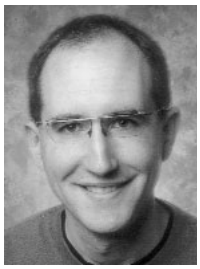
Processing text-based information in ABAP — best practices for improving performance

by Ralph Benzinger and Björn Mielenhausen



Ralph Benzinger
SAP NetWeaver
Application Server ABAP,
SAP AG

Björn Mielenhausen
SAP NetWeaver
Application Server ABAP,
SAP AG

(Full bios appear on page 68.)

A substantial part — if not the vast majority — of the data that is processed by typical business applications today is stored as *text*, or sequences of characters. Typical examples include names, telephone numbers, or product descriptions, as well as other kinds of information such as dates, currencies, or XML data stored using some textual representation. *Binary data* such as images, encrypted data, or compressed data is also used extensively by business applications, and while not represented by characters per se, may often be amenable to being processed as sequences of bytes. We refer to both kinds of data collectively as *textual data*.

The ABAP programming language tightly integrates text and byte processing capabilities into its runtime environment by providing a diverse set of data types and statements dedicated specifically to this task. Understanding the inner workings of the ABAP runtime in conjunction with a judicious choice of data types is thus key to processing text effectively and efficiently.

In this article, we provide a broad overview of the basic text and byte processing principles in ABAP and show how ABAP programs can be improved to increase their text processing performance. In addition to summarizing best practices in dealing with textual data, this article is also a prelude to a two-part article series on regular expressions to be published in upcoming issues of *SAP Professional Journal*. Regular expressions are the de facto standard tool for searching and manipulating text and offer many advanced and highly efficient operations that will boost the text processing capabilities of your ABAP programs.

But let's cover the basics first. In this article, we start with a brief introduction to the basic textual data types and their characteristics as well as their performance implications. This includes some technical information about conversions, internal storage, and string optimizations. The second part of the article covers the ABAP statements for creating, analyzing, and manipulating textual data.

```
DATA mytext TYPE c LENGTH 200.
mytext = 'Johann Gambolputty de von Ausfern' &
         'Schplendenschlittercrasscrenbonfried-' &
         'diggerdangledungleburstein'.
```

Figure 1 Concatenating character literals with ampersands

We hope that this article will be helpful to ABAP developers dealing with general business application programming. While we assume only a minimal level of ABAP knowledge, we present some mild technical details that will allow you to harness the text processing capabilities of ABAP for improved performance.

Data types for storing text

ABAP offers two fundamentally different ways of storing textual data in programs. *Fixed-sized fields* hold text that cannot exceed a predefined fixed number of characters or bytes, whereas *variable-sized fields* store arbitrarily large amounts of text. The most generic data types can store arbitrary data, but there are some special-purpose data types that impose certain restrictions upon the domain of their values as well.

Both fixed-sized fields and variable-sized fields have their own specific advantages and disadvantages that we need to keep in mind when choosing a type for our data. In the next sections, we examine each type more closely and give some practical advice to help developers make the optimal choice.

Fixed-sized data types

Fixed-sized data types are the most basic data types available in ABAP and store textual data of some predefined length. They come in two flavors: *Character-based data types* store sequences of characters, whereas *binary data types* store sequences of bytes. We continue to refer to textual data whenever the distinction between character-based data and binary data is of no importance.

Character data types

Fields of *character type* `c` store arbitrary characters, including letters, digits, punctuation symbols, control codes, and funny squiggles. Internally, characters are stored as integers whose interpretation as actual symbols depends on the locale settings of the SAP system and the current user session.

For fields of type `c`, the number of characters must be specified statically as part of the variable declaration, where the maximum length is restricted to 65,535 characters:

```
DATA mycfield TYPE c LENGTH 100.
```

Alas, this prevents us from allotting space dynamically in cases where we would have access to some additional information about the storage requirements at runtime. Recent SAP NetWeaver releases, however, provide functionality for dynamic type creation that can be employed as a workaround for this problem whenever variable-sized data types (discussed in an upcoming section) are not appropriate.

The easiest way to include textual data in your ABAP programs is through the use of *character literals*. A character literal is a character sequence enclosed in single quotes (`'…'`) that defines a constant character value. To include a single quote in a literal, the quote is doubled, as in:

```
'Let''s go!'
```

Literals cannot span lines in the editor, but they can be concatenated by using ampersands to construct larger literals, as shown in **Figure 1**. The total length of any literal, however, is limited to 255 characters.

Character literals should not contain any “exotic” characters above code point 127, since they may get converted into unexpected characters if the locale

About Unicode and multi-byte encoding…

In modern Unicode systems, a character can be any of over 65,000 funny symbols, including Latin, Asian, and Klingon characters. For older, non-Unicode systems, the story is slightly more complicated. By default, each character can be only one of 255 different symbols, which of course is way too few to accommodate all of Earth's known languages (not to mention the rest of the Milky Way). Because of this limitation, *code pages* were introduced that define how each of the 255 available values translates into a character symbol. Most languages have their own code page, so that by selecting the appropriate code page we can enter all of our language's symbols.

For some languages, however, this is still not enough, since more than 255 characters may be in use. For those languages *multi-byte code pages* have been defined. In a multi-byte code page, a single symbol may actually be stored internally as two characters, extending the value range from 255 to 65,535 and mimicking a Unicode system. Consequently, a `c` field of length 10 may store up to 10 characters, but on a multi-byte system, only 5 two-byte characters will fit in such a field. By mixing single-byte characters and two-byte characters of a multi-byte code page — e.g., Romanji and Kanji characters of the Japanese code page — any number between 5 and 10 characters may fit into our `c` field of length 10.

settings of the developer system and the locale settings of the user system running the program differ.

When developing low-level system reports, we often have to deal with control codes such as form-feeds, newlines, or carriage returns, which cannot be included in character literals. Suppose, for example, that we received some text document `mytext` that contains newline characters that we want to remove. The utility class `cl_abap_char_utilities` provides a selection of constants that we can use to work with control characters:

```
REPLACE ALL OCCURRENCES OF
        cl_abap_char_utilities=>newline
        IN mytext WITH ''.
```

Using `cl_abap_char_utilities` provides the additional advantage of automatically choosing the correct platform-dependent control code. If we have more peculiar needs, we can use field symbols to force the inclusion of non-standard characters. It should be noted, however, that this technique must be viewed as an undesirable kludge in all but the most low-level system programs.

