3

PL/SQL Data Types

Every PL/SQL constant, variable, parameter, and function return value has a **data type** that determines its storage format and its valid values and operations.

This chapter explains **scalar data types**, which store values with no internal components. For information about **composite data types**, see Chapter 5, "PL/SQL Collections and Records".

A scalar data type can have subtypes. A **subtype** is a data type that is a subset of another data type, which is its **base type**. A subtype has the same valid operations as its base type. A data type and its subtypes comprise a **data type family**.

PL/SQL predefines many types and subtypes in the package `STANDARD` and lets you define your own subtypes.

The PL/SQL scalar data types are:

- The SQL data types
- `BOOLEAN`
- `PLS_INTEGER`
- `BINARY_INTEGER`
- `REF CURSOR`, explained in "Cursor Variables" on page 6-28
- User-defined subtypes

Topics

- SQL Data Types
- BOOLEAN Data Type
- PLS_INTEGER and BINARY_INTEGER Data Types
- SIMPLE_INTEGER Subtype of PLS_INTEGER
- User-Defined PL/SQL Subtypes

See Also:

- "CREATE TYPE Statement" on page 14-73 for information about creating schema-level user-defined data types
- Appendix E, "PL/SQL Predefined Data Types" for the predefined PL/SQL data types and subtypes, grouped by data type family

SQL Data Types

The PL/SQL data types include the SQL data types. For information about the SQL data types, see *Oracle Database SQL Language Reference*—all information there about data types and subtypes, data type comparison rules, data conversion, literals, and format models applies to both SQL and PL/SQL, except as noted here:

- Different Maximum Sizes
- Additional PL/SQL Constants for BINARY_FLOAT and BINARY_DOUBLE
- Additional PL/SQL Subtypes of BINARY_FLOAT and BINARY_DOUBLE

Unlike SQL, PL/SQL lets you declare variables, to which the following topics apply:

- CHAR and VARCHAR2 Variables
- LONG and LONG RAW Variables
- ROWID and UROWID Variables

Different Maximum Sizes

The SQL data types listed in Table 3–1 have different maximum sizes in PL/SQL and SQL.

Table 3–1 Data Types with Different Maximum Sizes in PL/SQL and SQL

Data Type	Maximum Size in PL/SQL	Maximum Size in SQL
`CHAR`[1]	32,767 bytes	2,000 bytes
`NCHAR`[1]	32,767 bytes	2,000 bytes
`RAW`[1]	32,767 bytes	2,000 bytes
`VARCHAR2`[1]	32,767 bytes	4,000 bytes
`NVARCHAR2`[1]	32,767 bytes	4,000 bytes
`LONG`[2]	32,760 bytes	2 gigabytes (GB) - 1
`LONG RAW`[2]	32,760 bytes	2 GB
`BLOB`	128 terabytes (TB)	(4 GB - 1) * *`database_block_size`*
`CLOB`	128 TB	(4 GB - 1) * *`database_block_size`*
`NCLOB`	128 TB	(4 GB - 1) * *`database_block_size`*

[1] When specifying the maximum size of a value of this data type in PL/SQL, use an integer literal (not a constant or variable) whose value is in the range from 1 through 32,767.

[2] Supported only for backward compatibility with existing applications.

Additional PL/SQL Constants for BINARY_FLOAT and BINARY_DOUBLE

The SQL data types `BINARY_FLOAT` and `BINARY_DOUBLE` represent single-precision and double-precision IEEE 754-format floating-point numbers, respectively.

`BINARY_FLOAT` and `BINARY_DOUBLE` computations do not raise exceptions, so you must check the values that they produce for conditions such as overflow and underflow by comparing them to predefined constants (for examples, see *Oracle Database SQL Language Reference*). PL/SQL has more of these constants than SQL does.

Table 3–2 lists and describes the predefined PL/SQL constants for `BINARY_FLOAT` and `BINARY_DOUBLE`, and identifies those that SQL also defines.

