

How To Convert Between Built-in Types and Perform I/O

Occasionally you need to write code that converts a value from one of the built-in types (*Boolean*, *Character*, *Integer*, *Real*, or *Text*) to another. For example, you might have:

- an *Integer* object whose value you need to add to a *Real*-valued expression; or
- a *Character* object for which you need the ASCII code, which is an *Integer* value; or
- an *Integer* expression whose value you need to display in an `OUTPUT_TEXT` *Control*, but this requires that you convert it to a *Text* value.

In Resolve/C++, there are no *implicit* conversions of values from one type to another. You always have to make any such conversion *explicit* by invoking a function operation that takes as an argument a value (of the type to be converted from), and returns the converted value (of the type to be converted to).

There is a simple picture (Figure 1) that shows which conversions may be done using built-in conversion functions, and a simple rule for remembering the name of the conversion function to use. But there is no easy way to remember the preconditions for some of the conversion functions—they are inherently complicated. So there are also built-in functions to test the preconditions of the conversion functions.

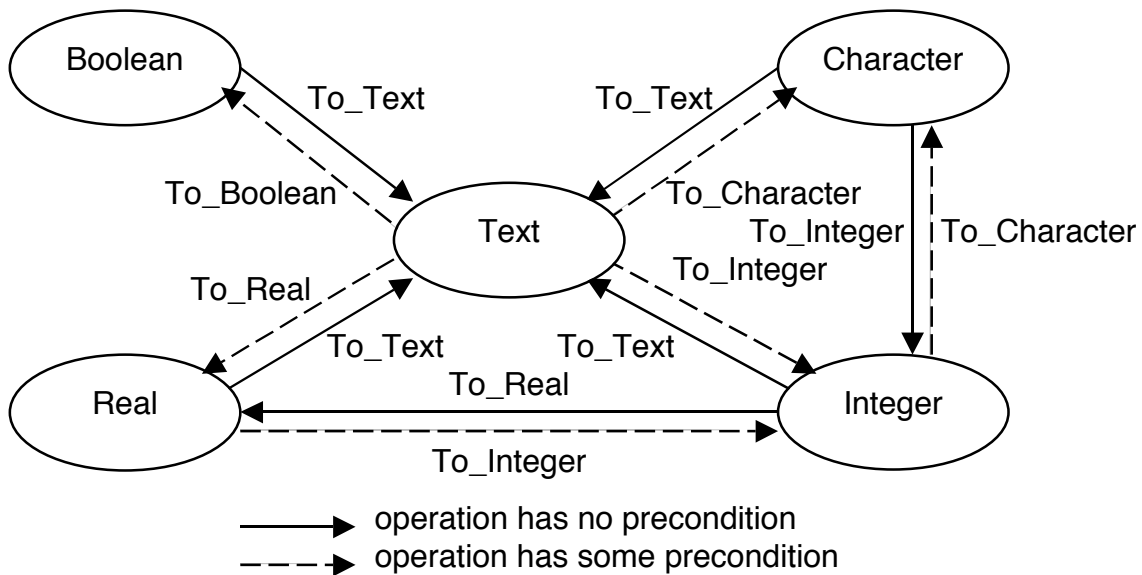


Figure 1: Possible conversions between built-in types

The purposes of this document are:

- to state the rules for naming and calling the conversion functions and the functions that check their preconditions;

- briefly and informally, to explain the uses and effects of the conversion functions; and
- briefly and informally, to summarize the preconditions for the conversion functions, where applicable.

The examples use the following objects:

```

object Boolean b;
object Character c;
object Integer i;
object Real r;
object Text t;
object Character_IStream in;
object Character_OStream out;

```

Calling Conversion Functions And Precondition Checking Functions

The rule for remembering the name of the function that converts a value from one built-in type to another is:

If there is a conversion function from built-in type *X* to built-in type *Y*, for some *X* and *Y*, then its name is *To_Y*. It takes as an argument a value of type *X* and returns a value of type *Y*.

The rule for remembering the name of the precondition checking function, which tells you whether you may convert a value from one built-in type to another, is:

If there is a precondition on the conversion function *To_Y* for some argument type *X*, then there is a precondition checking function for *To_Y*. Its name is *Can_Convert_To_Y*. It takes as an argument a value of type *X* and returns a *Boolean* value: *true* if the conversion *To_Y* will succeed for that argument, *false* if it will not.