Whenever we store a textual value whose length is shorter than the available space reserved for the field, the unused portions of the field are set to a well-defined value. In ABAP parlance, we say that the value is *padded* when stored in fixed-sized fields. For `c` fields, the unused part of the field is filled with blanks. Since we cannot distinguish between blanks that are the result of automatic padding and those that have been appended intentionally, trailing blanks are always assumed to be the result of padding and thus are almost never included in the actual value. As an example, after executing the statements:

```
DATA: mystr1 TYPE c LENGTH  7
             VALUE 'hello',
      mystr2 TYPE c LENGTH  7
             VALUE 'world',
      mystr3 TYPE c LENGTH 13.
CONCATENATE mystr1 mystr2 INTO mystr3.
```

fields `mystr1`, `mystr2`, and `mystr3` contain values `hello␣␣`, `world␣␣`, and `helloworld␣␣␣`, respectively, where we highlighted blanks by the symbol ␣. Since trailing blanks are ignored by ABAP, we opt

to ignore them for the remainder of our discussion as well and thus will say that `mystr1` has the value `hello` rather than `hello␣␣`. To spice things up, there are a few ABAP statements where trailing blanks *do* remain significant for some parameters, but in general these changes in behavior are well-founded and thoroughly documented.

Trailing blanks are also ignored if they occur in character literals. There is one special case, though, that we need to address. Since there are no fields of length 0, the empty character literal `''` is actually interpreted as a literal containing a single blank `'␣'`. In general we won't notice this, though, as this literal constitutes a trailing blank that will be ignored by most statements. There is also a predefined constant (viz., `space`) of value `'␣'` whose use makes ABAP programs much nicer to read.

Ignoring trailing blanks can lead to unexpected results for the unwary, and we will provide some examples in the upcoming section "Basic text processing operations." In most cases, potential pitfalls are easily circumnavigated by using strings, which we cover in the section "Variable-sized data types."

If we try to store a value that is longer than a given `c` field, the value is truncated. Depending on the ABAP statement involved, this truncation may be indicated by a special `sy-subrc` return value.

Binary data types

Fields of binary type `x` store sequences of 8-bit bytes. Most often, `x` fields are used to allocate storage space for binary objects that have no textual representation (such as a JPG image received via RFC or a spreadsheet read from the file system). In some instances, however, `x` fields may contain binary data that has some structure imposed on it and that is thus amenable to textual processing.

Most ABAP statements for text processing offer a special binary mode that is generally invoked by the `BYTE MODE` addition (other statements such as `MOVE` automatically detect the binary type of their operands). Assignment, searching, replacing, and concatenation are just some of the operations that offer such a binary mode. Most of our subsequent discussion on character fields thus holds equally true for binary fields; we will note any differences where appropriate.

By convention, a single byte is represented by a two-character sequence denoting its hexadecimal value ranging from `00` to `FF` (note that hexadecimal digits `A` through `F` must be in upper case). This convention is built right into ABAP, as we can witness when we try to assign or print binary values. As a consequence, there are no real binary literals; instead, binary values are defined by character literals, and most ABAP statements supporting a binary mode convert character literals into their corresponding binary values on the fly. In the assignment:

```
DATA mybytes TYPE x LENGTH 2.
MOVE '4F4B' INTO mybytes.
```

for example, the four-character literal `'4F4B'` is interpreted by the `MOVE` statement as a two-byte sequence of hexadecimal values `4F` and `4B` that, when re-interpreted as characters, would read `OK` in a non-Unicode system. As a general rule, since ABAP supports only hexadecimal representations of binary data, we can take note that the representation of an `x` field of length *n* requires 2*n* characters — which require 4*n* bytes of storage on a Unicode system!

Similar to `C` fields, we must specify the length of `x` fields at the time of their declaration, with a maximum length of 65,535 bytes. Values that are shorter than the size of an `x` field are padded with zeros (`00`) to the right, so that:

```
DATA mybytes TYPE x LENGTH 4
             VALUE 'ABCD'.
```

will assign byte sequence `ABCD0000` to variable `mybytes`. Trailing `00`s will *not* be ignored, though, since comparing binary objects stored in an `x` field requires a very strict definition of equality!

If we try to store a value that exceeds the available space of an `x` field, the value is truncated and the statement may set a special return code.

Since we are using character literals to define binary values, mischievous readers may already be

wondering what might happen when we try to assign an invalid value to an x field. The outcome of operations involving invalid values such as:

```
DATA mybytes TYPE x LENGTH 2.
mybytes = '12XY34'.
```

is sufficiently complex to warrant a section of its own. Suffice to say for now that the above assignment cuts off the value at the invalid byte so that mybytes will equal 1200 upon completion of the code. We will provide a more comprehensive discussion on invalid values in the section "Working with fixed-sized fields."

Special-purpose data types

While c and x are the most commonly used fixed-sized data types, we also have at our disposal three additional character data types tailored toward very specific tasks.

The character type n is a specialization of type c that is intended for *numerical data* such as customer numbers or invoice numbers. But unlike values of integer type i, type n fields should not be used in computations — they merely store the textual representation of numerical values. Because of this distinction, their padding rules differ from those of c fields in that digits are stored right-aligned and padded with leading zeros to the left. The following code:

```
DATA mynum TYPE n LENGTH 5 VALUE '123'.
```

stores value 00123 in variable mynum. Again, there are no explicit n literals; we simply reuse our character literals and rely on the MOVE statement to convert the character sequence into a numerical representation. Any non-digits part of the character sequence being moved into the n field are simply stripped from the value:

```
DATA: mynum1 TYPE n LENGTH 5
             VALUE '12X34',
      mynum2 TYPE n LENGTH 5.
mynum2 = '12X34'.
```

Both fields mynum1 and mynum2 now contain the character sequence 01234. Note that this behavior differs from that of x fields, where invalid values caused the conversion to be stopped.

The character types d and t are also specializations of type c and denote dates and times, respectively. Internally, they are character fields of length 8 (for type d) and 6 (for type t) that store date and time values encoded as YYYYMMDD and HHMMSS, where each letter represents the usual year, month, day, hour, minute, and second. Since their internal format is fixed, we cannot declare a length for d and t fields at all.

Values of type d and t are padded with blanks to the right, and invalid values such as character sequences containing non-digits are moved verbatim into the field. For example, in the case of the following code fragment:

```
DATA mydate TYPE d.
mydate = '12X34'.
```

variable mydate will contain the value 12X34□□□, which clearly is not a valid date. Why the grandparents of ABAP decided to pad dates with the invalid date character □ remains their own secret!

ABAP provides several services for processing fields of types n, d, and t that respect different cultural customs as defined by the locale. The code fragment:

```
DATA mydate TYPE d VALUE '20060713'.
WRITE / mydate.
```

for instance, will output the date with the month preceding the day as in 7/13/2006 when run in a US locale, whereas the same date is rendered as 13.7.2006 in most European locales.

