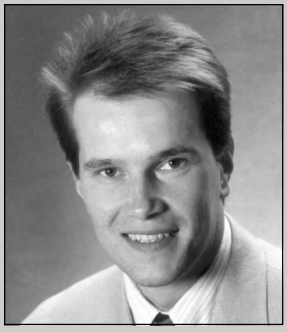
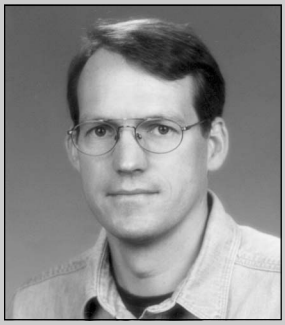
Steering Clear of the Top 10 Pitfalls Associated with ABAP Fundamental Operations and Data Types

Christoph Stöck and Horst Keller



Christoph Stöck
Business Programming Languages Group
SAP AG



Horst Keller
Business Programming Languages Group
SAP AG

(complete bios appear on page 22)

As every good programmer knows, a textbook treatment of a development topic will only take you so far. True mastery of software development comes about with experience — some good, some bad. It's that latter category of experience, the "gotchas," that leave us with bumps, bruises, and battle scars. The aim of this article is to help you avoid some of those painful mistakes. Here, you will find an overview of the most common pitfalls you are likely to encounter when dealing with ABAP fundamental operations and data types, like simple arithmetic and conversions, predefined ABAP data types, and flat structures. All in all, we will cover 10 prevalent pitfalls:

1. Assuming ABAP determines arithmetic operation types in the same way that C, C++, and Java do.
2. Ambivalent fix point and floating point behavior of ABAP type p.
3. Wrong rounding in fields of type f.
4. A rare but nasty "gotcha" when working with numeric literals.
5. Incurring needless conversion costs when using text literals with numeric values.
6. Mistakenly using numerical literals when you should be using text literals.
7. Misusing in-place data declarations for initializations.
8. Declaring data in wrong scopes.
9. Using flat structures with numeric elements as text fields.
10. Using flat structures with numeric elements in anonymous containers.

Where did this list of top 10 pitfalls come from? These ABAP developer trouble spots represent the most frequently reported problems logged by SAP's help desk concerning ABAP fundamental operations and data types over the past three years. Perhaps you recognize one or two![1]

Pitfall #1: Assuming ABAP Determines Arithmetic Operation Types in the Same Way That C, C++, and Java Do

Arithmetic calculations in ABAP are commonly realized by using the ABAP keyword `COMPUTE` with the syntax:

```
COMPUTE result = arithmetic expression.
```

or simply by omitting the keyword `COMPUTE` itself and writing:

```
result = arithmetic expression.
```

where `result` is the result field and `arithmetic expression` is an arbitrary arithmetic expression, as exemplified below:

```
r = 1 / 3.
```

Seems simple and predictable enough. And it is, except for those who bring to the table preconceived notions from other programming languages. If you think that ABAP follows the same strategy as C, C++, or Java to determine the type in which arithmetic operations are executed, you will encounter the first of our common ABAP pitfalls!

In C, C++, and Java, arithmetic expressions are executed by following a "bottom-up" strategy. This means that for every single operation, the arithmetic type is determined with respect to the involved operand types. After executing the elementary operations, the results then are propagated and serve as operands for the next level of operations. With this strategy, the calculation shown at the top of **Figure 1** is executed as follows: First, the elementary expression `1/3` is executed, where the two integer operands lead to a result of type `int`. The result is `0`. This result is then assigned to the result field of type `double`, which is filled with `0`.

Now turn your attention to the calculation shown at the bottom of Figure 1. Unlike the C, C++, or Java behavior shown at the top, the ABAP processor analyzes the whole expression with respect to all involved operands, operators, and built-in functions to determine the arithmetic type for the *whole statement*. This is done before executing the arithmetic statement.

To simplify a little bit, we can say that ABAP analyzes an arithmetic expression in the following hierarchy:

1. If all involved fields — namely all operands and the result field — have data type i (integer), ABAP performs a type i arithmetic.

Figure 1 C, C++, or Java vs. ABAP Execution of Arithmetic Calculation

C, C++, or Java Execution:

```
double d;
d = 1 / 3;      // d = 0  !!!
```

ABAP Execution:

```
DATA r TYPE f.
r = 1 / 3.      "  r = 0.333…
```

[1] The program examples in this article are written for Basis Release 4.6, but the essence of the example traps and pitfalls holds for earlier releases as well.

Figure 2 Overview of ABAP Numeric Types

Type	Type Kind	Internal Arithmetic Type	Accuracy	Value Range
f	binary floating point	binary floating point	at least 16 decimal places	0, ± 2.22507385850720E - 308 ... ±1.79769313486232E + 308
p	decimal fix point	decimal floating point*	31 decimal places	0, $\pm 10^{-14}$ ... $\pm 10^{31} - 1$**
i	integer	integer	at least 9 decimal places	-2147483648 ... +2147483647

* With an exponent range from -255 up to 0 (i.e., intermediate results can have values from $\pm 10^{-255}$ up to $\pm 10^{31} - 1$).

** Maximum/minimum values over all possible types. For the distinct types the value range is smaller.

2. If all involved fields have data type i, but at least one operand is of data type p (packed numbers), ABAP performs a type p arithmetic.

3. If at least one field has data type f (floating point), ABAP performs a type f arithmetic.[2]

With this strategy, ABAP executes the arithmetic calculation shown at the bottom of Figure 1 as follows: First the whole arithmetic expression is analyzed. This is usually done at compile time, except if one operand has a generic type. After the analysis, the arithmetic type for the whole statement is set. In our case, the result field `r` is found to be of data type f, and so the whole calculation is performed using type f arithmetic. This means that every operand is first converted into data type f and then the calculation is done. Therefore, the result is `0.333333333333333`.

Resolving arithmetic expressions in this way means that in nearly all cases you get what you would intuitively expect.

[2] A number of built-in functions like `sin`, `cos`, etc. have type f return values and lead to type f arithmetic, much like the `**` operator.

Pitfall #2: Ambivalent Fix Point and Floating Point Behavior of ABAP Type p

Using data type p (packed numbers) has certain advantages over data type f (floating point):

- An accuracy of up to 31 decimal places (see **Figure 2**).
- It comes with a true decimal arithmetic that is similar to the way you would do your calculations with paper and a pencil.
- Rounding takes place as decimal rounding.

Given these properties, data type p is typically used for sizes, lengths, weights, and sums of money.