You invoke a conversion function or a precondition checking function as a global function, i.e., using traditional function-call syntax as illustrated in the examples that follow. You do *not* invoke these functions using “object.operation” syntax.

Informal Specifications Of The Conversion Functions

The following examples illustrate the effects of all the conversion functions. We begin with the easy ones (those with no preconditions) and move on to the trickier ones (those with preconditions).

To_Text (Boolean)**To_Text (Character)****To_Text (Integer)****To_Text (Real)**

The *To_Text* function returns a *Text* value which is the value you'd probably expect if you were writing down the value being converted. There is no precondition on this function.

One little surprise might bite you here. For a *Real* argument, the resulting *Text* value always looks like a number rounded to six digits to the right of the decimal point in ordinary decimal (i.e., not scientific) notation. However, you have some control over this.

- If you add a second argument when converting *To_Text* from a *Real*, then it should be an *Integer* value that is the number of digits you want after the decimal point.
- If you also add a third argument, then it should be a *Boolean* value that is *true* iff you want scientific notation.

Call	Return Value
<code>To_Text (true)</code>	"true"
<code>To_Text (false)</code>	"false"
<code>To_Text ('a')</code>	"a"
<code>To_Text ('\n')</code>	"\n"
<code>To_Text (-147)</code>	"-147"
<code>To_Text (13)</code>	"13"
<code>To_Text (3.14)</code>	"3.140000"
<code>To_Text (-0.00563)</code>	"-0.005630"
<code>To_Text (-0.00563, 3)</code>	"-0.006"
<code>To_Text (-0.00563, 3, true)</code>	"-5.630E-03"

To_Integer (Character)

The *To_Integer* function, when given a *Character* value, returns the ASCII code of that *Character* (a number between 0 and 127, inclusive). There is no precondition on this function.

Watch out for a common pitfall here. An example: *To_Integer* ('8') is not equal to 8. That's because the ASCII code for '8' is 56, not 8. This also illustrates that there is a big difference between the *Character* value '8' and the *Text* value "8"; the latter is not a single character but a

string of characters that happens to contain exactly one character. It happens that *To_Integer* ("8") is equal to 8; see the explanation of *To_Integer (Text)* later.

Call	Return Value
<i>To_Integer</i> ('a')	97
<i>To_Integer</i> ('\n')	10

To_Real (Integer)

The *To_Real* function, when given an *Integer* value, returns the same number simply converted to a *Real* value. There is no precondition on this function.

Call	Return Value
<i>To_Real</i> (13)	13.0
<i>To_Real</i> (-2)	-2.0

To_Boolean (Text)

The *To_Boolean* function must be given a *Text* value which is a string of zero or more white space characters (' ', '\t', or '\n'), followed by a string which is either exactly "true" or exactly "false", followed by a string of zero or more white space characters. It returns *true* in the former case and *false* in the latter. Any other *Text* value as the argument violates the function's precondition, leading to this violated assertion message:

```
Violated assertion at:

File:
../../RESOLVE_Foundation/Conversion_Operations/
  Conversion_Operations.cpp
Line #: 512
Operation: To_Boolean

Violated assertion is:
=====
<OK_Text>      ::= <White_Space> {"true"|"false"} <White_Space>
<White_Space> ::= {' ', '\t', '\n'}*
=====
t is in language of OK_Text
=====
```

You can check the precondition for *To_Boolean* with the function *Can_Convert_To_Boolean*, as illustrated in the table that follows.

Call	Return Value
To_Boolean ("true")	true
To_Boolean ("\t false \n \t ")	false
Can_Convert_To_Boolean ("true")	true
Can_Convert_To_Boolean ("1")	false

To_Character (Text)

To_Character (Integer)

The *To_Character* function, when given a *Text* value, returns the first character in its argument. An empty string as the argument violates the function's precondition, leading to this violated assertion message:

```
Violated assertion at:

File:
../../RESOLVE_Foundation/Conversion_Operations/
  Conversion_Operations.cpp
Line #: 539
Operation: To_Character

Violated assertion is:
t != empty_string
```

The *To_Character* function, when given an *Integer* value between 0 and 127 inclusive, returns the character with that ASCII code. Any other *Integer* argument violates the function's precondition, leading to this violated assertion message:

```
Violated assertion at:

File:
../../RESOLVE_Foundation/Conversion_Operations/
  Conversion_Operations.cpp
Line #: 528
Operation: To_Character

Violated assertion is:
0 <= i <= 127
```

You can check the precondition for *To_Character* with the function *Can_Convert_To_Character*, as illustrated in the table that follows.