Working with fixed-sized fields

As a guiding principle, we may state that the outcome of any operation in ABAP depends mostly on the ABAP statement and less on the data type of its operands. In plainer words, certain conventions for a given data type may hold for some ABAP statements, but not for others. We have already mentioned this phenomenon in our previous discussion on the significance of trailing blanks.

While working with data types looks neat and tidy so far, things get more complicated when we try to mix and match values of different data types. This happens much more frequently than we might expect at first, since we must rely on `c` literals even for binary and numeric data.

The `MOVE` statement is the central authority for converting between values of different types, since all conversions are implemented in the `MOVE` statement, and other statements use the `MOVE` statement functionality for conversion.

For the five data types introduced so far (`c`, `n`, `x`, `d`, and `t`), we must familiarize ourselves with 5 × (5-1) = 20 different conversion rules — the actual number found in the ABAP online documentation for all 11 built-in types even exceeds 100! Fortunately, we rarely need to convert, say, a numerical field to a date or vice versa, so our working set of conversion rules is actually quite manageable.

While the `MOVE` statement goes to great lengths to avoid the assignment of invalid values by automatically applying appropriate conversions to the data, other ABAP statements are much more permissive in that they will perform the operation with both valid and invalid data. The `CONCATENATE` statement, for instance, does not perform any kind of conversion at all. Upon completion of the following code fragment:

```
DATA mynum TYPE n LENGTH 5.
CONCATENATE '1' 'X' '2' INTO mynum.
```

field `mynum` contains character sequence `1X2□□`, which not only contains an invalid non-digit but also is padded like a `c` field rather than an `n` field. The byte mode case is safe, however, since ABAP does not auto-convert character literals into byte sequences, meaning that:

```
DATA mybytes TYPE x LENGTH 5.
CONCATENATE '11' 'X' '22' INTO mybytes
            IN BYTE MODE.
```

is syntactically incorrect.

Similar to `n` fields, both `d` and `t` fields permit the storage of invalid characters (i.e., non-digits), which may yield unexpected results. Instead of trying to exploit the intrinsics of implicit date and time conversion, though, we strongly recommend viewing all `d` and `t` fields as opaque entities and not to use text processing operations on these data types unless explicitly required.

Fixed-sized fields are laid out as contiguous blocks of characters or bytes. If a structure contains only character-based components of type `c`, `n`, `d`, or `t`, the entire structure may be processed as a character field:

```
DATA: BEGIN OF mystruc,
        prod_code TYPE c LENGTH 5,
        cust_id   TYPE n LENGTH 5,
      END OF mystruc.
mystruc = 'PROD□12345'.
```

The value of such a structure comprises the values of all its components concatenated in the order of the declaration, including trailing blanks for all components but the very last.

We hasten to say that this code pattern reflects extremely bad style and should not be used in new development. If we were to operate on the entire structure, values of individual components might spill over if the entire character sequence is shifted, as in:

```
REPLACE 'D' IN mystruc WITH ''.
WRITE: / mystruc-prod_code,
       / mystruc-cust_id.
```

which will print:

```
PRO□1
2345□
```

In non-Unicode legacy systems, we could even include binary fields in the above fashion, but the resulting disarray was sufficient to have this misfeature disabled in current releases.

Variable-sized data types

The fixed-sized data types discussed in the previous section require us to allot a fixed amount of storage for each field, independent of the actual value that the

field is going to hold. This potential waste of storage space frequently turns into a serious design issue if we have a large number of fields and the average length of values is relatively small compared to the longest possible length.

Suppose, for example, that we want to declare a fixed-sized character field for storing customer names. What length should we assign to our field? If we expect mostly Asian names like *Chang*, *Lee*, or *Sato*, we may conclude that names will not exceed, say, 20 characters in length. But no matter how long our field, sooner or later we will meet customers *Edson Arantes do Nascimento* (aka Pelé, the famous Brazilian soccer player) or *Hadschi Halef Omar Ben Hadschi Abul Abbas Ibn Hadschi Dawuhd al Gossarah* (a fictitious Arabian character from a novel by Karl May). We would waste a lot of memory if we reserved 80 characters or more for our customer names just to accommodate these unconventional names, when in 99 percent of all cases 20 characters would be more than sufficient.

This fact was not lost on the ABAP developers, of course — ABAP release 4.6 introduced two *variable-sized data types* that do not require a length declaration but automatically grow and shrink depending on the storage requirements of their current value.

Strings

Variables of type `string` store character sequences of arbitrary length. The ABAP runtime automatically allocates additional storage if the string grows, and automatically frees unused storage if it shrinks. The resize strategy is highly optimized and keeps storage reorganizations to an absolute minimum by adding a cleverly computed safety margin to all storage allocations.

The maximum length of a string is currently limited to around 2 billion (2^{31}-1) characters. This is only a theoretical limit, though, since the profile parameter ztta/max_memreq_MB limits the maximum number of bytes that can be allocated for all dynamic data objects, including strings. The default value of 128 megabytes limits `string` variables to about 50 million characters if we assume a Unicode system that allocates two bytes per character.

Strings have their own literals, which are enclosed in single back quotes (`…`). To include a back quote in the string, we double the back quote, as in:

```
`the ``new' thing`
```

The empty string literal `` denotes an empty string of length 0 — recall from the previous section that this differs from fixed-sized fields where the empty character literal '' actually denoted ' '. Thus, in situations where trailing blanks are significant, we always use `` instead of '' if we require an empty character sequence.

Since the lengths of strings automatically adjust to their values, there is no need for padding, and hence trailing blanks are always significant. We need to keep this in mind when we compare strings with character fields, as the comparison in:

```
IF 'this  ' = `this  `.
   " never executed
ENDIF.
```

always evaluates to `abap_false`.

This also introduces a potential pitfall when we move numeric values to a string. Since ABAP uses the commercial convention of placing the sign behind the value, but uses a blank ( ) instead of a plus sign (+) for positive numbers, it is easy to introduce unintended trailing blanks in `string` variables. For example, in the following code fragment:

```
DATA: myvalue  TYPE i VALUE +123,
      mystring TYPE string.
mystring = myvalue.
```

the value of `mystring` will equal `123 ` instead of `123`, which may catch us off guard if we compare `mystring` to a character field or literal.

Analogous to fixed-sized data types, variable-sized data types also offer a type for binary data. Variables of type `xstring` store byte sequences of arbitrary length, subject to the same restrictions as those of character strings.

