Forth

Forth was created by Charles H. Moore in the 70s. It is an imperative, stack-based language and programming environment, being used in projects such as Open Firmware. It's also used by NASA.

Note: This article focuses predominantly on the Gforth implementation of Forth, but most of what is written here should work elsewhere.

```
\ This is a comment
( This is also a comment but it's only used when defining words )

\ ---------------------------------- Precursor ----------------------------------

\ All programming in Forth is done by manipulating the parameter stack (more
\ commonly just referred to as "the stack").
5 2 3 56 76 23 65    \ ok

\ Those numbers get added to the stack, from left to right.
.s    \ <7> 5 2 3 56 76 23 65 ok

\ In Forth, everything is either a word or a number.

\ ------------------------------ Basic Arithmetic ------------------------------

\ Arithmetic (in fact most words requiring data) works by manipulating data on
\ the stack.
5 4 +    \ ok

\ `.` pops the top result from the stack:
.    \ 9 ok

\ More examples of arithmetic:
6 7 * .        \ 42 ok
1360 23 - .    \ 1337 ok
12 12 / .      \ 1 ok
13 2 mod .     \ 1 ok

99 negate .    \ -99 ok
-99 abs .      \ 99 ok
52 23 max .    \ 52 ok
52 23 min .    \ 23 ok

\ ------------------------------ Stack Manipulation ------------------------------

\ Naturally, as we work with the stack, we'll want some useful methods:

3 dup -          \ duplicate the top item (1st now equals 2nd): 3 - 3
2 5 swap /       \ swap the top with the second element:        5 / 2
6 4 5 rot .s     \ rotate the top 3 elements:                   4 5 6
4 0 drop 2 /     \ remove the top item (dont print to screen):  4 / 2
1 2 3 nip .s     \ remove the second item (similar to drop):    1 3

\ ---------------------- More Advanced Stack Manipulation ----------------------

1 2 3 4 tuck   \ duplicate the top item into the second slot:      1 2 4 3 4 ok
```

```
1 2 3 4 over   \ duplicate the second item to the top:             1 2 3 4 3 ok
1 2 3 4 2 roll \ *move* the item at that position to the top:      1 3 4 2 ok
1 2 3 4 2 pick \ *duplicate* the item at that position to the top: 1 2 3 4 2 ok

\ When referring to stack indexes, they are zero-based.

\ ------------------------------ Creating Words --------------------------------

\ The `:` word sets Forth into compile mode until it sees the `;` word.
: square ( n -- n ) dup * ;    \ ok
5 square .                     \ 25 ok

\ We can view what a word does too:
see square     \ : square dup * ; ok

\ ------------------------------- Conditionals ---------------------------------

\ -1 == true, 0 == false. However, any non-zero value is usually treated as
\ being true:
42 42 =    \ -1 ok
12 53 =    \ 0 ok

\ `if` is a compile-only word. `if` <stuff to do> `then` <rest of program>.
: ?>64 ( n -- n ) dup 64 > if ." Greater than 64!" then ; \ ok
100 ?>64                                                  \ Greater than 64! ok

\ Else:
: ?>64 ( n -- n ) dup 64 > if ." Greater than 64!" else ." Less than 64!" then ;
100 ?>64    \ Greater than 64! ok
20 ?>64     \ Less than 64! ok

\ ---------------------------------- Loops -------------------------------------

\ `do` is also a compile-only word.
: myloop ( -- ) 5 0 do cr ." Hello!" loop ; \ ok
myloop
\ Hello!
\ Hello!
\ Hello!
\ Hello!
\ Hello! ok

\ `do` expects two numbers on the stack: the end number and the start number.

\ We can get the value of the index as we loop with `i`:
: one-to-12 ( -- ) 12 0 do i . loop ;     \ ok
one-to-12                                 \ 0 1 2 3 4 5 6 7 8 9 10 11 12 ok

\ `?do` works similarly, except it will skip the loop if the end and start
\ numbers are equal.
: squares ( n -- ) 0 ?do i square . loop ;   \ ok
10 squares                                   \ 0 1 4 9 16 25 36 49 64 81 ok

\ Change the "step" with `+loop`:
```

```
: threes ( n n -- ) ?do i . 3 +loop ;    \ ok
15 0 threes                              \ 0 3 6 9 12 ok

\ Indefinite loops with `begin` <stuff to do> <flag> `until`:
: death ( -- ) begin ." Are we there yet?" 0 until ;    \ ok

\ ---------------------------- Variables and Memory -----------------------------

\ Use `variable` to declare `age` to be a variable.
variable age    \ ok

\ Then we write 21 to age with the word `!`.
21 age !    \ ok

\ Finally we can print our variable using the "read" word `@`, which adds the
\ value to the stack, or use `?` that reads and prints it in one go.
age @ .    \ 21 ok
age ?      \ 21 ok

\ Constants are quite similar, except we don't bother with memory addresses:
100 constant WATER-BOILING-POINT    \ ok
WATER-BOILING-POINT .               \ 100 ok

\ ---------------------------------- Arrays -----------------------------------

\ Creating arrays is similar to variables, except we need to allocate more
\ memory to them.

\ You can use `2 cells allot` to create an array that's 3 cells long:
variable mynumbers 2 cells allot    \ ok

\ Initialize all the values to 0
mynumbers 3 cells erase    \ ok

\ Alternatively we could use `fill`:
mynumbers 3 cells 0 fill

\ or we can just skip all the above and initialize with specific values:
create mynumbers 64 , 9001 , 1337 , \ ok (the last `,` is important!)

\ ...which is equivalent to:

\ Manually writing values to each index:
64 mynumbers 0 cells + !      \ ok
9001 mynumbers 1 cells + !    \ ok
1337 mynumbers 2 cells + !    \ ok

\ Reading values at certain array indexes:
0 cells mynumbers + ?    \ 64 ok
1 cells mynumbers + ?    \ 9001 ok

\ We can simplify it a little by making a helper word for manipulating arrays:
: of-arr ( n n -- n ) cells + ;    \ ok
mynumbers 2 of-arr ?               \ 1337 ok
```

```
\ Which we can use for writing too:
20 mynumbers 1 of-arr !    \ ok
mynumbers 1 of-arr ?       \ 20 ok

\ ------------------------------ The Return Stack -------------------------------

\ The return stack is used to the hold pointers to things when words are
\ executing other words, e.g. loops.

\ We've already seen one use of it: `i`, which duplicates the top of the return
\ stack. `i` is equivalent to `r@`.
: myloop ( -- ) 5 0 do r@ . loop ;    \ ok

\ As well as reading, we can add to the return stack and remove from it:
5 6 4 >r swap r> .s    \ 6 5 4 ok

\ NOTE: Because Forth uses the return stack for word pointers,  `>r` should
\ always be followed by `r>`.

\ ------------------------- Floating Point Operations --------------------------

\ Most Forths tend to eschew the use of floating point operations.
8.3e 0.8e f+ f.    \ 9.1 ok

\ Usually we simply prepend words with 'f' when dealing with floats:
variable myfloatingvar    \ ok
4.4e myfloatingvar f!     \ ok
myfloatingvar f@ f.       \ 4.4 ok

\ --------------------------------- Final Notes ----------------------------------

\ Typing a non-existent word will empty the stack. However, there's also a word
\ specifically for that:
clearstack

\ Clear the screen:
page

\ Loading Forth files:
\ s" forthfile.fs" included

\ You can list every word that's in Forth's dictionary (but it's a huge list!):
\ words

\ Exiting Gforth:
\ bye
```

Ready For More?

- Starting Forth
- Simple Forth
- Thinking Forth