Brainfuck

From Wikipedia, the free encyclopedia

The **brainfuck** programming language is an esoteric programming language noted for its extreme minimalism. It is a Turing tarpit, designed to challenge and amuse programmers, and is not suitable for practical use.[1] The name of the language is generally not capitalized except at the start of a sentence, although it is a proper noun.

brainfuck

Paradigm(s)	esoteric
Appeared in	1993
Designed by	Urban Müller
Influenced by	P″, FALSE
Usual filename extensions	.b, .bf

Contents

Language design

Urban Müller created brainfuck in 1993 with the intention of designing a language which could be implemented with the smallest possible compiler,[2] inspired by the 1024-byte compiler for the FALSE

programming language.[3] Several brainfuck compilers have been made smaller than 200 bytes. The classic distribution is Müller's version 2 (http://www.aminet.net/package.php?package=dev/lang/brainfuck-2.lha) , containing a compiler for the Amiga, an interpreter, example programs, and a readme document.

The language consists of eight commands, listed below. A brainfuck program is a sequence of these commands, possibly interspersed with other characters (which are ignored). The commands are executed sequentially, except as noted below; an instruction pointer begins at the first command, and each command it points to is executed, after which it normally moves forward to the next command. The program terminates when the instruction pointer moves past the last command.

The brainfuck language uses a simple machine model consisting of the program and instruction pointer, as well as an array of at least 30,000 byte cells initialized to zero; a movable data pointer (initialized to point to the leftmost byte of the array); and two streams of bytes for input and output (most often connected to a keyboard and a monitor respectively, and using the ASCII character encoding).

Commands

The eight language commands, each consisting of a single character:

Character	Meaning
`>`	increment the data pointer (to point to the next cell to the right).
`<`	decrement the data pointer (to point to the next cell to the left).
`+`	increment (increase by one) the byte at the data pointer.
`-`	decrement (decrease by one) the byte at the data pointer.
`.`	output a character, the ASCII value of which being the byte at the data pointer.
`,`	accept one byte of input, storing its value in the byte at the data pointer.
`[`	if the byte at the data pointer is zero, then instead of moving the instruction pointer forward to the next command, jump it *forward* to the command after the *matching* `]` command*.
`]`	if the byte at the data pointer is nonzero, then instead of moving the instruction pointer forward to the next command, jump it *back* to the command after the *matching* `[` command*.

(Alternatively, the *]* command may instead be translated as an unconditional jump **to** the corresponding *[* command, or vice versa; programs will behave the same but will run more slowly, due to unnecessary double searching.)

**[* and *]* match as parentheses usually do: each *[* matches exactly one *]* and vice versa, the *[* comes first, and there can be no unmatched *[* or *]* between the two.

Brainfuck programs can be translated into C using the following substitutions, assuming `ptr` is of type `unsigned char*` and has been initialized to point to an array of zeroed bytes:

brainfuck command	C equivalent
`>`	`++ptr;`
`<`	`--ptr;`
`+`	`++*ptr;`

-	`--*ptr;`
.	`putchar(*ptr);`
,	`*ptr=getchar();`
[	`while (*ptr) {`
]	`}`

As the name suggests, brainfuck programs tend to be difficult to comprehend. This is partly because any mildly complex task requires a long sequence of commands; partly it is because the program's text gives no direct indications of the program's state. These, as well as brainfuck's inefficiency and its limited input/output capabilities, are some of the reasons it is not used for serious programming. Nonetheless, like any Turing-complete language, brainfuck is theoretically capable of computing any computable function or simulating any other computational model, if given access to an unlimited amount of memory.[4] A variety of brainfuck programs have been written.[5] Although brainfuck programs, especially complicated ones, are difficult to write, it is quite trivial to write an interpreter for brainfuck in a more typical language such as C due to its simplicity. It is possible to write a brainfuck interpreter in the brainfuck language itself.[6]

Brainfuck's formal "parent language"

Except for its two I/O commands, brainfuck is a minor variation of the formal programming language P″ created by Corrado Böhm in 1964. In fact, using six symbols equivalent to the respective brainfuck commands `+`, `-`, `<`, `>`, `[`, `]`, Böhm provided an explicit program for each of the basic functions that together serve to compute any computable function. So in a very real sense, the first "brainfuck" programs appear in Böhm's 1964 paper – and they were programs sufficient to prove Turing-completeness.

Examples

Hello World!