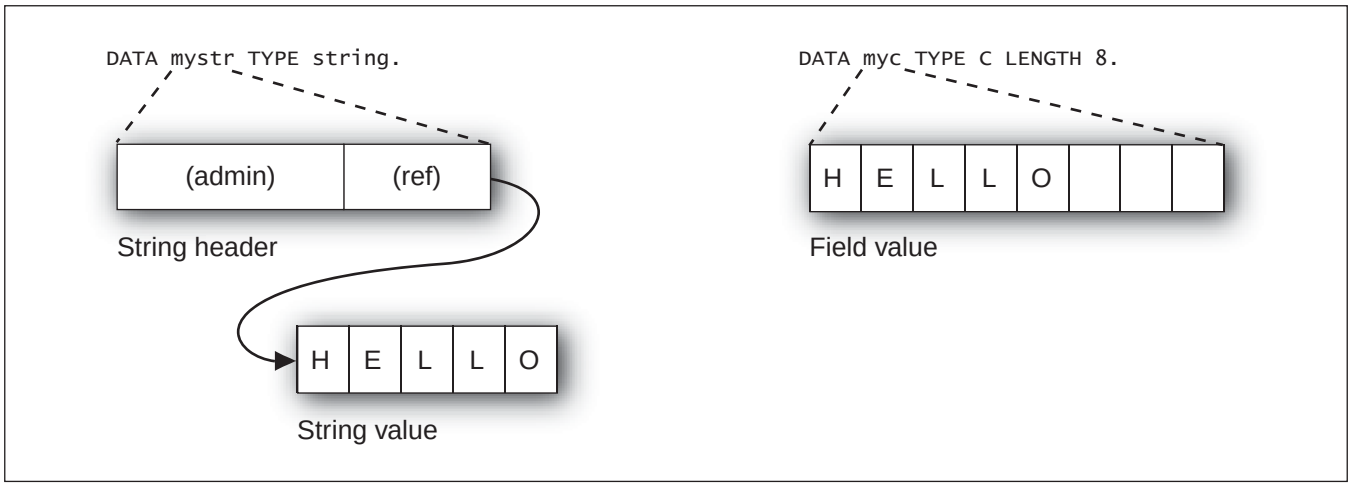


Figure 2 String variable vs. fixed-size field

```
DATA: mystring1 TYPE string,
      mystring2 TYPE string.
mystring1 = `hello, world`.
mystring2 = mystring1.                          " shared
CONCATENATE mystring1 '!' INTO mystring1.  " unshared
```

Figure 3 Sharing a string value

Sharing of string values

If we want strings to shrink and grow at runtime, we must split `string` variables into a value part and an administrative part (also called the *header*). The value part stores the actual text of the string and may float around in main memory. If the string grows beyond the current size of its value part, we simply allocate a larger chunk of memory and copy the string value to the new value part. Similarly, if the size of our string drops below a certain threshold, we move it to a smaller chunk of memory and release the larger value part. The administrative part keeps track of the current location of the value part and also stores some additional information such as the current size of the string. We may thus think of a `string` variable as a reference pointing to the string somewhere in memory, as shown in **Figure 2**.

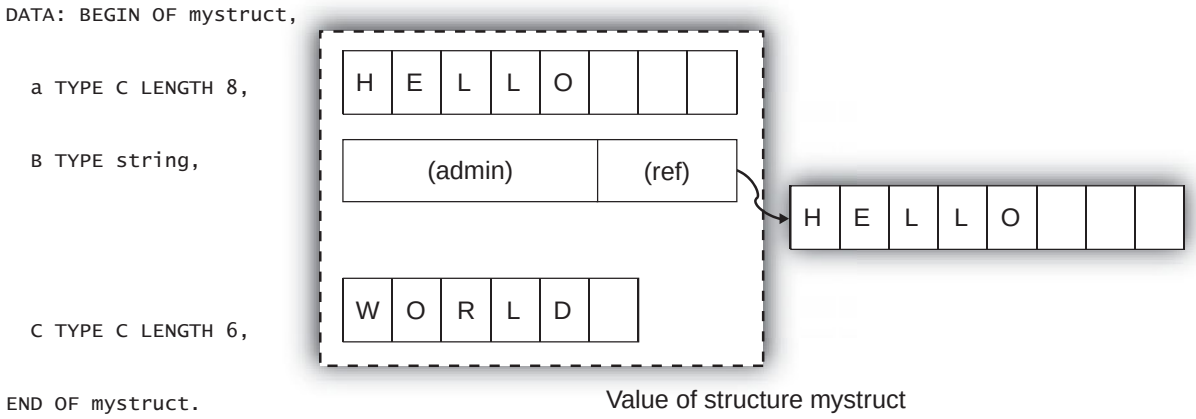
This decoupling of header and value is the basic idea that allows strings to grow and shrink on the fly. But the advantages do not stop there: if we separate the value of a string from its header, we can reuse a given value for multiple headers, i.e., for multiple `string` variables! This principle is known as *string sharing*, and it greatly increases performance while it reduces storage requirements at the same time.

Variables sharing the same value are called *synonyms*. If we change the value of one synonym, it ceases sharing its value by creating an exclusive copy of the text. Subsequently, we proceed by operating on the newly created copy. Because of this behavior, the technique is known as *copy-on-write* or *lazy copying*.

How does ABAP detect such synonyms? The most obvious opportunity for sharing some string value is when we move the value between two variables of type `string`, as shown in **Figure 3**. Instead of copying the string, the destination variable will share the value of the source variable. The same mechanism

About structures containing strings...

Since the value of a `string` variable is stored separately from the header, a structure containing a component of type `string` cannot be processed as a character field:



Strings resemble internal tables or object references in that they reference some memory location that holds the actual value. Thus, any structure containing a string is classified as *deep*. (Interestingly, structures that are not deep are not *shallow* — they are *flat*.) Some restrictions apply to deep structures, which must be considered when changing the type of a component of a structure from type `c` to type `string`.

is used for the value parameters of forms, functions, and methods.

There are additional situations where sharing is appropriate, such as when fetching data from the database into `string` variables. To increase the sharing ratio, the ABAP runtime utilizes a cache that stores a limited number of the most recently used string values. There is no guarantee, however, that any two strings with the same value will actually be shared; but we may rest assured that frequently used string values will be allocated no more than a few times.

Generic data types

ABAP features a number of *generic data types* whose aim is to bring a semblance of order to our zoo of textual data types. Since generic types are inherently ambiguous, they cannot be used to declare variables and are only permitted for use with formal parameters and field symbols.