Table 3–2 Predefined PL/SQL BINARY_FLOAT and BINARY_DOUBLE Constants[1]

Constant	Description
BINARY_FLOAT_NAN[1]	BINARY_FLOAT value for which the condition IS NAN (not a number) is true
BINARY_FLOAT_INFINITY[1]	Single-precision positive infinity
BINARY_FLOAT_MAX_NORMAL	Maximum normal BINARY_FLOAT value
BINARY_FLOAT_MIN_NORMAL	Minimum normal BINARY_FLOAT value
BINARY_FLOAT_MAX_SUBNORMAL	Maximum subnormal BINARY_FLOAT value
BINARY_FLOAT_MIN_SUBNORMAL	Minimum subnormal BINARY_FLOAT value
BINARY_DOUBLE_NAN[1]	BINARY_DOUBLE value for which the condition IS NAN (not a number) is true
BINARY_DOUBLE_INFINITY[1]	Double-precision positive infinity
BINARY_DOUBLE_MAX_NORMAL	Maximum normal BINARY_DOUBLE value
BINARY_DOUBLE_MIN_NORMAL	Minimum normal BINARY_DOUBLE value
BINARY_DOUBLE_MAX_SUBNORMAL	Maximum subnormal BINARY_DOUBLE value
BINARY_DOUBLE_MIN_SUBNORMAL	Minimum subnormal BINARY_DOUBLE value

[1] SQL also predefines this constant.

Additional PL/SQL Subtypes of BINARY_FLOAT and BINARY_DOUBLE

PL/SQL predefines these subtypes:

- SIMPLE_FLOAT, a subtype of SQL data type BINARY_FLOAT
- SIMPLE_DOUBLE, a subtype of SQL data type BINARY_DOUBLE

Each subtype has the same range as its base type and has a NOT NULL constraint (explained in "NOT NULL Constraint" on page 2-14).

If you know that a variable will never have the value NULL, declare it as SIMPLE_FLOAT or SIMPLE_DOUBLE, rather than BINARY_FLOAT or BINARY_DOUBLE. Without the overhead of checking for nullness, the subtypes provide significantly better performance than their base types. The performance improvement is greater with PLSQL_CODE_TYPE='NATIVE' than with PLSQL_CODE_TYPE='INTERPRETED' (for more information, see "Use Data Types that Use Hardware Arithmetic" on page 12-8).

CHAR and VARCHAR2 Variables

Topics

- Assigning or Inserting Too-Long Values
- Declaring Variables for Multibyte Characters
- Differences Between CHAR and VARCHAR2 Data Types

Assigning or Inserting Too-Long Values

If the value that you assign to a character variable is longer than the maximum size of the variable, an error occurs. For example:

```
DECLARE
  c VARCHAR2(3 CHAR);
```

```
BEGIN
  c := 'abc  ';
END;
/
```

Result:

```
DECLARE
*
ERROR at line 1:
ORA-06502: PL/SQL: numeric or value error: character string buffer too small
ORA-06512: at line 4
```

Similarly, if you insert a character variable into a column, and the value of the variable is longer than the defined width of the column, an error occurs. For example:

```
DROP TABLE t;
CREATE TABLE t (c CHAR(3 CHAR));

DECLARE
  s VARCHAR2(5 CHAR) := 'abc  ';
BEGIN
  INSERT INTO t(c) VALUES(s);
END;
/
```

Result:

```
BEGIN
*
ERROR at line 1:
ORA-12899: value too large for column "HR"."T"."C" (actual: 5, maximum: 3)
ORA-06512: at line 4
```

To strip trailing blanks from a character value before assigning it to a variable or inserting it into a column, use the `RTRIM` function, explained in *Oracle Database SQL Language Reference*. For example:

```
DECLARE
  c VARCHAR2(3 CHAR);
BEGIN
  c := RTRIM('abc  ');
  INSERT INTO t(c) VALUES(RTRIM('abc  '));
END;
/
```

Result:

```
PL/SQL procedure successfully completed.
```

Declaring Variables for Multibyte Characters

The maximum *size* of a `CHAR` or `VARCHAR2` variable is 32,767 bytes, whether you specify the maximum size in characters or bytes. The maximum *number of characters* in the variable depends on the character set type and sometimes on the characters themselves:

