

Chapter 20 – Runtime Error Trapping

As you have worked through the examples and created your own programs you have seen errors that happen while the program is running. These errors are called "runtime errors". BASIC-256 includes a group of special commands that allow your program to recover from or handle these errors.

You may already have seen programs that throw or display errors when they are running. They often occur when an invalid mathematical operation happens or when an unassigned variable is used. In Program 121 you see a program that works most of the time but will error and quit running if the denominator is zero.

```
1  # divider.kbs
2  # simple division
3
4  print "divide two numbers"
5  while true
6      input "numerator?", n
7      input "denominator?", d
8      q = n/d
9      print "quotient is  " + q
10 end while
```

Program 121: Simple Division Program That May Error

```
divide two numbers
numerator?6
denominator?9
quotient is  0.6666667
numerator?5
denominator?2
quotient is  2.5
numerator?9
```

```
denominator?0
ERROR on line 8: Division by zero.
```

Sample Output 121: Simple Division Program That May Error

Try a Statement and Catch an Error:

The **try/catch/end try** block is structured so that if a trappable runtime error occurs in the code between the **try** and the **catch**, the code immediately following the **catch** will be executed. The following example shows the simple division program now catching the division by zero error.

```
1  # trycatch.kbs
2  # simple try catch
3
4  print "divide two numbers"
5  while true
6      input "numerator?", n
7      input "denominator?", d
8      try
9          q = n/d
10         print "quotient is " + q
11     catch
12         print "I can't divide " + d + " into " + n
13     end try
14 end while
```

Program 122: Simple Division Program That Catches Error

```
divide two numbers
numerator?5
denominator?6
quotient is 0.8333333
numerator?99
denominator?0
I can't divide 0 into 99
numerator?4
```

```
denominator?3  
quotient is 1.3333333  
numerator?
```

Sample Output 122: Simple Division Program That Catches Error



New Concept

```
try  
    statement(s) to try  
catch  
    statement(s) to execute if an error occurs  
end try
```

The **try/catch/end try** ...

Trapping errors, when you do not mean too, can cause problems and mask other problems with your programs. Error trapping should only be used when needed and disabled when not.

Finding Out Which Error:

Sometimes just knowing that an error happened is not enough. There are functions that will return the error number (**lasterror**), the line where the error happened in the program (**lasterrorline**), a text message describing the error (**lasterrormessage**), and extra command specific error messages (**lasterrorextra**).

```
1 # trap.kbs  
2 # error trapping with reporting  
3  
4 try  
5     print "z = " + z
```

```
6 catch
7     print "Caught Error"
8     print "    Error = " + lasterror
9     print "    On Line = " + lasterrorline
10    print "    Message = " + lasterrormessage
11 end try
12 print "Still running after error"
```

Program 123: Try/Catch - With Messages

```
Caught Error
Error = 12
On Line = 4
Message = Unknown variable z
Still running after error
```

Sample Output 123: Try/Catch - With Messages



New Concept

`lasterror` or `lasterror()`
`lasterrorline` or `lasterrorline()`
`lasterrormessage` or `lasterrormessage()`
`lasterrorextra` or `lasterrorextra()`

The four "last error" functions will return information about the last trapped error. These values will remain unchanged until another error is encountered.

lasterror	Returns the number of the last trapped error. If no errors have been trapped this function will return a zero. See Appendix G: Errors and Warnings for a complete list of trappable errors.
lasterrorline	Returns the line number, of the program, where the last error was trapped.
lasterrormessage	Returns a string describing the last error.
lasterrorextra	Returns a string with additional error information. For most errors this function will not return any information.

Type Conversion Errors

BASIC-256 by default will return a zero when it is unable to convert a string to a number. You may have seen this previously when using the **inputinteger** and **inputfloat** statements. This will also happen when the **int()** and **float()** functions convert a string to a number.

You may optionally tell BASIC-256 to display a trappable warning or throw an error that stops execution of your program. You can change this setting in the "Preferences" dialog, on the User tab.

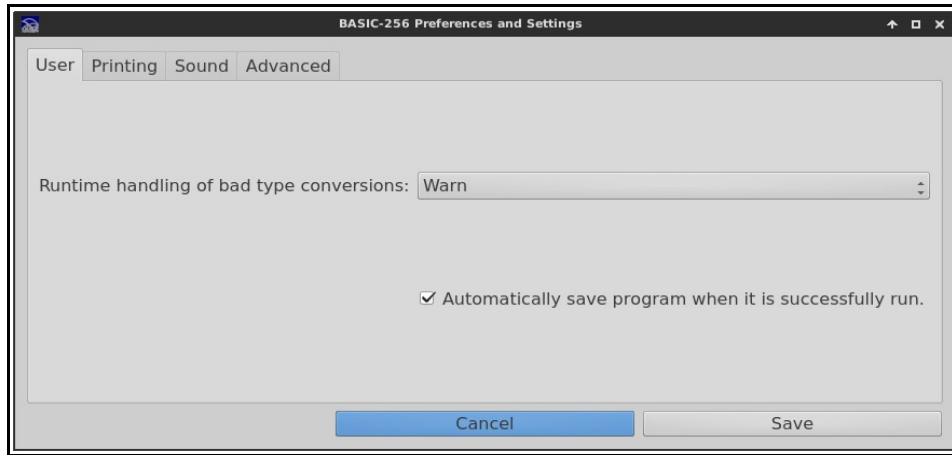


Illustration 38: Preferences - Type Conversion Ignore/Warn/Error

```
1 # inputnumber.kbs
2
3 input "enter a number> ",a
4 print a
```