The generic type `csequence` comprises types `c` and `string` and should be used whenever we expect some character-based value. The type `clike` is a little more permissive since it also includes types `d`, `n`, and `t`, as well as character-type flat (i.e., non-deep) structures. In non-Unicode systems, `clike` also includes type `x` — we strongly discourage relying on `x` being included in `clike`, however, as this may prevent your programs from being run in Unicode systems.

For binary data, the generic data type `xsequence` comprises types `x` and `xstring` and is the data type of choice for interfaces that expect binary data.

How to choose the right type

While `string` variables offer the most flexibility for processing text, they also introduce a certain amount of internal overhead that may slightly degrade performance. If we want to write efficient programs, we must thus organize our data structures judiciously. Alas, neither is there a bulletproof method for choosing the right data type, nor can the ABAP runtime exonerate us by automatically selecting the proper type during program execution. We collected some guidelines, though, that will assist us in determining the appropriate type for text-based variables.

Know the size of your text!

One significant indicator for the appropriate data type is the amount of data we expect. Because accessing strings involves a fixed overhead that is independent of the length of the string, strings do not perform well when used for very short character sequences. As string values increase in size, however, the fixed overhead becomes relatively less costly. As a rule of thumb, if our values are longer than 10-20 characters, we should store them in a `string` variable, and in a character field otherwise.

Track string sharing and unsharing!

While accessing strings is slightly more costly than accessing fixed-sized fields, we can copy strings very efficiently because of string sharing. This performance gain is offset by the amount of unsharing required when we frequently modify shared string values.

String sharing may also significantly reduce memory consumption if only a few distinct values may occur for a large set of variables or in some parts of an internal table. Suppose we have an internal table containing customer data. If there are many customers living in, say, New York City, we could improve memory utilization by declaring the `city` column of our internal table as type `string` since this would share one instance of `New York City` among all customers living in this city. Customer numbers, on the other hand, are unique for each customer, so the use of strings (in hopes of sharing common values) would be futile.

Mind your trailing blanks!

While we have argued before that short values should be stored in fixed-sized fields, we now describe a scenario in which even short strings may outperform their fixed-sized counterparts.

Most text-processing statements depend on the length of the value that they operate upon. For strings, this information is stored in the directly accessible header part. Fixed-sized character fields, on the other hand, need to be scanned right-to-left to account for trailing blanks, which must be ignored. The more trailing blanks there are — i.e., the shorter the actual value relative to the length of the field — the poorer the performance compared to a string. For fields whose value `IS INITIAL`, this results in scanning the entire field.

Basic text processing operations

Now that we have introduced all of our textual data types, we subsequently turn our attention to the text processing operations that the ABAP language offers. The most common ones, searching and replacing, have been given their own section.

Offset-length access

When talking about character sequences in ABAP, we frequently refer to subsequences or individual characters of some larger sequence. To avoid any feuds over the question of whether the first character of a sequence should have index 0 or 1, the designers of ABAP decided to please both camps by offering *offsets*, which are 0-based, and *positions*, which are 1-based. So, for literal `hello`, the `o` is found at offset 4, but also at position 5.

In ABAP we can easily access any subpart of some given field or string by using the offset-length idiom var+*off*(*len*). This expression denotes the subpart of *var* that comprises the characters from offset *off* up to and including the character at offset *off*+*len*-1. Both offset and length are optional and default to 0 and the


```
DATA mytext TYPE c LENGTH 12 VALUE 'hello world!'.
mytext+6(5) = 'sun'.    " mytext now equals 'hello␣sun␣␣!'
mytext+6(2) = 'world'.  " mytext now equals 'hello␣wor␣␣!'
```

Figure 4 Text that is longer than the text it replaces is truncated

length of the remainder of the sequence, respectively. An asterisk (`*`) as length also denotes the length of the remainder of the sequence. Both offset and length may be numerical variables or constants, but not full-fledged arithmetical expressions. The following `WRITE` statements:

```
DATA mytext TYPE string
            VALUE `hello world`.
WRITE: / mytext+3(2), / mytext(5),
       / mytext+6,    / mytext+6(*).
```

will thus output:

```
lo
hello
world
world
```

If we try to access some part outside the character sequence such as in `mytext+20` or `mytext(20)`, the ABAP runtime will point out our foolishness by raising exception `cx_sy_range_out_of_bounds`.

We may also use the offset-length idiom in a writing position if we want to modify a certain part of some text:

```
DATA mytext TYPE c LENGTH 20
            VALUE 'hello world'.
mytext+6(5) = 'Ralph'.
```

For technical reasons this construct may only be used for fixed-sized character fields, and not for strings. All is not lost, however, since the `REPLACE` statement offers a suitable alternative even for strings.

If the new text is shorter than the specified subpart to be replaced, it will be padded with blanks; if it is longer, it will be truncated, as shown in **Figure 4**.

As we can see, all parts outside the subfield specified by offset and length remain unchanged.

Concatenation

We can join two or more character values with the `CONCATENATE` statement, where we may specify an optional separator using the `SEPARATED BY` clause. For example, in the following code fragment:

```
DATA mystr TYPE string.
CONCATENATE 'hello' 'my' 'world'
            SEPARATED BY '␣' INTO mystr.
```

variable `mystr` will contain the value `hello␣my␣world`. Observant readers will have noticed that the `SEPARATED BY` clause will *not* ignore trailing blanks in its arguments — if it did, we would have ended up with `hellomyworld` instead. If we concatenate into a fixed-sized field, but the size of the field is insufficient to store the entire result, return code `sy-subrc` is set to value 4. This applies to most text processing statements in ABAP (although the notorious `REPLACE` statement opts to use the value 2 instead).

A common concatenation pattern is to repeatedly append smaller text fragments to build up some larger text:

```
DATA: itab TYPE TABLE OF string,
      mystr TYPE string.
FIELD-SYMBOLS <fs> TYPE string.
LOOP AT itab ASSIGNING <fs>.
  CONCATENATE mystr <fs> INTO mystr.
ENDLOOP.
```

The ABAP runtime will recognize this pattern and perform some special optimizations that will reduce the amount of reallocations required to accommodate

```
DATA mytext TYPE c LENGTH 12 VALUE 'hello world'.  " mytext now equals 'hello world '
SHIFT mytext RIGHT CIRCULAR BY 6 PLACES.           " mytext now equals 'world hello '
SHIFT mytext BY 5 PLACES.                          " mytext now equals ' hello      '
```

Figure 5 Shifting a fixed-sized field

the growing string `mystr`. This optimization only applies, however, when we append at the *end* of the string; the opposite case:

```
CONCATENATE <fs> mystr INTO mystr.
```

will not be optimized at all.