Character Set Type	Maximum Number of Characters
Single-byte character set	32,767

Character Set Type	Maximum Number of Characters
n-byte fixed-width multibyte character set (for example, AL16UTF16)	`FLOOR(32,767/n)`
n-byte variable-width multibyte character set with character widths between 1 and *n* bytes (for example, JA16SJIS or AL32UTF8)	Depends on characters themselves—can be anything from 32,767 (for a string containing only 1-byte characters) through `FLOOR(32,767/n)` (for a string containing only *n*-byte characters).

When declaring a `CHAR` or `VARCHAR2` variable, to ensure that it can always hold *n* characters in any multibyte character set, declare its length in characters—that is, `CHAR(n CHAR)` or `VARCHAR2(n CHAR)`, where *n* does not exceed `FLOOR(32767/4)` = 8191.

> **See Also:** *Oracle Database Globalization Support Guide* for information about Oracle Database character set support

Differences Between CHAR and VARCHAR2 Data Types

`CHAR` and `VARCHAR2` data types differ in:

- Predefined Subtypes
- Blank-Padding
- Value Comparisons

Predefined Subtypes The `CHAR` data type has one predefined subtype in both PL/SQL and SQL—`CHARACTER`.

The `VARCHAR2` data type has one predefined subtype in both PL/SQL and SQL, `VARCHAR`, and an additional predefined subtype in PL/SQL, `STRING`.

Each subtype has the same range of values as its base type.

> **Note:** In a future PL/SQL release, to accommodate emerging SQL standards, `VARCHAR` might become a separate data type, no longer synonymous with `VARCHAR2`.

Blank-Padding Consider these situations:

- The value that you assign to a variable is shorter than the maximum size of the variable.
- The value that you insert into a column is shorter than the defined width of the column.
- The value that you retrieve from a column into a variable is shorter than the maximum size of the variable.

If the data type of the receiver is `CHAR`, PL/SQL blank-pads the value to the maximum size. Information about trailing blanks in the original value is lost.

If the data type of the receiver is `VARCHAR2`, PL/SQL neither blank-pads the value nor strips trailing blanks. Character values are assigned intact, and no information is lost.

In Example 3–1, both the `CHAR` variable and the `VARCHAR2` variable have the maximum size of 10 characters. Each variable receives a five-character value with one trailing blank. The value assigned to the `CHAR` variable is blank-padded to 10 characters, and you cannot tell that one of the six trailing blanks in the resulting value was in the

original value. The value assigned to the VARCHAR2 variable is not changed, and you can see that it has one trailing blank.

Example 3–1 CHAR and VARCHAR2 Blank-Padding Difference

```
DECLARE
  first_name  CHAR(10 CHAR);
  last_name   VARCHAR2(10 CHAR);
BEGIN
  first_name := 'John ';
  last_name  := 'Chen ';

  DBMS_OUTPUT.PUT_LINE('*' || first_name || '*');
  DBMS_OUTPUT.PUT_LINE('*' || last_name || '*');
END;
/
```

Result:

```
*John      *
*Chen *
```

Value Comparisons The SQL rules for comparing character values apply to PL/SQL character variables. Whenever one or both values in the comparison have the data type VARCHAR2 or NVARCHAR2, nonpadded comparison semantics apply; otherwise, blank-padded semantics apply. For more information, see *Oracle Database SQL Language Reference*.

LONG and LONG RAW Variables

Note: Oracle supports the LONG and LONG RAW data types only for backward compatibility with existing applications. For new applications:

- Instead of LONG, use VARCHAR2(32760), BLOB, CLOB or NCLOB.
- Instead of LONG RAW, use BLOB.

You can insert any LONG value into a LONG column. You can insert any LONG RAW value into a LONG RAW column. You cannot retrieve a value longer than 32,760 bytes from a LONG or LONG RAW column into a LONG or LONG RAW variable.

You can insert any CHAR or VARCHAR2 value into a LONG column. You cannot retrieve a value longer than 32,767 bytes from a LONG column into a CHAR or VARCHAR2 variable.

