

# Birthstone Language Reference

Copyright (C) 2010 by Robert Tolliver (Robb.Tolli@gmail.com)

This document is part of the Birthstone documentation.

See the file doc\_license.txt for copying conditions.

## Variables

### Data Types

The data types in birthstone are Number (floating point), Boolean, String, List, and None. Birthstone is loosely typed, so types will be implicitly converted according the context. To explicitly convert between types, read "type conversion" in the "Tips & Tricks" section of this document. In general, the type of the result of a binary operation (like  $a + b$ ) will be the type of the first operand. The exception is the comparison and logical operators, which always return booleans.

The unary type operator will return a string representation of the type of it's operand. For example:

```
type "abc";           # returns "String"
type [1, "aa", true] # returns "List"
type 0;               # returns "Number"
type true;            # returns "Boolean"
type x;               # returns "None" if x is undeclared,
                     # or returns one of the above types if x has been declared
```

### Lists

Lists are declared using a comma separated list of values enclosed by brackets (`[]`). The `length` keyword follow by a list literal or variable will give the length of the list. To access an element in the list use the name of the list followed by an index in bracket. An error will be given if the number is out of range. The valid values for a list index are `-(length list)` to `(length list)-1`. If the index is negative, it counts from the end of the list (`list[-1]` is the end of the list and `list[-(length list)]` is the beginning of the list) . The idea of negative indexing was taken from Python.

```
evens := [2,4,6,8]; # initialize the evens list
print length evens; # prints 4
print evens[0];     # prints 0
print evens[3];     # prints 8
print evens[-1];    # prints 8
print evens[-4];    # prints 2
print evens[-5];    # Error: out of range
print evens[4];     # Error: out of range
```

## Declaring, Initializing, and Assigning Variables

To declare and initialize a variable use the initialize operator ( `:=` ).

```
number := 0;
string := "";
boolean:= false;
```

The example above declares the variables `number`, `bool`, and `string`, and initializes them to `0`, `""`, and `false`, respectively.

A variable may be re-declared any number of times -- the variable will be bound to a new type and value.

A variable may also be re-assigned using the assignment operator ( `=` ). When a variable is re-assigned (as opposed to re-declared), the variable retains it's old type. Here's an example:

```
x := "aa";
y := "bb";
x := 1;
y = 2;
```

In this example, `x` is first initialized to `"aa"`, then it is **re-declared** (re-bound) to the value `1` (a number). The variable `y` is first initialized to `"bb"`, and then **re-assigned** to the value `2`. Since `y` is a string, it stays a string when it is reassigned, so the number `2` is converted to the string `"2"`, and `y` ends up with a value of `"2"` (as a string).

## Input and Output

### Writing to the console

Birthstone has two commands to write to the console: `print` and `write`. The difference is that `print` outputs a newline character after printing the data, while `write` prints just the data without appending a newline character. Here's an example of the classic "Hello, world" program written in birthstone:

```
write "Hello, ";
print "world!";
```

This prints out "Hello, world!" to the console, followed by a newline.

Most type of variables, literals, and expressions can be printed using the `print` or `write` command. The one exception is that printing a function variable will result in an error.

### Reading from the console

To read input from the console, use the `read` command. The syntax for the `read` command is `read variableName`. If you are reading into a variable that has already been declared, the input will be the same type as the variable. If you are reading into a variable that has not been previously declared, a string is read in. Here's an example:

```
num := 0;
str := "";
bool:= false;
```

```

write "Enter a number: ";
read num;

write "Enter a string: ";
read str;

write "Enter a boolean: ";
read bool;

write "Enter a string: ";
read var;

```

This example reads a number into `num`, a string into `str`, a boolean into `bool`, and a string into `var` (since `var` hasn't been declared previously).

## Formatting strings

Formatting strings in Birthstone involves a formatting string, followed any number of arguments each proceeded by the format operator (%). For example:

```

length := 4;
width  := 5;
message := 'The area of a %.2f x %.2f rectangle is %.2f.' % length
          % width % (length * width);

```

In this example, the first “%.2f” formats the first argument (`length`), the second “%.2f” formats the second argument (`width`), and the third “%.2f” formats the third argument (`length*width`). The result is that the string `message` is initialized with the value “The area of a 4.00 x 5.00 rectangle is 20.00.”

The Birthstone format string supports all of the standard C/C++ `printf` format flags plus some additional format flags. Formatting strings are implemented using the [Boost C++ format library](#), which implements all the format flags of [Unix98 open-group's printf](#). You can see those websites for details about which format flags can be used.