Splitting

If we can concatenate text we surely ought to be able to split it up again. The `SPLIT` statement will separate a given text into individual parts at some one-character delimiter:

```
DATA mytext TYPE string
            VALUE `Ralph,12345,SAP`.
SPLIT mytext AT ',' INTO arg1 arg2 arg3.
```

Variables `arg1`, `arg2`, and `arg3` now contain values `Ralph`, `12345`, and `SAP`, respectively. If splitting yields fewer parts than there are variables to fill, the extraneous variables will be cleared. In the opposite case, though, the last variable will receive the unsplit remainder of whatever was left at this point:

```
SPLIT mytext AT ',' INTO arg1 arg2.
  " arg1 now equals 'Ralph'
  " arg2 now equals '12345,SAP'
```

The variant of the `SPLIT` statement used above works best if we know the number of delimiters contained in our text in advance. For the general case, we had better supply an internal table that will receive each part:

```
DATA itab TYPE TABLE OF string.
SPLIT mytext AT ',' INTO TABLE itab.
```

Shifting

We may move text around in various directions with the `SHIFT` statement. The default is to shift to the left, but by adding `RIGHT` we shift to the right. When shifting fixed-sized fields, characters shifted outside the field will be dropped unless we add the `CIRCULAR` option, as shown in **Figure 5**, which will reintroduce those characters at the opposite end.

The same applies for strings, with the exception that shifting non-circularly to the right will always grow the string and never drop any characters (see **Figure 6**).

In addition to specifying the number of characters to shift directly with the `BY n PLACES` addition, we may also use the `UP TO` addition to shift left up to a particular character sequence.

```
DATA mytext TYPE string
            VALUE `hello world`.
SHIFT mytext UP TO 'or'.
```

Variable `mytext` now contains value `orld`.

Formatting text

The `WRITE` statement not only puts text into the print spool but also formats text when invoked with the `TO` addition. (Alas, `WRITE TO` can only write into character fields but not into strings.) The final format depends on *implicit rules* according to different locale-specific settings and *explicit rules* that control aspects such as justification or rounding operations on numbers.

Let's illustrate the results of implicit and explicit rules by formatting a date. Implicit rules are derived


```
DATA mytext TYPE string VALUE `hello world`.  " mytext now equals `hello world`
SHIFT mytext RIGHT BY 1 PLACES.               " mytext now equals ` hello world`
SHIFT mytext LEFT CIRCULAR BY 7 PLACES.       " mytext now equals `world hello `
SHIFT mytext BY 6 PLACES.                     " mytext now equals `hello `
```

Figure 6 Shifting a string

```
DATA: mydate TYPE d VALUE '20060131',
      mytext TYPE c LENGTH 15.
SET COUNTRY 'US'.  " US locale
WRITE mydate TO mytext.  " mytext now equals '01/31/2006'
SET COUNTRY 'JP'.  " Japanese locale
WRITE mydate TO mytext.  " mytext now equals '2006/01/31'
```

Figure 7 Changing an implicit date formatting rule with the SET COUNTRY statement

from the settings of the master user record but may be changed in reports with the `SET COUNTRY` statement, as shown in **Figure 7**.

The available country codes for `SET COUNTRY` are stored in database table `T005X`. Further implicit rules control the appearance of decimal points, thousand separators, or time stamps. We can even define our own implicit rules through conversion exits that we assign to a domain in the Data Dictionary.

Explicit formatting rules further control the appearance of the result and are activated by additions. The addition `RIGHT-JUSTIFIED`, for example, pads the value to the left so that the result is aligned with the end of the field:

```
WRITE mydate TO mytext RIGHT-JUSTIFIED.
  " mytext equals '     01/31/2006'
```

The list of available explicit formatting rules can be found in the online documentation for the `WRITE` statement under the section `int_format_options`.

Other operations

The `TRANSLATE` statement converts a given text to upper or lower case:

```
DATA mytext TYPE string
            VALUE `Hello World!`.
TRANSLATE mytext TO UPPER CASE.
  " mytext now equals 'HELLO WORLD!'
```

As usual, the classification of upper and lower characters depends on the current locale settings.

If we want to remove extraneous blanks — i.e., two or more blanks in a row — we may either employ the `REPLACE` statement or we can simply use the `CONDENSE` statement:

```
DATA mytext TYPE c LENGTH 25.
mytext = 'hello  my world    !'.
CONDENSE mytext.
```

Upon completion, `mytext` contains the value `hello my world !          `. Obviously, trailing blanks cannot be condensed. By adding `NO-GAPS`, we can purge all non-trailing blanks from our text.

About explicit formatting rules for dates...

Explicit formatting rules for dates frequently cause confusion. The additions `DD/MM/YYYY` and `MM/DD/YYYY`, for instance, do *not* control the order of day and month in the result — instead, they both specify that the year is formatted with 4 digits and that separators are to be included. The order of day and month and the choice of appropriate separators, however, are still determined by the locale settings. The same observations hold true for the additions `DD/MM/YY` vs. `MM/DD/YY` and `DDMMYY` vs. `MMDDYY`.

Once we know about this peculiarity, everything could be bright and beautiful, but no — addition `YYMMDD` has to ruin it for everyone by not sticking to the locale settings:

```
DATA: mydate TYPE d VALUE '20060131',
      mytext TYPE c LENGTH 15.
SET COUNTRY 'US'.
WRITE mydate TO mytext YYMMDD.
```

This will yield the Japanese-style date `060131` without separators even though we selected a US locale. What a spoil-sport!

ABAP classes for text processing

Some of ABAP's text processing functionality is provided by classes rather than built-in statements:

- `cl_abap_char_utilities` — This class provides some fundamental services concerning single characters. This class is particularly useful for platform-independent operations since it defines constant attributes for control characters such as newline, horizontal tab, and form-feed. The constant `char_size` may be used to check if the current system is a Unicode system or not.
- `cl_abap_string_utilities` — This class addresses issues caused by the different semantics of trailing blanks of types `string` and `c`. The method `c2str_preserving_blanks` copies the value of a character field into a `string` variable while preserving all trailing blanks. The method `del_trailing_blanks` removes any trailing blanks from a `string` variable.
- `cl_abap_conv_in_ce`, `cl_abap_conv_out_ce`, and `cl_abap_conv_x2x_ce` — These classes convert between different code pages or number formats, respectively. They serve as a replacement for the obsolete variants of the `TRANSLATE … CODE PAGE` and `TRANSLATE … NUMBER FORMAT` statements.