The basic syntax for declaring a data object of type p is:

```
DATA p(len) TYPE p DECIMALS d.
```

When declaring a field of data type p, you can specify the length, `len`, in bytes from 1 to 16 (default is 8) and the number of decimal places, `DECIMALS`,

Figure 3 Calculating a Fraction Using Fix Point Arithmetic

```
DATA: value(3)   TYPE p DECIMALS 2,
      result(3)  type p DECIMALS 2,
      percent(3) type p DECIMALS 2.

value   = 100.
percent = '55.55'.

* Calculation 1
result  = value * ( percent / 100 ). " result = 55.55

* Calculation 2
percent = percent / 100.
result  = value * percent.           " result = 56.00 ???
```

from 0 to 14 (default is 0). The total number of decimal digits for the resulting packed number is then calculated from the given length in bytes, `len`, according to the formula:

$$d = 2 \times len - 1$$

Each byte contains 2 decimal numbers except the last byte, where a half-byte is reserved for the encoded sign. For example, a field of data type p with 5 decimal digits, including 2 decimal places, can be declared by:

```
DATA my_p(3) TYPE p DECIMALS 2.
```

The number of decimal places for a field of data type p is a property of the specified type and not part of the data that is stored in the data object. So data type p is a **fix point number** type, meaning that the decimal point is fixed at a defined position.

This is in contrast to **floating point numbers** like data type f, where the decimal point position value *is* part of the field value. As a consequence, in floating point numbers, the field value contains two parts of information:

- The value itself, represented by a certain number of decimal digits, for instance.
- An exponent, which specifies how many of the decimal digits are decimal places and how many zeros have to be added to the left or to the right to get the field value.

The difference between floating point and fix point arithmetic has led many misinformed ABAP programmers astray! Take a look at **Figure 3**. A fraction of 55.55 percent of an initial value of 100 is calculated by using fix point numbers of data type p with 2 decimals.

In the first calculation, the result is computed in a single expression by dividing the percent value of `55.55` by `100` and multiplying this with the initial value to get the percent fraction. We receive the exact result of `55.55`.

In the second calculation, the division of the percent value of `55.55` by `100` is done in one step and then the multiplication is performed in another. Surprisingly, we receive a rounded result of `56.00`.

What's going on here? Well, the example program in Figure 3 has been set up to operate with data in the fields `value`, `result`, and `percent` with 5 decimal digits, including 2 decimal places, for each of these three fields. This means that any value stored in any one of the

fields cannot exceed 999.99 and is internally rounded to 2 decimal places.

In the second calculation, we find a misuse of field `percent` as a storage place for an intermediate result. The result of the division of 55.55 by 100 yields 0.5555 and is stored in field `percent`, where rounding to a value of 0.56 takes place, hence the second calculation step produces a value of `56.00` for `result`.

The interesting question that arises is, "Why does the first calculation yield an accurate result at all?" The answer lies in the way that ABAP performs type p arithmetic:

- An arithmetic expression of type p is executed with an internal accuracy of 31 decimal digits and with a decimal floating point semantic. This means that inside arithmetic expressions, intermediate results of data type p behave like decimal floating point numbers due to a variable position of the decimal point. Note that this is different from user-defined fields of type p, where the decimal point is fixed!

- Compared to the standard type f arithmetic, the decimal floating point semantic never omits positions before the decimal point. If the accuracy is not sufficient, only decimal places can be lost by rounding. Instead of omitting positions before the decimal point, the system raises a catchable overflow exception instead.[3]

The computation of the first calculation is performed in a single statement, using an internal precision of 31 significant decimal digits in conjunction with the decimal floating point behavior. The intermediate result of 0.5555 is treated accurately, without any rounding steps.

[3] The behavior is actually a bit more complex. Starting with Basis Release 4.6, when an overflow exception is raised because an intermediate result becomes larger than $10^{31} - 1$, the whole arithmetic expression is executed in a second attempt using an internal accuracy of 63 decimal digits or a maximum value of $10^{63} - 1$ for intermediate results.

When you use data objects of type p for intermediate results, be aware that it is not the system but you who steers the accuracy of the complete calculation by specifying the length and the number of decimal places of the fields. To avoid the trap that single-step calculations may lead to results other than storing intermediate results in fields of data type p, and to avoid inadequate value rounding, use fields of type p only for values of comparable accuracy.

Pitfall #3: Wrong Rounding in Fields of Type f

ABAP developers often use data type f for calculations, because the type f arithmetic is performed much more quickly than type p arithmetic. This is because calculations with type f, much like type i, can be done directly by the fast machine instructions of the underlying hardware. Type p arithmetic, on the other hand, is based on an SAP-specific software emulation, since for nearly all platforms no hardware support is available.

We have observed that when the calculation part of an ABAP program begins to significantly impact the program's performance, some developers are tempted to switch from type p to type f. This switch should not be made too hastily! Before you go this route, you must give careful consideration to two things:

1. The accuracy of your calculation decreases. You go from a possible 31 decimal digits for type p down to typically 16 decimal digits for type f.

2. Type f arithmetic is *not* decimal arithmetic. Multiplication, for instance, using powers of 10 (positive or negative), is not an exact operation and rounding to a certain number of decimal places may be incorrect. Herein lies a rather treacherous trap, because nearly every rounding operation with floating point types may be affected.

Figure 4 Assigning Different Values to a Type f "Float" Field

```
DATA float TYPE f.

float = '1.5'.
WRITE / float DECIMALS 0. "Output : 2E+00

float = '1.05'.
WRITE / float DECIMALS 1. "Output : 1.1E+00

float = '1.005'.
WRITE / float DECIMALS 2. "Output : 1.00E+00 ???
```

To better understand this trap, turn your attention to **Figure 4**, where you can see that we are assigning different values to the field `float` of data type f. We then write the contents of the field to a list by rounding it to different decimal places by using the addition `DECIMALS` of the `WRITE` statement.

Rounding the values `1.5` and `1.05` to `0` decimal places, or `1` decimal place, results in the expected output of `2E+00` and `1.1E+00`, respectively. But rounding the value of `1.005` to `2` decimal places results in an output of `1.00E+00` rather than `1.01E+00`, which is what one would have expected from the rules of decimal rounding.

This phenomenon is by no means unique to ABAP. As long as the underlying machine type of a data type is a binary floating point type, you will encounter rounding phenomena like this in other programming languages like C or C++.

Data type f is built on the 8-byte floating point numbers of the underlying platforms. These numbers are usually encoded in a sign "s" of 1 bit, an exponent "e" of 11 bits, and a mantissa "m" of 52 bits. This leads us to a decimal accuracy of at least 16 decimal digits and an exponent range from about 10^{-308} to 10^{+308}. In contrast to the usual decimal fraction representation, floating point numbers are internally encoded in a dual fraction representation, and therefore the exponent part "e" contributes with an expression to the basis of 2 to the resulting value. For 8-byte floating point numbers, this expression is 2^{e-1023}.

The rounding problem results from the fact that the decimal fraction representation and the internal dual fraction representation cannot always express each other's value, because they have a finite accuracy.

Figure 5 shows a comparison of the numbers from our example in decimal and dual fraction representation. Since the decimal value of 1.5 can be represented through a simple expression to the basis of 2 (namely $1.5 = 1 \times 2^0 + 1 \times 2^{-1}$), the dual fraction and the decimal fraction representations are equal. For the next value of the example, 1.05, the dual fraction and the decimal fraction representations are not equal, because the accuracy of the 8-byte floating point number is not sufficient. So the system chooses the closest value nearby, which corresponds to a value of 1.0500000000000003 in the decimal representation and is slightly bigger than 1.05. Fortunately, this difference vanishes in the final rounding.