You can insert any RAW value into a LONG RAW column. You cannot retrieve a value longer than 32,767 bytes from a LONG RAW column into a RAW variable.

See Also: "Trigger LONG and LONG RAW Data Type Restrictions" on page 9-37 for restrictions on LONG and LONG RAW data types in triggers

ROWID and UROWID Variables

When you retrieve a rowid into a ROWID variable, use the ROWIDTOCHAR function to convert the binary value to a character value. For information about this function, see *Oracle Database SQL Language Reference*.

To convert the value of a ROWID variable to a rowid, use the CHARTOROWID function, explained in *Oracle Database SQL Language Reference*. If the value does not represent a valid rowid, PL/SQL raises the predefined exception SYS_INVALID_ROWID.

To retrieve a rowid into a UROWID variable, or to convert the value of a UROWID variable to a rowid, use an assignment statement; conversion is implicit.

Note:

- UROWID is a more versatile data type than ROWID, because it is compatible with both logical and physical rowids.
- When you update a row in a table compressed with Hybrid Columnar Compression (HCC), the ROWID of the row changes. HCC, a feature of certain Oracle storage systems, is described in *Oracle Database Concepts*.

See Also: *Oracle Database PL/SQL Packages and Types Reference* for information about the DBMS_ROWID package, whose subprograms let you create and return information about ROWID values (but not UROWID values)

BOOLEAN Data Type

The PL/SQL data type BOOLEAN stores **logical values**, which are the Boolean values TRUE and FALSE and the value NULL. NULL represents an unknown value.

The syntax for declaring an BOOLEAN variable is:

```
variable_name BOOLEAN
```

The only value that you can assign to a BOOLEAN variable is a BOOLEAN expression. For details, see "BOOLEAN Expressions" on page 2-38.

Because SQL has no data type equivalent to BOOLEAN, you cannot:

- Assign a BOOLEAN value to a database table column
- Select or fetch the value of a database table column into a BOOLEAN variable
- Use a BOOLEAN value in a SQL statement, SQL function, or PL/SQL function invoked from a SQL statement

You cannot pass a BOOLEAN value to the DBMS_OUTPUT.PUT or DBMS_OUTPUT.PUTLINE subprogram. To print a BOOLEAN value, use an IF or CASE statement to translate it to a character value (for information about these statements, see "Conditional Selection Statements" on page 4-1).

In Example 3–2, the procedure accepts a BOOLEAN parameter and uses a CASE statement to print Unknown if the value of the parameter is NULL, Yes if it is TRUE, and No if it is FALSE.

Example 3–2 Printing BOOLEAN Values

```
CREATE PROCEDURE print_boolean (b BOOLEAN) AUTHID DEFINER
AS
BEGIN
  DBMS_OUTPUT.put_line (
    CASE
```

```
      WHEN b IS NULL THEN 'Unknown'
      WHEN b THEN 'Yes'
      WHEN NOT b THEN 'No'
    END
  );
END;
/
BEGIN
  print_boolean(TRUE);
  print_boolean(FALSE);
  print_boolean(NULL);
END;
/
```

Result:

```
Yes
No
Unknown
```

> **See Also:** Example 2–35, which creates a `print_boolean` procedure that uses an `IF` statement.

PLS_INTEGER and BINARY_INTEGER Data Types

The PL/SQL data types `PLS_INTEGER` and `BINARY_INTEGER` are identical. For simplicity, this document uses `PLS_INTEGER` to mean both `PLS_INTEGER` and `BINARY_INTEGER`.

The `PLS_INTEGER` data type stores signed integers in the range -2,147,483,648 through 2,147,483,647, represented in 32 bits.

The `PLS_INTEGER` data type has these advantages over the `NUMBER` data type and `NUMBER` subtypes:

- `PLS_INTEGER` values require less storage.
- `PLS_INTEGER` operations use hardware arithmetic, so they are faster than `NUMBER` operations, which use library arithmetic.

For efficiency, use `PLS_INTEGER` values for all calculations in its range.