All of the above classes are described in detail in the ABAP online documentation.

Searching and replacing

Searching and replacing text in a larger body of data are among the most frequently needed text processing operations and thus warrant a great amount of attention. In ABAP, most of this functionality is provided by the `FIND` and `REPLACE` statements, both of which offer a plethora of additions that allow us to customize searches and replacements to our liking.

The FIND statement

The `FIND` statement is the standard ABAP Swiss army knife for locating text in some body of data. While its complete syntax diagram looks worse than the

About the FIND, SEARCH, and REPLACE statements…

For the benefit of historically inclined readers, we mention that the FIND statement was originally introduced with release 4.6 as a replacement for the less powerful SEARCH statement. The REPLACE statement has been around much longer, but it used to sport a different syntax that is still supported but considered deprecated by now. As of SAP NetWeaver 2004s, both FIND and REPLACE natively support regular expressions, making them extremely versatile text processing powerhouses that solve many text processing tasks that previously would have required one to write many lines of ABAP code. In this article we are limiting our discussion to searching and replacing with substrings only, but forthcoming *SAP Professional Journal* companion articles will cover the use of regular expressions in ABAP in detail.

New York City subway map, its basic usage is simple and quite straightforward:

```
FIND SUBSTRING mystr IN mytext
     MATCH OFFSET off MATCH LENGTH len.
```

This statement searches the text mytext from left to right for the first occurrence of substring mystr and, if found, returns its offset in variable off. Similarly, the length of the substring found is returned in variable len, which is a bit silly since we already had a pretty good idea of how long that occurrence was going to be. (The situation will be completely different, though, when we search using a regular expression.) Note that the use of the keyword SUBSTRING is purely optional, so we will omit it from here on. Also note that despite the name of the addition, mystr need not be of type string but may be of any clike type.

If the substring was found, FIND sets sy-subrc to 0; if not, sy-subrc is set to 4 and variables off and len will not be changed.

ABAP actually offers not one but two ways to restrict the range we want to search. The canonical solution that is available for all statements is to use the offset-length idiom when specifying the text we want to search:

```
FIND mystr IN mytext+myoff(mylen).
```

This will limit the search to the part of mytext starting at offset off and ending just before offset off+len. This solution is perfectly fine for fixed-length fields, but incurs a small penalty for strings since a new temporary string object must be created. A better way to define ranges is to use the corresponding FIND additions. Since this approach avoids the creation of a temporary string, it will perform equally well for strings and fields:

```
FIND mystr IN SECTION OFFSET myoff
                      LENGTH mylen
              OF mytext.
```

As of ABAP release 7.0, we can also search internal tables of the appropriate type on a line-by-line basis:

```
DATA myitab TYPE TABLE OF string.
FIND mystr IN TABLE myitab
     MATCH LINE lin
     MATCH OFFSET off MATCH LENGTH len.
```

The above statement will store the line number of the substring found in lin. Note that there is no direct way to search the table with individual lines adjoined. This would be useful, for instance, if we had some natural language text split into 80-character chunks and stored in an internal table that we wanted to search. With the current FIND implementation, we would not be able to find text that spanned consecutive lines. In release 7.1 and higher, though, this is easily accomplished by converting the table into a string beforehand:

```
CONCATENATE LINES OF myitab INTO mytext.
FIND mystr IN mytext.
     "search with lines wrapped around
```

If we want to search for all occurrences of some substring, we can use the ALL OCCURRENCES addition that was introduced in release 7.0:

```
FIND ALL OCCURRENCES OF mystr IN mytext
     MATCH COUNT cnt
     MATCH OFFSET off MATCH LENGTH len.
```

In addition to setting sy-subrc, the success of the search is returned in cnt as the number of occurrences found. Obviously, there is not enough space to return more than one match using MATCH OFFSET and MATCH LENGTH, so when using these additions together with ALL OCCURRENCES, the length and position of the *last* substring found will be returned.

To obtain the complete list of occurrences of mystr in mytext we use the RESULTS addition that stores the search result in an internal table of type match_result_tab:

```
DATA restab TYPE match_result_tab.
FIND ALL OCCURRENCES OF mystr IN mytext
     RESULTS restab.
```

The type match_result_tab is predefined in the Data Dictionary and has the line type shown in **Figure 8**.

To process the substrings found, we simply loop over the RESULTS table, preferably assigning a field symbol to each line:

```
FIELD-SYMBOLS <res> LIKE LINE OF restab.
LOOP AT restab ASSIGNING <res>.
   WRITE: / 'Found substring at ',
                  <res>-offset.
ENDLOOP.
```

The FIND statement is one of those statements where trailing blanks are significant for some argument positions. For example:

```
DATA: mystr  TYPE c LENGTH 80,
      mytext LIKE mystr.
FIND mystr IN mytext.
```

will search the complete contents of mytext, including any trailing blanks, even though mytext is a fixed-length field. And just to keep ABAP developers on their toes, this only applies to mytext — any trailing blanks in mystr *are* ignored.

We can also search binary data by using the IN BYTE MODE addition. Most of the above holds similarly true for binary mode, but we need to keep in mind that both offset and length returned are measured in terms of bytes rather than characters:

```
DATA: mybytes TYPE xstring
              VALUE `1234567890ABCD`,
      myhex   TYPE xstring VALUE `90`.
FIND myhex IN mybytes
     MATCH OFFSET off IN BYTE MODE.
  " off now equals 4, not 8!
```

Component	Contents
offset	the offset of the occurrence
length	the length of the occurrence
line	the line of the occurrence
submatches	always empty for SUBSTRING searches

Figure 8 Data Dictionary definition of type match_result_tab

About searching large amounts of text…

Searching large amounts of text is a time-consuming operation that depends on both the size of the text and the size of the part we are searching for.

If *n* is the length of the text to search and *m* is the length of the part to look for, a naïve search algorithm may need up to $m \times n$ character comparisons (assuming, of course, the worst case where the part is found at the last possible occasion). For typical inputs, however, the average number of comparisons required turns out to be proportional to *n* — let's not investigate this result in this article but trust the academics who calculated this for us before.

The `FIND` statement, on the other hand, makes use of a more sophisticated search technique known as the Boyer-Moore algorithm (there is another popular one by Knuth, Morris, and Pratt). This algorithm preprocesses the search string to eliminate redundant comparisons. Typically, this reduces the number of steps required to find the substring down to n / m. Interested readers may find the details of both algorithms in introductory computer science textbooks.

This can be confusing at times thanks to the character representation of binary values.

Searching binary data case-insensitively is obviously nonsensical and thus not supported.