This, however, does not work in the third example with a value of 1.005. Here, the dual fraction and the decimal fraction representations are not equal. In this case, the closest dual fraction value nearby is slightly smaller, namely 1.0049999999999998. Therefore, in the final rounding, the result is 1.00 rather than 1.01.

Figure 5 A Comparison of Decimal and Dual Fraction Representation

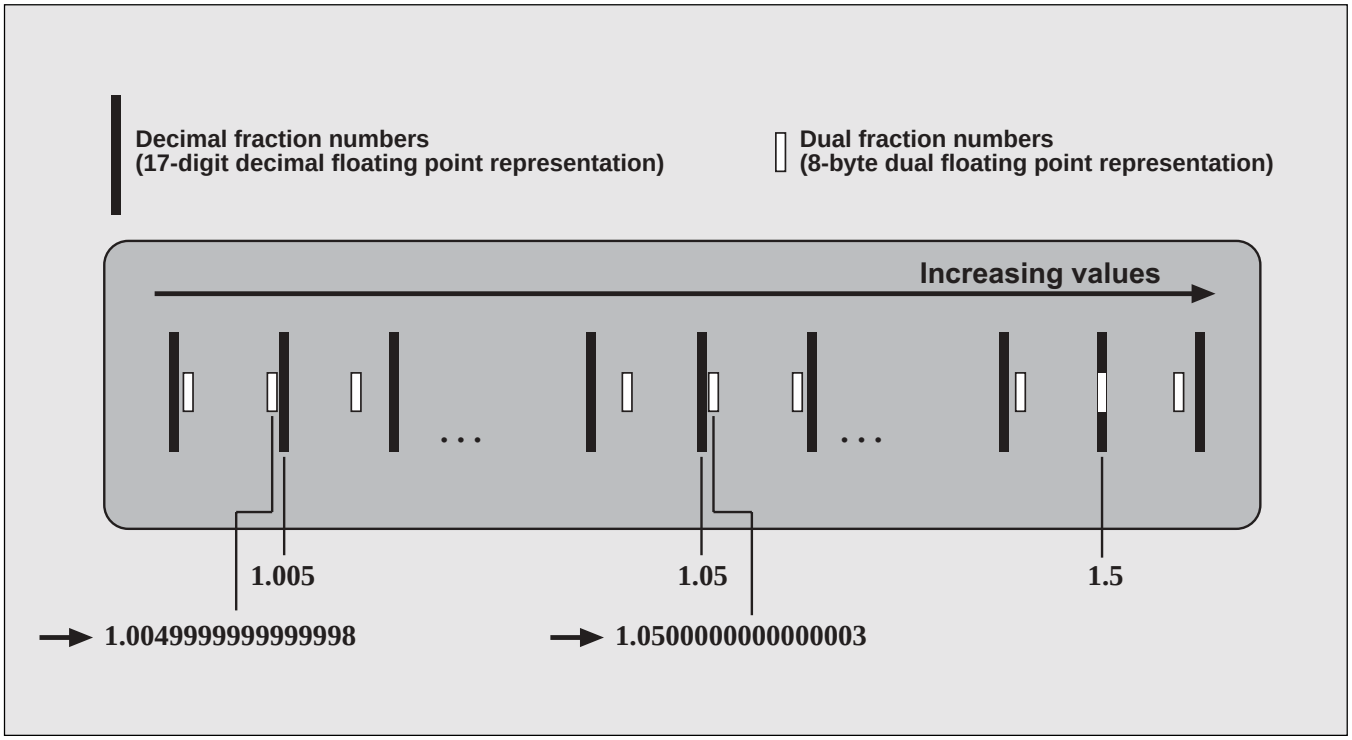


Figure 6 Avoiding Incorrect Results by Rounding Decimals in Two Steps

Value in Dual Fraction	Rounding to 15 Places	Result with 2 Places
1.5000000000000000	1.50000000000000	1.50E+00
1.0500000000000003	1.05000000000000	1.05E+00
1.0049999999999998	1.00500000000000	1.01E+00

We could point to lots of examples like our value 1.005, and you probably could, too. Unfortunately, these situations cannot be foreseen. There is no way to know ahead of time if the conversion from the decimal fraction to the internal dual fraction representation will result in a value that is smaller than the exact value and will therefore produce the wrong decimal rounding.

The only way to circumvent the trap of wrong rounding in fields of type f is to round in two steps instead of in a single step. In the first step, you round to 15 decimal digits, for instance — that is, one digit less than the accuracy of the 8-byte floating point representation translated to the decimal system. Then you round the resulting value to the final decimal digits in a second step to get the expected result. **Figure 6** shows how this circumvents the trap for the values of the previous example.

Figure 7 Using the Static Method ROUND_F_TO_15_DECS

```
DATA float TYPE f.
CLASS cl_abap_math DEFINITION LOAD.

float = '1.5'.
float = cl_abap_math=>round_f_to_15_decs( float ).
WRITE / float DECIMALS 2.      "Output : 1.50E+00

float = '1.05'.
float = cl_abap_math=>round_f_to_15_decs( float ).
WRITE / float DECIMALS 2.      "Output : 1.05E+00

float = '1.005'.
float = cl_abap_math=>round_f_to_15_decs( float ).
WRITE / float DECIMALS 2.      "Output : 1.01E+00
```

In order to round to 15 decimal digits, you can use the static method ROUND_F_TO_15_DECS of class CL_ABAP_MATH from the ABAP class library. **Figure 7** shows an example.

The most important thing to remember when using this method is that the method rounds the field of data type f to 15 decimal digits and leaves f in its binary floating point format, and the following coding can still operate with a field of type f. As before, the second rounding step is done by using the addition DECIMALS of the WRITE statement. Such rounding of floating point numbers by using features of presentation statements is a common practice.

You should always keep in mind that the use of method ROUND_F_TO_15_DECS means a loss of accuracy for the floating point value from 16 to 15 decimal digits. Therefore, you should use the method only if you want to avoid confusing floating point rounding effects in the data presentation.

> **✓ *Tip***
>
> *To protect your application coding from a loss of accuracy, encapsulate all rounding that is done for presentation purposes in the coding that is responsible for the presentation.*

Pitfall #4: A Rare but Nasty "Gotcha" When Working with Numeric Literals

The ABAP type concept puts 10 built-in elementary data types — namely c, n, d, t, x, i, f, p, string, and xstring — at your disposal. Using these types, you can declare (e.g., with the DATA statement) data objects of 10 different elementary types. Such data objects have names and are called *named data objects*. In an ABAP program, you can also use data objects without names: so-called *literals*. There are two types of literals: text literals and numeric literals.[4]

To define text literals in your program code, you enclose characters in single quotes, as for example:

```
'Hello World' and '1234'
```

ABAP handles all text literals like data objects of data type c.

Numeric literals consist of the digits 0-9. You can define numeric literals with up to 31 decimal digits, and they may be prefixed by a minus sign (-). Examples are:

```
12345 and -9876
```

[4] From Release 6.10 of the SAP Web Application Server on, there will be a third kind of literal, namely string literals with data type string.