Topics

- Preventing PLS_INTEGER Overflow
- Predefined PLS_INTEGER Subtypes
- SIMPLE_INTEGER Subtype of PLS_INTEGER

Preventing PLS_INTEGER Overflow

A calculation with two `PLS_INTEGER` values that overflows the `PLS_INTEGER` range raises an overflow exception, even if you assign the result to a `NUMBER` data type (as in Example 3–3). For calculations outside the `PLS_INTEGER` range, use `INTEGER`, a predefined subtype of the `NUMBER` data type (as in Example 3–4).

Example 3–3 PLS_INTEGER Calculation Raises Overflow Exception

```
DECLARE
  p1 PLS_INTEGER := 2147483647;
  p2 PLS_INTEGER := 1;
  n NUMBER;
```

```
BEGIN
  n := p1 + p2;
END;
/
```

Result:

```
DECLARE
*
ERROR at line 1:
ORA-01426: numeric overflow
ORA-06512: at line 6
```

Example 3–4 Preventing Example 3–3 Overflow

```
DECLARE
  p1 PLS_INTEGER := 2147483647;
  p2 INTEGER := 1;
  n NUMBER;
BEGIN
  n := p1 + p2;
END;
/
```

Result:

```
PL/SQL procedure successfully completed.
```

Predefined PLS_INTEGER Subtypes

Table 3–3 lists the predefined subtypes of the PLS_INTEGER data type and describes the data they store.

Table 3–3 Predefined Subtypes of PLS_INTEGER Data Type

Data Type	Data Description
NATURAL	Nonnegative PLS_INTEGER value
NATURALN	Nonnegative PLS_INTEGER value with NOT NULL constraint[1]
POSITIVE	Positive PLS_INTEGER value
POSITIVEN	Positive PLS_INTEGER value with NOT NULL constraint[1]
SIGNTYPE	PLS_INTEGER value -1, 0, or 1 (useful for programming tri-state logic)
SIMPLE_INTEGER	PLS_INTEGER value with NOT NULL constraint. For more information, see "SIMPLE_INTEGER Subtype of PLS_INTEGER" on page 3-10.

[1] For information about the NOT NULL constraint, see "NOT NULL Constraint" on page 2-14.

PLS_INTEGER and its subtypes can be implicitly converted to these data types:

- CHAR
- VARCHAR2
- NUMBER
- LONG

All of the preceding data types except LONG, and all PLS_INTEGER subtypes, can be implicitly converted to PLS_INTEGER.

A PLS_INTEGER value can be implicitly converted to a PLS_INTEGER subtype only if the value does not violate a constraint of the subtype. For example, casting the PLS_INTEGER value NULL to the SIMPLE_INTEGER subtype raises an exception, as Example 3–5 shows.

Example 3–5 Violating Constraint of SIMPLE_INTEGER Subtype

```
DECLARE
  a SIMPLE_INTEGER := 1;
  b PLS_INTEGER := NULL;
BEGIN
  a := b;
END;
/
```

Result:

```
DECLARE
*
ERROR at line 1:
ORA-06502: PL/SQL: numeric or value error
ORA-06512: at line 5
```

SIMPLE_INTEGER Subtype of PLS_INTEGER

SIMPLE_INTEGER is a predefined subtype of the PLS_INTEGER data type that has the same range as PLS_INTEGER and has a NOT NULL constraint (explained in "NOT NULL Constraint" on page 2-14). It differs significantly from PLS_INTEGER in its overflow semantics.

If you know that a variable will never have the value NULL or need overflow checking, declare it as SIMPLE_INTEGER rather than PLS_INTEGER. Without the overhead of checking for nullness and overflow, SIMPLE_INTEGER performs significantly better than PLS_INTEGER.