Program 124: Type Conversion Error

Program run with the errors "Ignored".

```
enter a number> foo
0
```

Sample Output 124: Type Conversion Error - Ignored (Default)

Program run with the "Warning" enabled. Notice that the program continues running but displays a message. The **try/catch/end try** statements will catch the warning so that you may display a custom message or do special processing.

```
enter a number> sdfsd
```

```
WARNING on line 3: Unable to convert string to  
number, zero used.  
0
```

Sample Output 124: Type Conversion Error - Warning

This third example had the property set to "Error". When an invalid type conversion happens an error is displayed and program execution stops. This error is trappable with the **try/catch/end try** statements.

```
enter a number> abcd  
ERROR on line 3: Unable to convert string to  
number.
```

Sample Output 124: Type Conversion Error - Error

Creating An Error Trapping Routine:

There is a second way to trap run-time errors, by using an error trapping subroutine. When this type of error trapping is turned on, with the **onerror** statement, the program will call a specified subroutine when an error occurs. When the error trap returns the program will automatically continue with the next line in the program.

If we look at Program 125 we will see that the program calls the subroutine when it tries to read the value of z (an undefined variable). If we try to run the same program with line one commented out or removed the program will terminate when the error happens.

```
1 # simpletrap.kbs  
2 # simple error trapping  
3  
4 onerror trap  
5  
6 print "z = " + z
```

```

7   print "Still running after error"
8   end
9
10  subroutine trap()
11      print "I trapped an error."
12  end subroutine

```


Program 125: Simple Runtime Error Trap

```

I trapped an error.
z = 0
Still running after error

```

Sample Output 125: Simple Runtime Error Trap

 <p>New Concept</p>	<p>onerror label</p>
	<p>Create an error trap that will automatically jump to the subroutine at the specified label when an error occurs.</p>

You may use the **lasterror**, **lasterrorline**, **lasterrormessage**, and **lasterrorextra** functions within your error trap subroutine to display any messages or do any processing you wish to do. Additionally you may not define an **onerror** trap inside a **try/catch**.

Turning Off Error Trapping Routine:

Sometimes in a program we will want to trap errors during part of the program and not trap other errors. The **offerror** statement turns error trapping off. This causes all errors encountered to stop the program.


```
1  # trapoff.kbs
2  # error trapping with reporting
3
4  onerror errortrap
5  print "z = " + z
6  print "Still running after first error"
7
8  offerror
9  print "z = " + z
10 print "Still running after second error"
11 end
12
13 subroutine errortrap()
14     print "Error Trap - Activated"
15 end subroutine
```


Program 126: Turning Off the Trap

```
Error Trap - Activated
z = 0
Still running after first error
ERROR on line 6: Unknown variable
```

Sample Output 126: Turning Off the Trap

Exercises:

 <p>Word Search</p>	<pre> e u q r l w f e p j x s p w n c p g u b i r r h f j w w w o c p b l a s t e r r o r e x t r a p q e e s v w j l p g a m w l o q t a n n s r q o i i t m r a n o r f x i d e o u c a t c h t e y y h z r l t m r f k o s k v r i q o i b m r r r r s i e f b r f x l f x o z o y o e l b b i o a y k m f z o r r q r t s k e r a z a h l e i r y r p r s f g y m i l i l n r e j f e p e a n r l a q c m t q r k o g t l t l u u r e u k z b b o u f l s g s t j m s u h l a r x r m v w a q a l u b z r l h a l k p a r t l n l </pre> <p>catch, endtry, error, lasterror, lasterrorextra, lasterrorline, lasterrormessage, offerror, onerror, trap, try</p>
---	--

 <p>Problems</p>	<p>1. Set the "runtime handling of bad type conversion" "Preference" to "warn" or "Error" and write a simple program that asks the user to enter a number. If the user enters something that is not a number, trap the warning/error and ask again.</p> <pre> enter a number> gdf2345 bad entry. try again. enter a number> fdg545 bad entry. try again. enter a number> 43fdgdf bad entry. try again. </pre>
--	--

```
enter a number> 22  
You entered 22
```

2. Take the logic you just developed in Problem 1 and create a function that takes one argument, the prompt message, repeatedly asks the user for a number until they enter one, and returns the user's numeric entry.
3. Write a program that causes many errors to occur, trap and them. Be sure to check out Appendix G: Errors and Warnings for a complete list