Figure 8 An Unexpected Result in a Sequence of Numeric Literals

```
DATA i TYPE i.

i = 1 / 3  * 2.                    " result 0
i = 11 / 33  * 2.                  " result 0
i = 111 / 333  * 2.                " result 0
  ...
i = 111111111 / 333333333  * 2.    " result 0
i = 1111111111 / 3333333333  * 2.  " result 1 ???
```

ABAP handles numeric literals like data objects of either type i or type p. Literals with values between -2147483648 (or -2^{31}) and 2147483647 (or $2^{31} - 1$) are handled as type i. All other literals with values up to $\pm 10^{31} - 1$ are handled as type p. If you type more than 31 decimal digits, you get a syntax error.

Determining the data type of numeric literals according to their value is done for optimization reasons. In all cases, where type i can be used, it is significantly faster than the corresponding type p treatment. So the ABAP compiler uses type i whenever it can. In *nearly* all cases this makes no difference to the coding, since conversion and arithmetic of type p without decimal places and type i are so similar. *But in rare cases it does make a difference and sets up a nasty little trap for the unsuspecting ABAP programmer!*

To illustrate this trap, let us have a look at the coding in **Figure 8**. In a sequence of arithmetic expressions with numeric literals, the last expression gives an unexpected result.

All numeric literals used in this example are treated as data objects of type i except the highlighted literal `3333333333`. Its value is larger than $2^{31} - 1$, and therefore is treated as a data object of type p. As a result, all calculations are completely executed in type i arithmetic, except the last, where type p arithmetic is used.

Because of the decimal floating point semantic described previously, the intermediate result of the division in the last calculation is not 0 but 0.333... Therefore, unlike in all the other statements, the subsequent multiplication by 2 yields an intermediate result of 0.666... and the final off-rounded result is `1`.

The use of numeric literals that are bigger than the maximal values for type i, as well as a situation where the difference between type i and type p arithmetic leads to different results, is a rare occurrence. Although this is not as common a pitfall as others in this article, we list it here, since if somebody is trapped here, he will have a hard time explaining the behavior. This is because using type i instead of type p is a silent optimization of the ABAP runtime at an unexpected threshold.

To avoid this snag, keep an eye on the number of decimal digits in numeric decimals when you use them in places where you would expect type i arithmetic.

Pitfall # 5: Incurring Needless Conversion Costs When Using Text Literals with Numeric Values

Since numeric literals in ABAP are only integer numeric literals, for decimal fraction values or exponent expressions you must use text literals instead, as for example:

`'3.1415926'` or `'6.023E23'`

Figure 9 Runtime Measurement of a Text Literal and a Numeric Literal Calculation

```
DATA: t1 TYPE i,
      t2 TYPE i,
      r1 TYPE i,
      r2 TYPE i,
      fr TYPE p DECIMALS 2.
DATA result TYPE f.

DO 1000 TIMES.

  GET RUN TIME FIELD t1.
  result = '123' / '456'.  " calculation 1
  GET RUN TIME FIELD t2.
  r1 = r1 + t2 - t1.

  GET RUN TIME FIELD t1.
  result = 123 / 456.      " calculation 2
  GET RUN TIME FIELD t2.
  r2 = r2 + t2 - t1.

ENDDO.

fr = r1 / r2.              " about 10 !!!
```

In arithmetic expressions, such text literals are converted to the arithmetic type of the expression. You might also tend to use text literals for integer values instead of numeric literals. But this leads to unnecessary conversion costs.

The example in **Figure 9** shows the runtime measurement of two almost identical calculations. The only difference is that the first calculation uses two text literals instead of numeric literals. But that increases the runtime consumption by almost a factor of 10! In both cases, the arithmetic type is f, since `result` is of type f. But in the first example, the conversion from the text literals (type c) to type f dominates the runtime of the calculation, since it is one of the most expensive conversions. In the second calculation, the runtime consumption of the conversion from numeric literals of type i to arguments of type f is of the same order of magnitude as the division itself.

The lesson we hope to impart here is that you should always avoid single quotes around integer values in arithmetic expressions. But since both calculations yield the same result, the example in Figure 9 is by no means a major pitfall. For mere value assignments, whether you should use single quotes depends on the data type of the target field. This is clearly shown in our next example.

Pitfall #6: Mistakenly Using Numeric Literals When You Should Be Using Text Literals

In **Figure 10**, initializations are made for field `date` of type d and for field `time` of type t. In both cases, the initialization seems to be correct and many programmers expect the field `date` to contain

Figure 10 Initializations for a Type d "Date" Field and a Type t "Time" Field

```
DATA date TYPE d VALUE 20001231.
DATA time TYPE t VALUE 235959.

WRITE /(10) date.         " output 0000/00/00 ???
WRITE /(8)  time.         " output 17:32:39   ???
```

`20001231` (the 31st of December 2000) and the field `time` to be filled with `235959` (a time of 23:59:59) after the initialization. But both fields contain unexpected values afterward.

Both types d and t are character-like types, like data type c, where each position in a field takes up the space for the code of one character. The decoding format for type d is YYYYMMDD, where YYYY specifies the year, MM the month, and DD the day. For type t, the decoding format is HHMMSS, where HH is the hour, MM the minute, and SS the second.

Since YYYYMMDD and HHMMSS are text formats, the initializations of data type d and t should always be performed with text literals rather than with numeric literals. And indeed, transforming the numeric literals to text literals by adding four single quotes to the coding in Figure 10 will lead to the intended behavior.

Numeric literals for initializations of types d and t are inadequate because it results in a forced conversion. In order to understand the unexpected output values of the example in Figure 10, let us look briefly at the conversion rules for types d and t:

- In arithmetic expressions and in conversions from or to numeric values, dates in fields of type d are converted to the number of days since the first of January in the year 0001. A field of type d is set to the initial value, if it should be filled from a numeric value that represents an invalid date.

- Numeric values in conjunction with type t are treated as the number of seconds since midnight (note that midnight is represented by 00:00:00). If the value exceeds 86,400, which is the number of seconds of one day, the considered numerical is changed to the result of the value modulo 86,400.

During the initialization of field `date` in our example, the numeric literal is interpreted as a number of days since 0001/01/01, namely 20,001,231 — that is, a date in the 54th millennium. Since the maximum possible date for type d is the 31st of December in the year 9999, the value of field `date` is set to its initial value, `0000/00/00`.

During the initialization of field `time`, the numeric literal `235959` is interpreted as the number of seconds since midnight, which is equivalent to two days and a time of `17:32:39` at the third day.

Remember, there are 10 built-in ABAP data types, but only two literals. That is, most built-in types do not have a corresponding literal and you can only use text literals (data type c) or numeric literals (data type i or p). In conjunction with character-like data types such as c, string, or n, text literals should be used.[5] In conjunction with the numeric data types i, p, and f, you should use numeric literals whenever possible.