Topics

- SIMPLE_INTEGER Overflow Semantics
- Expressions with Both SIMPLE_INTEGER and Other Operands
- Integer Literals in SIMPLE_INTEGER Range

SIMPLE_INTEGER Overflow Semantics

If and only if all operands in an expression have the data type SIMPLE_INTEGER, PL/SQL uses two's complement arithmetic and ignores overflows. Because overflows are ignored, values can wrap from positive to negative or from negative to positive; for example:

$2^{30} + 2^{30} = 0x40000000 + 0x40000000 = 0x80000000 = -2^{31}$

$-2^{31} + -2^{31} = 0x80000000 + 0x80000000 = 0x00000000 = 0$

For example, this block runs without errors:

```
DECLARE
  n SIMPLE_INTEGER := 2147483645;
BEGIN
  FOR j IN 1..4 LOOP
    n := n + 1;
    DBMS_OUTPUT.PUT_LINE(TO_CHAR(n, 'S9999999999'));
```

```
  END LOOP;
  FOR j IN 1..4 LOOP
   n := n - 1;
   DBMS_OUTPUT.PUT_LINE(TO_CHAR(n, 'S9999999999'));
  END LOOP;
END;
/
```

Result:

```
+2147483646
+2147483647
-2147483648
-2147483647
-2147483648
+2147483647
+2147483646
+2147483645

PL/SQL procedure successfully completed.
```

Expressions with Both SIMPLE_INTEGER and Other Operands

If an expression has both `SIMPLE_INTEGER` and other operands, PL/SQL implicitly converts the `SIMPLE_INTEGER` values to `PLS_INTEGER NOT NULL`.

The PL/SQL compiler issues a warning when `SIMPLE_INTEGER` and other values are mixed in a way that might negatively impact performance by inhibiting some optimizations.

Integer Literals in SIMPLE_INTEGER Range

Integer literals in the `SIMPLE_INTEGER` range have the data type `SIMPLE_INTEGER`. However, to ensure backward compatibility, when all operands in an arithmetic expression are integer literals, PL/SQL treats the integer literals as if they were cast to `PLS_INTEGER`.

User-Defined PL/SQL Subtypes

PL/SQL lets you define your own subtypes. The base type can be any scalar or user-defined PL/SQL data type specifier such as `CHAR`, `DATE`, or `RECORD` (including a previously defined user-defined subtype).

> **Note:** The information in this topic applies to both user-defined subtypes and the predefined subtypes listed in Appendix E, "PL/SQL Predefined Data Types".

Subtypes can:

- Provide compatibility with ANSI/ISO data types
- Show the intended use of data items of that type
- Detect out-of-range values

Topics

- Unconstrained Subtypes
- Constrained Subtypes

- Subtypes with Base Types in Same Data Type Family

Unconstrained Subtypes

An **unconstrained subtype** has the same set of values as its base type, so it is only another name for the base type. Therefore, unconstrained subtypes of the same base type are interchangeable with each other and with the base type. No data type conversion occurs.

To define an unconstrained subtype, use this syntax:

```
SUBTYPE subtype_name IS base_type
```

For information about *subtype_name* and *base_type*, see **subtype_definition** on page 13-14.

An example of an unconstrained subtype, which PL/SQL predefines for compatibility with ANSI, is:

```
SUBTYPE "DOUBLE PRECISION" IS FLOAT
```

In Example 3–6, the unconstrained subtypes `Balance` and `Counter` show the intended uses of data items of their types.

Example 3–6 User-Defined Unconstrained Subtypes Show Intended Use

```
DECLARE
  SUBTYPE Balance IS NUMBER;

  checking_account        Balance(6,2);
  savings_account         Balance(8,2);
  certificate_of_deposit  Balance(8,2);
  max_insured  CONSTANT   Balance(8,2) := 250000.00;

  SUBTYPE Counter IS NATURAL;

  accounts     Counter := 1;
  deposits     Counter := 0;
  withdrawals  Counter := 0;
  overdrafts   Counter := 0;

  PROCEDURE deposit (
    account  IN OUT Balance,
    amount   IN     Balance
  ) IS
  BEGIN
    account  := account + amount;
    deposits := deposits + 1;
  END;

BEGIN
  NULL;
END;
/
```

Constrained Subtypes

A **constrained subtype** has only a subset of the values of its base type.