Call	Return Value
To_Character ("x")	'x'
To_Character ("nix")	'n'
To_Character (97)	'a'
To_Character (10)	'\n'
Can_Convert_To_Character ("nix")	true
Can_Convert_To_Character ("")	false
Can_Convert_To_Character (97)	true
Can_Convert_To_Character (-46)	false

To_Integer (Text)

To_Integer (Real)

The *To_Integer* function, when given a *Text* value, returns the *Integer* value which you'd think it should if its argument is interpretable as an integer. Phrased loosely, suppose *t* consists of a string of zero or more white space characters, followed by a string which could have been the return value of *To_Text* (*i*) for some *Integer* value *i*, followed by a string of zero or more white space characters. Then *To_Integer* (*t*) returns *i*.

A *Text* argument that doesn't follow the rule described above violates the function's precondition, leading to this violated assertion message:

Violated assertion at:

File:

```
../../RESOLVE_Foundation/Conversion_Operations/
  Conversion_Operations.cpp
```

Line #: 570

Operation: To_Integer

Violated assertion is:

```
=====
<OK_Text>      ::= <White_Space> ['+'|'-'] <Digits> <White_Space>
<White_Space> ::= {' ', '\t', '\n'}*
<Digits>       ::= {'0', '1', '2', '3', '4', '5', '6', '7', '8', '9'}+
-----
t is in language of OK_Text and
-2147483648 <= TO_INTEGER(t) <= 2147483647
=====
```

The *To_Integer* function, when given a *Real* value whose integer part is within the bounds for an *Integer*, returns this integer part. The number is “truncated”, i.e., the fractional part is simply discarded, so “rounding” is always toward zero. Any other *Real* argument violates the function's precondition, leading to this violated assertion message:

Violated assertion at:

File:
 ../../RESOLVE_Foundation/Conversion_Operations/
 Conversion_Operations.cpp
 Line #: 559
 Operation: To_Integer

Violated assertion is:
 -2147483649.0 < r < 2147483648.0

You can check the precondition for *To_Integer* with the function *Can_Convert_To_Integer*, as illustrated in the table that follows.

Call	Return Value
To_Integer (" \t\n 123 \t")	123
To_Integer ("-4567")	-4567
To_Integer (69.4)	69
To_Integer (-33.75)	-33
Can_Convert_To_Integer ("-4567")	true
Can_Convert_To_Integer ("blah")	false
Can_Convert_To_Integer (69.4)	true
Can_Convert_To_Integer (2E99)	false

To_Real (Text)

The *To_Real* function, when given a *Text* value, returns the *Real* value which you'd think it should if its argument is interpretable as a real number. Phrased loosely, suppose *t* consists of a string of zero or more white space characters, followed by a string which could have been the return value of *To_Text* (*r*) for some *Real* value *r*, followed by a string of zero or more white space characters. Then *To_Real* (*t*) returns *r*.

A *Text* argument that doesn't follow the rule described above violates the function's precondition, leading to this violated assertion message:

Violated assertion at:

File:
 ../../RESOLVE_Foundation/Conversion_Operations/
 Conversion_Operations.cpp
 Line #: 594
 Operation: To_Real

Violated assertion is:

```
=====
<OK_Text>      ::= <White_Space> <Number> <White_Space>
```

```

<White_Space> ::= {' ', '\t', '\n'}*
<Number>      ::= ['+'|'-'] <Digits> ['.' <Digits>]
               [{ 'e'|'E' } ['+'|'-'] <Digits>]
<Digits>      ::= {'0','1','2','3','4','5','6','7','8','9'}+
-----
t is in language of OK_Text and
-1.7976931348623157e+308 <= TO_REAL(t) <=
 1.7976931348623157e+308
=====

```

You can check the precondition for *To_Real* with the function *Can_Convert_To_Real*, as illustrated in the table that follows.

Call	Return Value
To_Real (" \t 123.45 ")	123.45
To_Real ("-45E-3")	-0.045
Can_Convert_To_Real ("-45E-3")	true
Can_Convert_To_Real ("eh?")	false

Input And Output Of Built-in Types

The centrality of *Text* in Figure 1 simplifies the explanation of Resolve/C++ output and input. You can understand output and input of the built-in types using *Character_OStream* operator << and *Character_IStream* operator >> by understanding conversion to and from *Text*.