Data types d and t — and also x or xstring, but we won't go into detail here — show two sides of a coin. On the one hand, they are character-like types, a fact that is exploited quite often. For example, to get only the month information out of a date you can simply write `date+4(2)` using the offset-and-length

[5] From Release 6.10 of the SAP Web Application Server on, you should use string literals for strings, of course.

technique that is allowed for character-like fields. On the other hand, an important part of the functionality that comes with data types d and t is based on their internal numeric representation — that is, the number of days since the 1st of January 0001 and the number of seconds since midnight, respectively. Common examples include adding a number of days to a specific date or determining a time period by subtracting two times. Therefore, when you work with data types d and t, the use of text or numeric literals must be scrutinized. If types d or t are to be used in their numeric representation, numeric literals should be used, otherwise use text literals.

Pitfall #7: Misusing In-Place Data Declarations for Initializations

Coming from the Java or C++ scene, you may be used to declaring and instantiating objects in one step and in-place, for example:

```
Object Obj = new Object();
```

The variable `Obj` is then visible from this statement on in the current scope and is instantiated with the given statement in-place. The advantage of this technique is that you are able to circumvent the potentially high costs of instantiating huge objects where you don't need them. With native object type declarations and initializations, like

```
int i = 123;
```

C++ or Java programmers often proceed in the same way.

The typical statement for declaring data objects in ABAP is `DATA`. The ABAP equivalent for the previous declaration is:

```
DATA i TYPE i VALUE 123.
```

Much like in C++ or Java, the ABAP syntax allows you to declare data objects at almost every position of a program. This means you might write the `DATA` statement just before the statements where you need the data objects. But transferring the in-place data declaration and initialization technique for native object types from C++ or Java to ABAP will lead you into a nasty trap.

Figure 11 shows an example inside a loop where an increasing number, namely the loop index `sy-index`, is added to a variable `value`. The variable `value` is declared and initialized with a `DATA` statement inside the `DO` block. At first glance, one might expect that value will be set to the values 11, 12, 13, etc., but the result is rather different.

Unlike in C++ or Java, in ABAP there is no possibility to open a local scope by declaring data in control blocks, as for example `IF` … `ENDIF` or `DO` … `ENDDO`. In ABAP there are exactly three scopes where data can be declared[6]:

- Within the scope of a procedure (methods, functions, or subroutines)
- Within the scope of a class
- Within the scope of the ABAP program itself

Procedures have the most local data. Data declared in the scope of the ABAP program tend to be global data.

The variable `value` in the example program has the scope of method `m1`. It lives during the execution of the method and is initialized in the moment, when the method is called. As a consequence, the initialization done by the `DATA` statement is not repeated in every loop step of the `DO` statement. Hence, at the beginning of the second loop step, the content of `value` is 11, not 10 as you might expect from the sequence of the coding. You can place declaration statements with initializations anywhere inside a scope, with the effect that the data is visible only behind its declaration. But the initializations are always done at the beginning of the scope.

[6] See Horst Keller's article "ABAP Programming — An Integrated Overview," which appeared in the January/February 2000 issue of the *SAP Professional Journal*.

Figure 11 Adding an Increasing Number to a Variable "Value"

```
METHOD m1.
   ...
   DO 10 TIMES.
     DATA value TYPE i VALUE 10.
     ...
     value = value + sy-index. " value = 11, 13, 16, ...
     ...
   ENDDO.
 ENDMETHOD.
```

Figure 12 Assigning Start Values In-Place

```
METHOD m1.
   DATA value TYPE i.
   ...
   DO 10 TIMES.
     value = 10.
     ...
     value = value + sy-index. " value = 11, 12, 13, ...
     ...
   ENDDO.
 ENDMETHOD.
```

Since data declarations and their initializations are executed at the beginning of the current scope, you should also write the DATA statement at this position. For procedures this is directly behind the defining statements METHOD, FUNCTION, or FORM, while in ABAP programs you should declare data only in the global data declaration part between the introducing statement (PROGRAM, REPORT, FUNCTION-POOL, etc.) and the first processing block. In classes, you can declare data only in the declaration part introduced by CLASS c DEFINITION anyway.

If you want to carry out an initialization at the beginning of a statement block, as for example our DO loop, you must program a respective value assignment. **Figure 12** shows the corrected example.

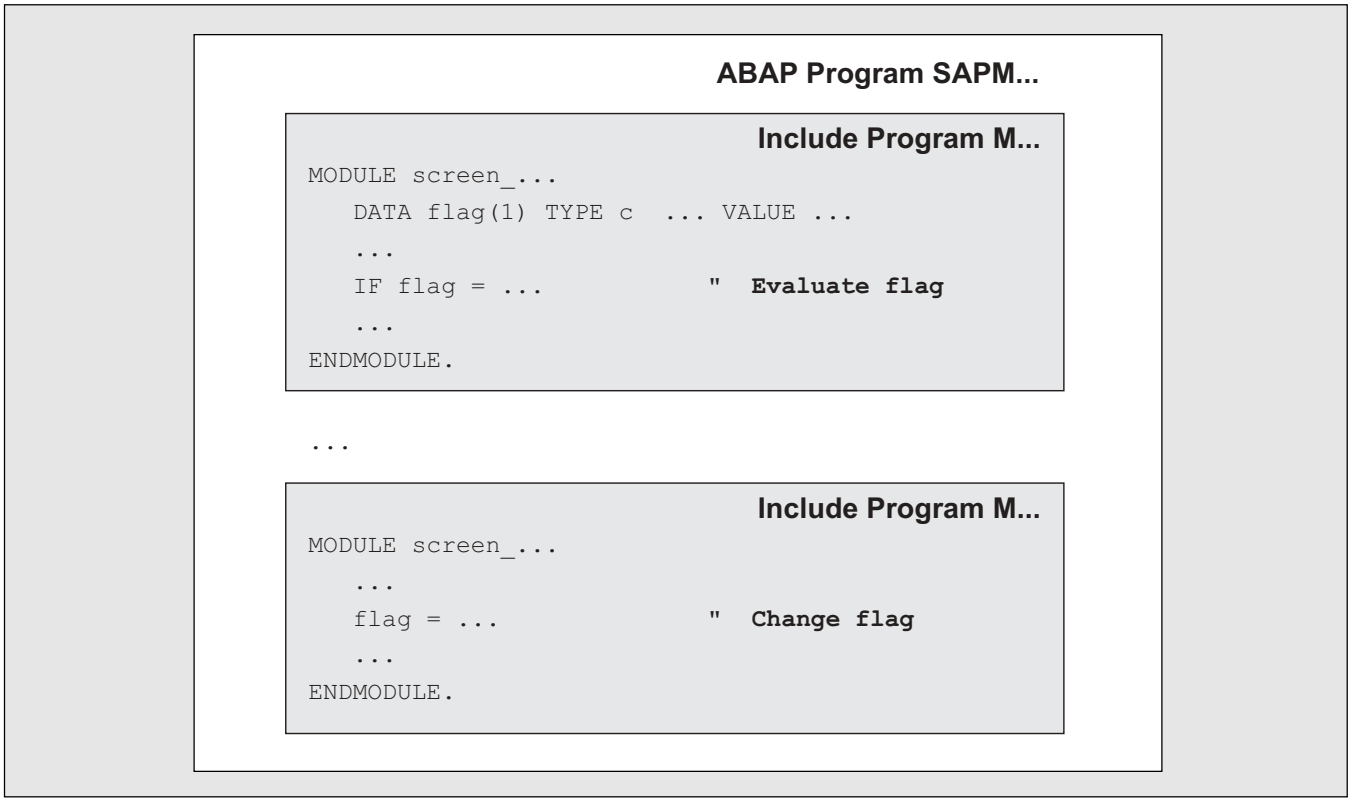
The fact that initializations generated by declaration statements are executed at the beginning of the current scope leads us directly to the next trap, namely data declarations in wrong scopes.