## File I/O

File I/O in Birthstone is a work in progress. Once implemented, file I/O will work something like this:

```

inFile := open "input.txt"; # open the input file
outFile := open "output.txt"; # open the output file
num := 0;
fread inFile str; # read a word from inFile into str
fread inFile num; # read a number from inFile into num
fwrite outFile num; # write the number to the output file
fprintf outFile "hello"; # write "hello" and a newline to outFile
close inFile; # close the input file
close outFile; # close the output file

```

# Conditionals

A conditional statement in Birthstone consists of an `if` statement, any number of `elif` (else if) statements, and optionally an `else` statement. **Note:** *elif, elsif, and elseif (no space) are all supported for the else if part of the conditional.* The body of each part of the conditional can either be either a single statement or a block (group of statements enclosed in braces `{}`). Below is the syntax for each loop (the items in angle brackets are placeholders for code and the square brackets `[]` indicate optional components).

```
if (<condition1>)
{
    ...
}
[elif (<condition2>) {...} [elif (<condition3>) {...} [...] ] ]
[else {...}]
```

# Loops

Birthstone supports 3 types of loops: do-while, while, and (three-statement) for loops. These loops behave in the same way as in most programming languages. While loops are used when the condition require to continue through the loop is known. A do-while loop is similar except that the body of the loop will be executed once before the condition is checked. A for loop is used when the required number of iterations is know, or the programmer need to keep track of which iteration the loop is on or how many iterations it has gone through when the loop is done. Below is the syntax for each loop (the items in angle brackets are placeholders for code). The body of each type of loop can either be a single statement or a block (group of statements) delimited by braces `{}`. The body of a loop make contain `break` or `continue` statements. Break and continue act the same way as in most programming languages. A `break` statement will quit out of the loop. A `continue` statement will resume execution at the beginning of the next iteration of the loop, ignoring everything below the continue in the current iteration of the loop.

```
do <statement>; while (<condition>);

do
{
    <statement>;
    ...
} while (<condition>);

while(<condition>) <statement>;

while(<condition>)
{
    <statement>;
    ...
}
```

```

for (<initialize>;<condition>;<increment>) <statement>;

for (<initialize>;<condition>;<increment>)
{
    <statement>;
    ...
}

```

Here is an example:

```

# find the first 20 Fibonacci numbers
fibs := [0,1];
for (i:=2; i < 20; ++i)
{
    fibs = fibs + (fibs[-1] + fibs[-2]);
}
print fibs;

```

This will print the first 20 Fibonacci numbers: [1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144, 233, 377, 610, 987, 1597, 2584, 4181, 6765].

## Functions

### Declaring functions

To declare a function in Birthstone, use the keyword `def`, then the name of the function, then a comma delimited list of parameters enclosed in parenthesis, then a block of code to be executed when the function is called. For example, consider this simple greet function:

```

def greet(name)
{
    return "Hello, " + name + "!";
}

```

Calling `greet("John")` will return the string `"Hello, John!"`.

## Type Conversion

### Converting to Number

To convert a string to a number, you can either negate it twice or add 0 to it. To convert a boolean to a number, only double negation is possible, because addition is not valid on booleans.

For Example:

```

a := - -"4";      # a is the number 4
b := --true;      # b is the number 1
c := 0 + "7";     # c is the number 7

```

Note: When double negating a variable you must use `'- -x'` or `'- (-x)'`, otherwise the interpreter sees `'--'` as the decrement operator, which is only valid on numeric variables. When double negating a literal, the space or parenthesis are unnecessary.

## Converting to a Boolean

To convert to a boolean, simply use a double not (!!). Empty strings (""), 0, and false will be converted to false and other strings and numbers will be converted to true.

For Example:

```
a := !!""; # a is false
b := !!7;  # b is true
```

Alternatively, you could OR the value with false or AND the value with true;

```
c := false || "string"; # c is true
d := true && 0;          # d is false
```

## Converting to a String

To convert a number or boolean to a string, concatenate the empty string with it.

For Example:

```
a := "" + 7;      # a is the string "7"
b := "" + false;  # b is the string "false"
```

## Reserved Keywords

None	in	fprint	false
if	break	delete	exit
elif	continue	def	quit
elsif	read	class	
elseif	write	return	
else	print	and	
do	open	or	
while	close	not	
until	fread	type	
for	fwrite	true	