Opening and Closing Input/Output Streams

When you declare an input/output stream, you can connect it to an *external* source/sink for characters—standard input/output, a file, or a socket—or an *internal* source/sink. You must “open” a stream by designating one of these before you can read/write with it. Remember to “close” it when you are finished using it.

The most common I/O situation is a connection to standard input/output: by default, the keyboard/screen, or “redirected” input/output from/to a file or pipe if specified by the end-user (e.g., on the command line that invokes the program). Here is an example:

```

object Character_IStream in;
object Character_OStream out;
...
in.Open_External ("");
out.Open_External ("");
...
// code to read/write, as explained below
...
in.Close_External ();
out.Close_External ();
...

```

To read/write from/to files with names known at the time the program is compiled, you can simply provide the file name as a *Text* argument to *Open_Internal*. The *Text* value "" (empty string) denotes standard input/output, as illustrated above. To read input from the file named input.txt, and to write output to the file named output.html, for example, you might write code like this:

```

object Character_IStream in;
object Character_OStream out;
...
in.Open_External ("input.txt");
out.Open_External ("output.html");
...
// code to read/write, as explained below
...
in.Close_External ();
out.Close_External ();
...

```

Reading/writing from/to a socket connection with another program is supported but is not discussed in the Software Component Engineering courses. On-line code examples of reader/write and client/server processes that use this feature should be consulted for details.

An internal source/sink of characters acts just like an external source/sink except that no input/output takes place between the computer and a peripheral device such as the keyboard, a disk, the network, or the screen. All characters are simply stored “internally” to be transferred from place to place within a single program. The primary use for this feature—and the only one illustrated here—is to construct a complex *Text* value. This approach dramatically simplifies the construction of large *Text* values that cannot be created by assigning a single literal quoted string to a *Text* object. For example:

```

object Text t;
object Character_OStream out;
...
out.Open_Internal ();
// code to write characters to out, as explained below
out.Close_Internal (t);
// t's value equals the entire string written to out, since out was opened

```

Output

Output is very simple once an output stream is open. It works as if *Character_OStream* operator << knows only about *Text*, so you can imagine that every other type has to be converted to *Text* in order to be output. Suppose you write an *Integer* expression to a *Character_OStream* object, e.g.:

```
out << "i + 1 = " << i + 1;
```

You can imagine that what really happens is this: *To_Text* of the *Integer* expression is output. In other words, the above statement has the same effect as:

```
out << "i + 1 = " << To_Text (i + 1);
```

The same rule applies for understanding output of the other built-in types.

Input

Input works in a similar fashion. It is as if *Character_IStream* operator `>>` knows only about *Text*, so you can imagine that every other type has to be converted from *Text* as it is input. But input is more complicated than output. For while you can always read input into a *Text* object without fear of an error (unless you're at the end of the *Character_IStream*), it is not always possible to convert that *Text* value to the appropriate type. This means that reading input into objects of the other built-in types will result in run-time errors if the *Character_IStream* doesn't deliver exactly the right characters at exactly the right times.

Suppose you read from a *Character_IStream* object into an *Integer* object. You can imagine that what happens is as follows. First, the entire next line of input—the string of all characters up through the next newline ('\n') character—is read into a temporary *Text* object. (The newline is simply discarded; it does not make it into the *Text* object.) Then the *Text* object's value is converted to an *Integer* value using *To_Integer* and the return value is assigned to the *Integer* object. Of course, this means that the input line must make sense if you try to interpret it as an *Integer* value; i.e., the precondition of *To_Integer* must be satisfied for this line of input. So, if your program executes this statement:

```
in >> i;
```

the effect is the same as this:

```
{
    object Text temp;
    in >> temp;
    i = To_Integer (temp);
}
```

If you are getting input from a file you know will contain the right information, then this code works. But if you are getting input from an end user who might type anything, so *To_Integer* (*temp*) might fail, then a better approach is actually to read input into a *Text* object and convert it yourself. Here is a flexible (and rather pesky) procedure you might write for doing this:

```
procedure_body Get_Integer_Input (
    alters Character_OStream out,
    preserves Text prompt,
    preserves Text error_message,
    alters Character_IStream in,
    produces Integer& i
)
{
    object Text t;
    out << prompt;
    in >> t;
    if (Can_Convert_To_Integer (t))
    {
        i = To_Integer (t);
    }
    else
    {
        out << error_message;
        Get_Integer_Input (out, prompt, error_message, in, i);
    }
}
```