✓ *Tip*

If you not only want to assign start values to existing data during the sequence of a processing block, but to instantiate data objects dynamically at special positions, you can use the in-place statement CREATE DATA that is available in ABAP since Basis Release 4.6. While the instantiation of data objects declared with the DATA statement is executed implicitly at the beginning of their scope, with CREATE DATA you steer the instantiation explicitly. Therefore, you can create large and complex data only when they are needed in order to optimize the memory consumption of your program.[7]

[7] Deletion of a dynamically created data object is done automatically by ABAP's built-in garbage collector.

Figure 13 A Classical Module Pool Serves As a Container for Several Dialog Modules



Pitfall #8: Declaring Data in Wrong Scopes

In the preceding section, you learned that there are exactly three scopes where data can be declared in ABAP: procedures, classes, and the program itself. Now compare that with the program in **Figure 13**.

The ABAP program in Figure 13 presents a classical module pool that serves as a container for several dialog modules.[8] In Figure 13, a data object `flag` is declared and initialized within a dialog module. The `DATA` statement is neatly written at the beginning of the dialog module, and therefore many ABAP programmers tend to believe that `flag` is local to the dialog module and is initialized every time the module is called from the screen flow logic. Well, they do so until — maybe years after releasing the program — an error occurs and the dialog module does not behave as expected.

In ABAP, only procedures (e.g., methods, functions, or subroutines) can have local data. So you need to be mindful that not all processing blocks can have local data. This is especially true for dialog modules defined between `MODULE` and `ENDMODULE` and for event blocks defined after event keywords, as, for example, `START-OF-SELECTION` or `AT LINE-SELECTION`. All data declarations in dialog modules or event blocks have the scope of the respective ABAP program and are part of its global data.[9]

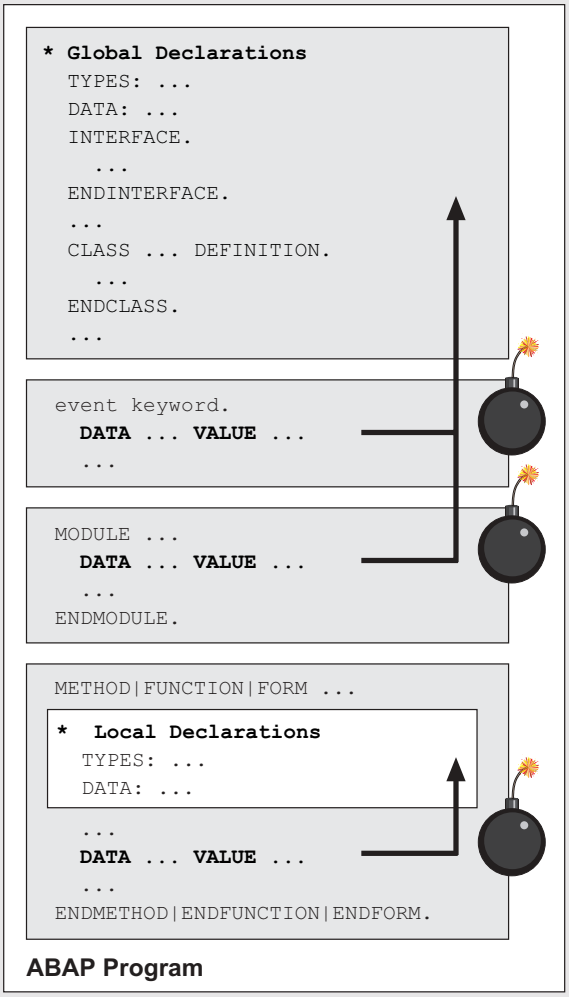
[8] Typically, the dialog modules are defined in several Include programs. Note that the trap shown here is independent from the program's organization in Include programs. Furthermore, it might also occur in other types of programs such as executables or function pools.

[9] Unfortunately, there is no rule without exceptions. Here, the exception is that event blocks defined by `AT SELECTION-SCREEN` and `GET` are internally implemented as procedures, and therefore can have local data. In order to avoid confusion, you should not exploit this fact, but work with them as with normal event blocks without local data.

All statements in all processing blocks behind the data declaration know and can work with that data.

Tip

In-place data declarations and data declarations in event blocks or dialog modules are principally dangerous. Therefore, place all data declarations at those program positions where they really belong — that is, local data at the beginning of the procedure and global data in the global declaration part. If you want to encapsulate data, use procedures and call them from event blocks and dialog modules.



Imagine that in our simple example (Figure 13) the second dialog module is programmed in another Include program, by another developer and in a later stage of the project than the first dialog module. The second module knows `flag` implicitly and changes it according to its own needs. Especially in large dialog programs with a lot of screens and many different application scenarios, extensive testing and even the productive use of such programs might not lead directly to an error. But someday in some situation, it is possible that the second dialog module will change the data object before the first dialog module is called from the screen flow logic. As a result, the first dialog module initialization might be wrong!

In order to avoid this trap, be sure to declare global program data only in the global declaration part of the program. In large programs, the program coding of the global declaration is normally written into the TOP Include of the program. Never declare data in processing blocks that cannot have local data.

✓ Tip

Use as little global data as possible. As a rule of thumb, encapsulate your data first in procedures and put only the data that is needed in several procedures in a common class (poor man's object-orientation!). With the exceptions of transferring data from and to screens or logical databases, there is absolutely no need for global data in ABAP programs at all.

Pitfall #9: Using Flat Structures with Numeric Elements As Text Fields

You know that in ABAP elementary data types can be combined to complex data types, namely structured types and table types. For the structured types, we must distinguish flat structures and deep structures. The outstanding feature of deep structures is that they contain at least one deep component that is a string,

Figure 14 Deep Structures Contain at Least One String, Internal Table, or Reference Variable

