

Inform

in four minutes

A quick reference to the Inform programming language

Inform is copyright © 2002 by Graham Nelson
<http://www.gnelson.demon.co.uk/>

This guide is copyright © 2002 by Roger Firth
<http://www.firthworks.com/roger/>

Version 1.3 (March 2002)

The road to brevity is via imprecision and through solecism – refer to the *Inform Designer's Manual* for the definitive story.

•• Literals •••••

A Z-code **word** literal uses sixteen bits (whereas a Glulx word has thirty-two bits). A **byte** literal is always eight bits.

- Decimal: -32768 to 32767
Hexadecimal: \$0 to \$FFFF
Binary: \$\$0 to \$\$1111111111111111
- Action: ##Look
- Character: 'a'
- Dictionary word: 'aardvark' (up to nine characters significant); use circumflex “^” to denote apostrophe.
Plural word: 'aardvarks//p'
Single-character word: "a" (name property only) or 'a//'
- String: "aardvark's adventure" (maximum around 4000 characters); can include special values including:

^	newline
~	quotes “ ”
@@64	at sign “@”
@@92	backslash “\”
@@94	circumflex “^”
@@126	tilde “~”
@`a	a with a grave accent “à”, et al
@LL	pound sign “£”, et al
@00 ... @31	low string 0..31

•• Names •••••

The identifier of an Inform *constant*, *variable*, *array*, *class*, *object*, *property*, *attribute*, *routine* or *label*. Up to 32 characters: alphabetic (case not significant), numeric and underscore, with the first character not a digit.

•• Expressions and Operators •••••

Use parentheses (...) to control the order of evaluation. Arithmetic/logical expressions support these operators:

$p + q$	addition
$p - q$	subtraction
$p * q$	multiplication
p / q	integer division
$p \% q$	remainder
$p++$	increments p , evaluates to original value
$++p$	increments p , evaluates to new value
$p--$	decrements p , evaluates to original value
$--p$	decrements p , evaluates to new value
$p \& q$	bitwise AND
$p q$	bitwise OR
$\sim p$	bitwise NOT (inversion)

Conditional expressions return true (1) or false (0); q may be a list of alternatives $q1$ or $q2$ or ... qN :

$p == q$	p is equal to q
$p \sim= q$	p isn't equal to q
$p > q$	p is greater than q
$p < q$	p is less than q
$p >= q$	p is greater than or equal to q
$p <= q$	p is less than or equal to q
$p \text{ ofclass } q$	object p is of class q
$p \text{ in } q$	object p is a child of object q
$p \text{ notin } q$	object p isn't a child of object q
$p \text{ provides } q$	object p provides property q
$p \text{ has } q$	object p has attribute q
$p \text{ hasnt } q$	object p hasn't attribute q

Boolean expressions return true (1) or false (0):

$p \&\& q$	both p and q are true (non-zero)
$p q$	either p or q is true (non-zero)
$\sim\sim p$	p is false (zero)

To return -1, 0 or 1 based on unsigned comparison:

`UnsignedCompare(p,q)`

To return true if object q is a child or grand-child or... of p :

`IndirectlyContains(p,q)`

To return a random number 1..N, or one from a list of constant values:

`random(N)`
`random(value,value, ... value)`

•• Constants •••••

Named word values, unchanging at run-time, which are by default initialised to zero:

```
Constant constant;  
Constant constant = expr;
```

Standard constants are true (1), false (0) and nothing (0), also NULL (-1).

To define a constant (unless it already exists):

```
Default constant expr;
```

•• Variables and Arrays •••••

Named word/byte values which can change at run-time and are by default initialised to zero.

A **global** variable is a single word:

```
Global variable;  
Global variable = expr;
```

A **word array** is a set of global words accessed using `array-->0`, `array-->1`, ... `array-->(N-1)`:

```
Array array --> N;  
Array array --> expr1 expr2 ... exprN;  
Array array --> "string";
```

A **table array** is a set of global words accessed using `array-->1`, `array-->2`, ... `array-->N`, with `array-->0` initialised to N :

```
Array array table N;  
Array array table expr1 expr2 ... exprN;  
Array array table "string";
```

A **byte array** is a set of global bytes accessed using `array->0`, `array->1`, ... `array->(N-1)`:

```
Array array -> N;  
Array array -> expr1 expr2 ... exprN;  
Array array -> "string";
```

A **string array** is a set of global bytes accessed using `array->1`, `array->2`, ... `array->N`, with `array->0` initialised to N :

```
Array array string N;  
Array array string expr1 expr2 ... exprN;  
Array array string "string";
```

In all these cases, the characters of the initialising *string* are unpacked to the individual word/byte elements of the array.

See also Objects (for **property** variables) and Routines (for **local** variables).