The following program prints "Hello World!" and a newline to the screen:

```
+++++ +++++             initialize counter (cell #0) to 10
[                       use loop to set the next four cells to 70/100/3
    > +++++ ++              add  7 to cell #1
    > +++++ +++++           add 10 to cell #2
    > +++                   add  3 to cell #3
    > +                     add  1 to cell #4
    <<<< -                  decrement counter (cell #0)
]
> ++ .                  print 'H'
> + .                   print 'e'
+++++ ++ .              print 'l'
.                       print 'l'
+++ .                   print 'o'
> ++ .                  print ' '
<< +++++ +++++ +++++ .  print 'W'
> .                     print 'o'
+++ .                   print 'r'
----- - .               print 'l'
----- --- .             print 'd'
> + .                   print '!'
> .                     print '\n'
```

For readability, this code has been spread across many lines and blanks and comments have been added. Brainfuck ignores all characters except the eight commands `+-<>[],.` so no special syntax for comments is needed (as long as the comments don't contain the command characters). The code could just as well have been written as:

```
++++++++++[>+++++++>++++++++++>+++>+<<<<-]>++.>+.+++++++..+++.>++.<<++
```

The first line initialises `a[0] = 10` by simply incrementing ten times from 0. The loop from line 2 effectively sets the initial values for the array: `a[1] = 70` (close to 72, the ASCII code for the character 'H'), `a[2] = 100` (close to 101 or 'e'), `a[3] = 30` (close to 32, the code for space) and `a[4] = 10` (newline). The loop works by multiplying the value of `a[0]`, `10`, by 7, 10, 3, and 1, saving the results in other cells. After the loop is finished, a[0] is zero. `>++.` then moves the pointer to `a[1]` which holds `70`, adds two to it (producing 72 which is the ASCII character code of a capital H), and outputs it.

The next line moves the array pointer to `a[2]` and adds one to it, producing `101`, a lower-case 'e', which is then output.

As 'l' happens to be the seventh letter after 'e', to output 'll' another seven are added (`+++++++`) to `a[2]` and the result is output twice.

'o' is the third letter after 'l', so `a[2]` is incremented three more times and output the result.

The rest of the program goes on in the same way. For the space and capital letters, different array cells are selected and incremented or decremented as needed.

Trivial

Cell-clear

```
[-]
```

A simple program fragment that sets the value at the current location to 0, by iteratively decrementing until it is equal to 0.

Clear previous cells

```
[
    [-]
    <
]
```

Clears the current cell and all those before it, stopping at the first 0 value.

Rewind

```
[<]>
```

Decrements the data pointer until 0 is found, then increments it. Note: This guarantees a non-zero value at the final pointer if the initial pointer or next pointer has a non-zero value.

Fast-forward

```
[>]<
```

Increments the data pointer until 0 is found, then decrements it. Note: This guarantees a non-zero value at the final pointer if the initial pointer or previous pointer has a non-zero value.

Simple loop

```
,[.,]
```

A continuous loop that takes text input from the keyboard and echoes it to the screen (similar to the Unix cat program). Note that this assumes the cell is set to 0 when a ',' command is executed after the end of input (sometimes called end-of-file or "EOF"); implementations vary on this point. For implementations that set the cell to -1 on EOF, or leave the cell's value unchanged, this program would be written

```
,+[-.,+]
```

or

```
,[.[-],]
```

respectively.

Moving the pointer

```
>,[.>,]
```

A version of the last one that also saves all the input in the array for future use, by moving the pointer each time.

Add

```
[
    ->   cell #0: before loop: left addend; after loop: 0
    +<   cell #1: before loop: right addend; after loop: sum
]
```

This adds the current location (destructively, as its value is zeroed in the process) to the next location.

```
[        counter is cell #0
    -    cell #0
    >+   cell #1
    >+   cell #2
    <<
]
>>
[        loop to restore value of cell #0; counter is cell #2
    -    cell #2
    <<+  cell #0
    >>   cell #2
]
```

This code will not destroy the original value (in cell #0), but it is 1.5 times more memory hungry than the previous example, requiring 3 bytes of memory

Conditional loop statements

```
,----------[----------------------.,----------]
```

This program will take lowercase input from the keyboard and make it uppercase, exiting when the user presses the enter key. It requires only one cell of memory, into which the user repeatedly enters keystrokes. If any of the keystrokes is the newline character—that is, the user presses the enter key—most brainfuck implementations will return the value 10. When the result of

```
,
```

is 10, interpreting the segment

```
,----------
```

will cause the cell's value to be 0. In this case, the program will fail to enter to the loop and terminate.

On the other hand, if the character input was not a 10, the program boldly assumes it was a lowercase letter and enters the loop, wherein it subtracts another 22 from the input. This yields a total deduction of 32, which is the difference between an ASCII lowercase letter and the corresponding uppercase letter (for example, lowercase f = ASCII 102 and uppercase F = ASCII 70). The result of all this arithmetic is realized when the uppercase character is outputted.