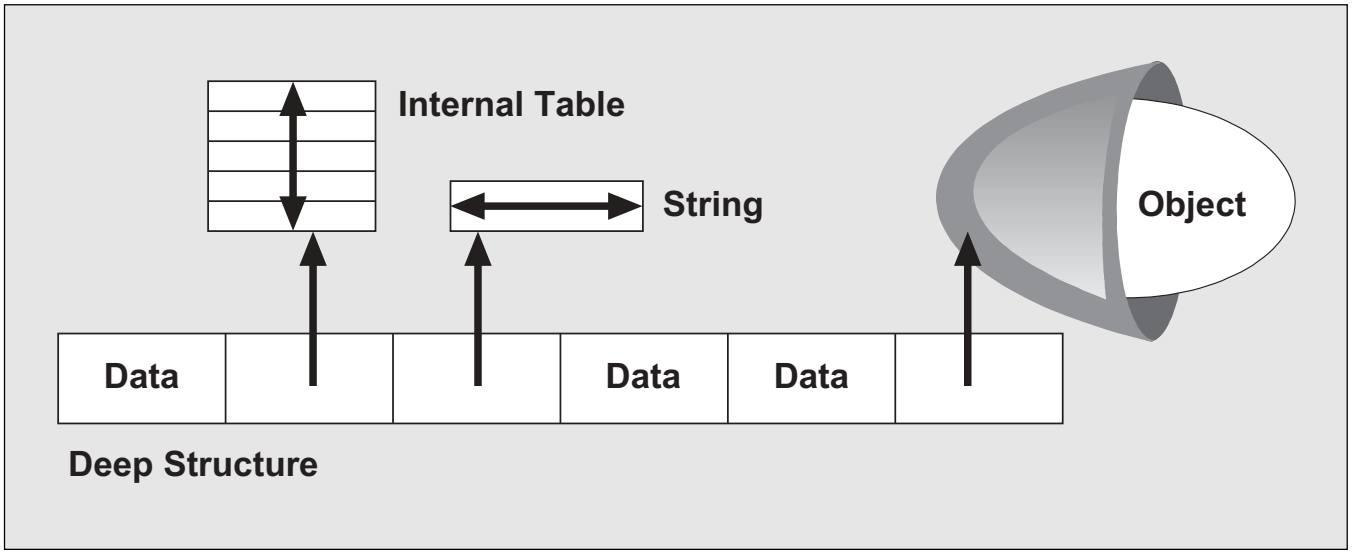


Figure 15 Operations Affecting Deep Structure Technical References Are Forbidden

```
FIELD-SYMBOLS <fs> TYPE ANY.

DATA: BEGIN OF deep_struc,
            str TYPE string,
      END OF deep_struc.

ASSIGN deep_struc TO <fs> CASTING TYPE c. " Syntax error
```

an internal table, or a reference variable pointing to an object. The data for such deep components cannot be stored in-place inside the structure itself because of their dynamic nature. Therefore, for each deep component only a technical reference can be found inside the structure, pointing to the outsourced data (see **Figure 14**).

When you work with deep structures, all operations are forbidden that could potentially destroy their technical reference information. Even brute force casting operations that might affect the technical references are not allowed. The example code in **Figure 15** will, therefore, throw off a syntax error.

This is not the case with flat structures, where all the data is in-place. A flat structure represents a data area of a specific length, where structure components are all stored one behind the other. The memory in the data area is fully accessible by the program and you can reinterpret and manipulate it directly by using different casting techniques. (Note that even nested structures are flat, as long as all substructures contain only flat components.)

The most common casting technique used for flat structures is the interpretation of the complete structure as a field of type c with the length of the structure. In order to do this, you simply specify the name of the structure, where elementary fields are

expected. Then ABAP interprets the whole structure as one field of type c in the structure's length. This is often not realized to be a type cast, but in fact it is. It can be used in a variety of ABAP statements, as can be seen from the code example in **Figure 16**.

Figure 16 Casting the Flat Structure Type by Interpreting It As a Type c Field

```
DATA: BEGIN OF flat_struc,
        date TYPE d,
        time TYPE t,
      END OF flat_struc.

DATA  date TYPE d.

WRITE / flat_struc.
date = flat_struc(8).
flat_struc = 'Hello, World !!'.
```

The flat structure `flat_struc` in Figure 16 consists of two flat components, `date` of type d and `time` of type t. The `WRITE` statement takes `flat_struc` as a field of type c in the complete structure length and prints it out as a sequence of 14 characters. In the next statement only the first 8 characters of `flat_struc` are selected by the specification of offset and length and stored in a field `date` of elementary type d. And finally, the last statement stores the character literal `'Hello, World !!'` in `flat_struc`, implicitly filling the components `flat_struc-date` and `flat_struc-time` with values of `'Hello, w'` and `'orld !!'`, respectively.

In many cases, this kind of implicit type cast is helpful. Especially if all components of the involved flat structures have character-like types (c, d, n, or t), this technique is more a kind of wrapping mechanism than a real casting technique. However, if one or more components of the flat structures have numeric types like type i, f, or p, casting the whole structure to type c and assigning a text value, for instance, will result in at least nonsense data for the numeric fields. In the worst case, invalid data and even runtime errors are possible.

For example, the code in **Figure 17** terminates with an exception at runtime. The system does not find data in the field `flat_struc-number` that can be interpreted as a numerical value.

To avoid such invalid numeric data resulting from implicit flat structure type casts, you have to take care that the raw byte offsets of the numeric components as well as the numeric types in the involved flat structures fit to one another. For example, a flat structure with three components, two leading fields of type c length 5, and a supplement field of type f, can without serious problems be assigned to a flat structure with two components, one leading field of type c length 10, and a field of type f.[10]

[10] From Release 6.10 of the SAP Web Application Server on, ABAP supports Unicode. In a Unicode system, only such valid constellations are accepted by the compiler. For invalid constellations, where numeric field values would be destroyed, a syntax error occurs. This *safe mode* is available for non-Unicode systems as well, if you simply enable the new optional program attribute *Unicode checks active*.

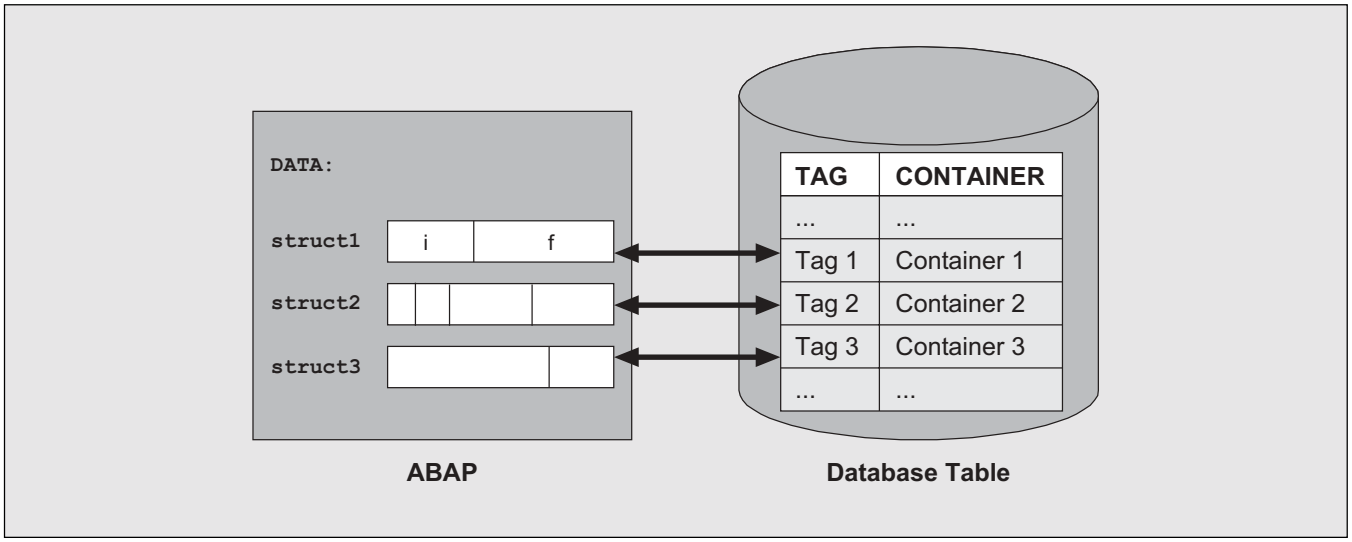
Figure 17 Code Terminates When It Finds No Numerical Value

```
DATA: BEGIN OF flat_struc,
        number TYPE p,
END OF flat_struc.

flat_struc = 'Hello, World !!'.

ADD 1 TO flat_struc-number.  " Exception !!
```

Figure 18 The Anonymous Container Technique



But even following this rule is not really a safe bet in all possible scenarios, as you can see in the next example!