The REPLACE statement

The syntax of the `REPLACE` statement is very similar to that of the `FIND` statement, with one crucial addition:

```
REPLACE mystr IN mytext WITH myrepl.
```

The `WITH` addition tells the `REPLACE` statement what the first occurrence of substring `mystr` found in `mytext` should be replaced with. The `MATCH` additions of the `FIND` statement have their natural counterparts in a bundle of `REPLACEMENT` additions of the `REPLACE` statement.

```
DATA: mytext TYPE c LENGTH 40
             VALUE 'this is my text',
      off TYPE i, len TYPE i.
REPLACE 'my' IN mytext WITH 'your'
        REPLACEMENT OFFSET off
        REPLACEMENT LENGTH len.
```

The big difference in behavior, though, is that all values refer to the text inserted — *not* the text being replaced! So, for example, the previous statement will assign value 4 to variable `len`.

The `REPLACE` statement also sets `sy-subrc` to 0 or 4 based on whether or not at least one replacement was made. There is an additional `sy-subrc` value of 2 that indicates that a replacement was made but the result was truncated in the process. This may only happen when we replace inside a fixed-length field.

As with `FIND`, trailing blanks contained in variable `mytext` are always significant for `REPLACE` whereas trailing blanks contained in `mystr` and `myreplacement` are ignored unless the variables are of type `string`.

Searching with logical expressions

If we are only interested in the information about the presence (or absence) of some substring or character in a given text, a variety of relational operators provides an alternative for searching text conveniently within logical expressions.

Expression	Meaning
s CO *t*	Contains Only: True, if *s* only contains characters from *t*
s CN *t*	Contains Not Only: True, if *s* contains at least one character not in *t*
s CA *t*	Contains Any: True, if *s* contains at least one character from *t*
s NA *t*	Contains Not Any: True, if *s* contains none of *t* (this is the negation of CA)

Figure 9 Single-character operators

Checking for single characters

To check quickly if a specific set of characters is contained in some given text, we can use the *contains any* operator `CA`:

```
IF mytext CA 'aeiou'.
   " contains at least one vowel
ENDIF.
```

There are other operators as well that go well beyond simple searching. **Figure 9** shows the complete list of single-character operators.

Note that despite the suggestive naming, `CN` is not the negation of `CO` (but `NA` is that of `CA`). All operators are case-sensitive and respect trailing blanks for argument *t* but not for argument *s*.

If we need to know the position in the text where the condition of our logical expression failed, we can look into system variable `sy-fdpos`.

```
IF mytext CO 'bcdfghklmnpqrstvwxz'.
  WRITE: / 'Obviously a German word!'.
ELSE.
  WRITE: / 'Found vowel at position',
           sy-fdpos.
ENDIF.
```

If the condition is true, `sy-fdpos` is set to the length of the text.

For binary data, ABAP provides byte-mode operators `BYTE-CO`, `BYTE-CN`, `BYTE-CA`, and `BYTE-NA` that work similarly to their character-mode counterparts.

Searching for substrings

The *contains string* operator `CS` checks exactly that — if some string (or character field) is contained in a given text:

```
IF mytext CS 'solution'.
  WRITE / 'Eureka!'.
ENDIF.
```

The operator searches case-insensitively and sets system variable `sy-fdpos` to the position of the first occurrence of the substring. If the search fails, `sy-fdpos` is set to the length of text.

Again, ABAP also provides a similar byte-mode operator `BYTE-CS` for binary data.

Matching patterns

Patterns are a way of specifying multiple character sequences with a single textual representation. If some text is represented by a given pattern, we say that the pattern *matches* the text. A common task then is to decide if a given pattern matches a given text, or not.

Metacharacter	Meaning
+	Wildcard; any single character
*	Wildcard; any character sequence
#	Escape character; overrides special meaning of subsequent character

Figure 10 Single-character operators

The main idea of all patterns is that they introduce certain wildcards or *metacharacters* that represent more than one character or character sequence (see **Figure 10**). In SAP systems patterns are used, for example, in selection screens and user dialogs to select among multiple objects when pressing the F4 key.

These SAP patterns are also available in the ABAP language. The *contains pattern* operator `CP` checks if a given text matches a certain pattern:

```
CHECK myfilename CP '*.txt'.
```

Execution will continue with the statement following the `CHECK` statement if the value of `myfilename` ends in `.txt`.

If we want to match against `*` or `+`, we need to prepend that symbol with the escape character (`#`). We also use the escape character to override the by-default case-insensitive match operation for that particular character.

In the case of success, `sy-fdpos` contains the offset of the first non-`*` part of the pattern in the text, and the length of the text otherwise.

Finally, we would like to point out that unlike all the other operators introduced here, the `C` part of `CP` resolves into “covers”, not “contains”! If we want to search text using a pattern — for example, to locate HTML tags enclosed in angle brackets — we will have to enclose our pattern in additional asterisks:

```
IF mytext CP '*<*>*'.
  WRITE: / 'Found HTML tag at',
            sy-fdpos.
ENDIF.
```

Fortunately, the semantics of `sy-fdpos` will tell us right where in `mytext` the opening angle bracket is located.

While these kinds of patterns are quite useful, they are still far less powerful than regular expressions, which we will introduce in our next *SAP Professional Journal* article.

Summary

Since text processing is ubiquitous in business applications, knowledge about textual data types and text processing statements is paramount for writing effective and efficient ABAP programs.

In this article, we:

- Introduced the textual data types available in ABAP and highlighted their distinctive features
- Outlined the padding and conversion rules required for working with textual data
- Gave practical advice on selecting appropriate data types
- Discussed the main statements for processing text in ABAP

We hope that you have gained a better understanding of the advantages of each individual data type and received some inspiration for optimizing performance in your next ABAP development project.

Those of you whose appetite for additional information on this subject have been whetted by this article can look forward to our two upcoming articles on regular expressions — these articles will provide both an introduction to this advanced toolset and best-practice examples that are ready to use in your own ABAP programs.

Ralph Benzinger joined SAP in 2003. He is currently a developer in the ABAP language core development group, where he is working on the ABAP compiler and runtime environment. Ralph received a Ph.D. in Computer Science from Cornell University in the US. Prior to joining SAP, he worked as a management consultant in the Business Technology Office of McKinsey & Company. You can reach him at ralph.benzinger@sap.com.

Björn Mielenhausen joined SAP in 1998. He holds a degree in Computer Science from the University of Oldenburg in Germany. He has been working on the ABAP compiler and runtime environment for several years. Since 2003, he has been the Development Manager for the ABAP core team. You can reach him at bjoern.mielenhausen@sap.com.