If the base type lets you specify size, precision and scale, or a range of values, then you can specify them for its subtypes. The subtype definition syntax is:

```
SUBTYPE subtype_name IS base_type
  { precision [, scale ] | RANGE low_value .. high_value } [ NOT NULL ]
```

Otherwise, the only constraint that you can put on its subtypes is `NOT NULL` (described in "NOT NULL Constraint" on page 2-14):

```
SUBTYPE subtype_name IS base_type [ NOT NULL ]
```

> **Note:** The only base types for which you can specify a range of values are `PLS_INTEGER` and its subtypes (both predefined and user-defined).

> **See Also:** Syntax diagram "subtype_definition ::=" on page 13-11 and semantic description "subtype_definition" on page 13-14.

In Example 3–7, the constrained subtype `Balance` detects out-of-range values.

Example 3–7 User-Defined Constrained Subtype Detects Out-of-Range Values

```
DECLARE
  SUBTYPE Balance IS NUMBER(8,2);

  checking_account  Balance;
  savings_account   Balance;

BEGIN
  checking_account := 2000.00;
  savings_account  := 1000000.00;
END;
/
```

Result:

```
DECLARE
*
ERROR at line 1:
ORA-06502: PL/SQL: numeric or value error: number precision too large
ORA-06512: at line 9
```

A constrained subtype can be implicitly converted to its base type, but the base type can be implicitly converted to the constrained subtype only if the value does not violate a constraint of the subtype (see Example 3–5).

A constrained subtype can be implicitly converted to another constrained subtype with the same base type only if the source value does not violate a constraint of the target subtype.

In Example 3–8, the three constrained subtypes have the same base type. The first two subtypes can be implicitly converted to the third subtype, but not to each other.

Example 3–8 Implicit Conversion Between Constrained Subtypes with Same Base Type

```
DECLARE
  SUBTYPE Digit        IS PLS_INTEGER RANGE 0..9;
  SUBTYPE Double_digit IS PLS_INTEGER RANGE 10..99;
  SUBTYPE Under_100    IS PLS_INTEGER RANGE 0..99;

  d   Digit        :=  4;
  dd  Double_digit := 35;
```

```
  u   Under_100;
BEGIN
  u := d;   -- Succeeds; Under_100 range includes Digit range
  u := dd;  -- Succeeds; Under_100 range includes Double_digit range
  dd := d;  -- Raises error; Double_digit range does not include Digit range
END;
/
```

Result:

```
DECLARE
*
ERROR at line 1:
ORA-06502: PL/SQL: numeric or value error
ORA-06512: at line 12
```

> **See Also:** "Formal and Actual Subprogram Parameters" on page 8-9 for information about subprogram parameters of constrained data types

Subtypes with Base Types in Same Data Type Family

If two subtypes have different base types in the same data type family, then one subtype can be implicitly converted to the other only if the source value does not violate a constraint of the target subtype. (For the predefined PL/SQL data types and subtypes, grouped by data type family, see Appendix E, "PL/SQL Predefined Data Types".)

In Example 3–9, the subtypes `Word` and `Text` have different base types in the same data type family. The first assignment statement implicitly converts a `Word` value to `Text`. The second assignment statement implicitly converts a `Text` value to `Word`. The third assignment statement cannot implicitly convert the `Text` value to `Word`, because the value is too long.

Example 3–9 Implicit Conversion Between Subtypes with Base Types in Same Family

```
DECLARE
  SUBTYPE Word IS CHAR(6);
  SUBTYPE Text IS VARCHAR2(15);

  verb       Word := 'run';
  sentence1  Text;
  sentence2  Text := 'Hurry!';
  sentence3  Text := 'See Tom run.';

BEGIN
  sentence1 := verb;  -- 3-character value, 15-character limit
  verb := sentence2;  -- 5-character value, 6-character limit
  verb := sentence3;  -- 12-character value, 6-character limit
END;
/
```

Result:

```
DECLARE
*
ERROR at line 1:
ORA-06502: PL/SQL: numeric or value error: character string buffer too small
ORA-06512: at line 13
```