Pitfall #10: Using Flat Structures with Numeric Elements in Anonymous Containers

Figure 18 illustrates a frequently used container technique. A database table can contain data of different structure types in an anonymous container. The type of the anonymous database container field typically is type c (or type x). A preceding field in each database table line contains tags that serve as an identifier for the structure of the container data. With the tag value, an ABAP program can determine which structured type it has to use to cast the contents of the container. For example, "Tag 1" in the first column of the database table might identify a structured type that consists of two fields, a leading field of type i and a second field of type f.

Let us assume that for both fields a value of 1 was once set and stored into the corresponding container for structure `struct1` in the database. Now our task is to read the data out again. This can be done by the sample code shown in **Figure 19**.

Surprisingly, the output can depend on the hardware platform of the application server. **Figure 20** shows the two possible results. The second result is probably not what you expected!

SAP supports a variety of different application server platforms — i.e., MS Windows, Solaris, HP-UX, AIX, Linux, and much more. The different platforms use different hardware architectures, and one remarkable difference is the byte order in which built-in types, like 4-byte integer, are stored. For example, on Intel-based platforms like Windows NT or Linux, the byte order is *little endian,* where the most significant byte is stored at the highest address where the integer value resides. In contrast, most UNIX platforms have a *big endian* byte order, which places the least significant byte at the highest address. Therefore, in switching from Windows NT to Solaris, for instance, the order of the 4 bytes for an integer is actually the other way around.

The different byte orders typically do not present a problem, and little endian mixed with big endian machines can coexist in the same SAP system pretty well. In fact, conversions that support data exchange between the servers are automatically performed to fit the endian architecture.

But in order to initiate the endian conversion, the

Figure 19 Reading the Data Out Again After the Value Is Stored

```
DATA: BEGIN OF struct1,
        int   TYPE i,
        float TYPE f,
      END OF struct1.

SELECT ... FROM dbtab INTO wa
           WHERE tag = 'TAG 1'.

struct1 = wa-container. "   Cast !

WRITE / 'struct1-int:   ', struct1-int.
WRITE / 'struct1-float: ', struct1-float.
```

Figure 20 The Two Possible Output Results

Field	Result 1	Result 2
struct1-int	1	16,777,216
struct1-float	1,0000000000000000E+00	3,0386519416174186E-319

type information of the raw data must still be available. This is the essential prerequisite. If raw data of 4 bytes, for example, is transferred from one server to the other and the type information is missing that says these 4 bytes are an integer, no endian conversion can take place.

Such situations can result from container techniques, as in the previous example. First, a sequence of a 4-byte integer (data type i) and an 8-byte floating point value (data type f) is stored in the anonymous container of the database, conserving the byte order of the updating application server. On the database, there is no more type information available for the single components of the structure stored in the container. Later, the data is re-imported by the example code from Figure 19. If this happens on a server with a different byte order, the implicit structure cast during the assignment from `wa-container` to `struct1` leaves the byte order unchanged, and the data is misinterpreted.

Using container techniques with numeric values like data objects of type i and f is a serious pitfall, since everything works fine as long as you execute your programs in homogeneous server environments. Only when saving and reading of containers is done in environments with servers of different endian architecture do unexpected errors occur.

The only effective measure to take to prevent this trap is to avoid any numeric components in structures that are saved in anonymous containers. Only the character-like types c, n, d, or t can be saved without impunity. In order to store numeric information in anonymous containers, convert the numeric data objects to character-like components via simple value assignments.

Conclusion

This concludes our tour of the most common traps and pitfalls we have encountered when dealing with fundamental operations and data types in ABAP. In short:

✓ For basic arithmetic, you learned how ABAP determines the arithmetic type and why fields of data type p should be well dimensioned to avoid inadequate rounding, as well as how to circumvent the notorious floating point rounding problem.

✓ When dealing with literals, you now know when you should use text literals and when to use numeric literals.

✓ We showed you the problems with in-place data declarations, which you should generally avoid at almost all costs.

✓ Finally, we hope you now realize the perils of numeric elements in flat structures when dealing with implicit flat structure casts, especially in conjunction with anonymous containers, where numeric elements should be avoided. The error of using anonymous containers with numeric fields does not come to light while working in a homogeneous applications server environment. Only in a system containing servers with different byte orders will the error occur. But when this problem does arise, it's a very serious one, particularly since it can occur in an environment in which you do not have access and therefore cannot perform any testing. The only surefire way to avoid this pitfall is to avoid using numeric fields in conjunction with anonymous container techniques.

We hope we leave you well prepared with the knowledge of the most prevalent traps and pitfalls concerning fundamental operations and data types in ABAP, in order to avoid the often difficult and tiresome process of searching for software bugs.

Christoph Stöck studied chemistry at the Georg-August-University of Göttingen, Germany, and received his doctorate in physical chemistry at the Max Planck Institute of Flow Research, Göttingen. He joined SAP in 1996, where he became a member of the Business Programming Languages Group.

Working as a kernel developer, Christoph is mainly responsible for the arithmetic and conversion features of the ABAP language. This includes, in particular, the software emulation of type p arithmetic and the design of semantics for and implementation of the conversion between all ABAP data object types.

Moreover, Christoph is responsible for the time service of the SAP application server, time zone and time-stamp handling in the ABAP language, and is engaged in many other fields of the ABAP processor, like ABAP byte code generation.

He can be reached at christoph.stoeck@sap.com.

Horst Keller studied and received his doctorate in physics at the Darmstadt University of Technology, Germany. After doing research in several international laboratories, he joined SAP in 1995, where he worked in the ABAP Workbench Support Group for three years. During that time, he authored the ABAP programming books for the printed ABAP Workbench Documentation Set for R/3 Release 3.0.

Horst is a member of the Business Programming Languages Group, where he is responsible for information rollout. He is documenting the ABAP language with an emphasis on ABAP Objects. He is the author of several articles and a widely renowned workshop on this subject, and his best-selling book "ABAP Objects" is currently being translated into English. Besides his writing tasks, Horst is developing the programs that display the ABAP documentation and the corresponding examples in the SAP Basis System.

He can be reached at horst.keller@sap.com.