Next the segment

```
,----------
```

is repeated, putting a keystroke's value in the cell and subtracting 10. It has the same effect as before: if the inputted character is a linefeed, the loop is exited and the program terminates. Otherwise, the interpreter goes back to the start of the loop, subtracts another 22, outputs, and so on.

Copying a byte

In the following section, [n] denotes the *n*th byte in the array: [0], [1], [2] and so on.

Brainfuck does not include an operation for copying bytes. This must be done with the looping construct and arithmetical operators. Moving a byte is simple enough; moving the value of [0] to [1] can be done as follows:

```
>[-]<[->+<]
```

However, this resets the value of [0] to 0. The value of [0] can be restored after copying by taking advantage of the ability to copy a value to two places at once. To copy the value of [0] to both [1] and [2] is simple:

```
>[-]>[-]<<[->+>+<<]
```

This can be used to restore the value of [0]. Therefore, [0] can be nondestructively copied to [1] (using [2] as scratch space) as follows:

```
>[-]>[-]<<[->+>+<<]>>[-<<+>>]<<
```

Seek

```
-[+>-]+
```

Moves the pointer forward until it lands on a byte with a value of 1, preserving all bytes it passes; then it stops. To find other values, repeat each + and - that number of times. To seek a cell with a value of 10:

```
----------[++++++++++>----------]++++++++++
```

Likewise, to search backward for a value of 10:

```
----------[++++++++++<----------]++++++++++
```

Arithmetic

Addition

```
,>++++++[<-------->-],[<+>-]<.
```

This program adds two single-digit numbers and displays the result correctly if the result also has only one digit:

```
43
```

```
7
```

The first number is input in [0], and 48 is subtracted from it to correct it (the ASCII codes for the digits 0-9 are 48-57). This is done by putting a 6 in [1] and using a loop to subtract 8 from [0] that many times. (This is a common method of adding or subtracting large numbers.) Next, the second number is input in [1].

The next loop `[<+>-]` does the real work, moving the second number onto the first, adding them together and zeroing [1]. Each time through, it adds one to [0] and subtracts one from [1]; so by the time [1] is zeroed, as many have been added to [0] as have been removed from [1]. Now a return is input in [1]. (We're not error-checking the input at all.)

Then the pointer is moved back to the [0], which is then output. ([0] is now a + (b + 48), since b isn't corrected; which is identical to (a + b) + 48, which is the desired result.) Now the pointer is moved to [1], which holds the return that was input; it is now outputted, and the program is complete.

Apparently, some implementations prefer this variant which does not use linefeeds at all:

```
,>------[<-------->+],[<+>-]<.
```

Multiplication

```
,>,>++++++++[<------<------>>-]
<<[>[>+>+<<-]>>[<<+>>-]<<<-]
>>>++++++[<++++++++>-]<.>.
```

Like the previous, but does multiplication, not addition.

The first number is input in [0], the second number is input in [1], and both numbers are corrected by having 48 subtracted.

Now the program enters the main multiplication loop. The basic idea is that each time through it subtracts one from [0] and adds [1] to the running total kept in [2]. In particular: the first inner loop moves [1] onto both [2] and [3], while zeroing [1]. (This is the basic way to duplicate a number.) The next inner loop moves [3] back into [1], zeroing [3]. Then one is subtracted from [0], and the outer loop is ended. On exiting this loop, [0] is zero, [1] still has the second number in it, and [2] has the product of the two numbers. (Had the first number needed to be kept, the program could have added one to [4] each time through the outer loop, then moved the value from [4] back to [0] afterward.)

Now 48 is added to the product, a return is input as [3], the ASCIIfied product is output, followed by the return just stored.

Division

This example accepts two single-digit numbers from the user, divides them, and displays the truncated quotient (i.e. integer division). The dividend is stored in memory location (0) and the divisor is stored in (1) The code then enters the main loop. With each iteration of the loop, the code subtracts the divisor from the dividend. If the difference is greater than zero, the cell holding the quotient is incremented, and the process repeated until the dividend reaches zero.

```
,>,>++++++[-<--------<-------->>] Store 2 numbers from keyboard in (0)
                                  and subtract 48 from each
<<[                               This is the main loop which continues
                                  the dividend in (0) is zero
>[->+>+<<]                        Destructively copy the divisor from (
                                  and (3); setting (1) to zero
>[-<<-                            Subtract the divisor in (2) from the
                                  in (0); the difference is stored in (
                                  (2) is cleared
[>]>>>[<[>>>-<<<[-]]>>]<<]        If the dividend in (0) is zero; exit
>>>+                              Add one to the quotient in (5)
<<[-<<+>>]                        Destructively copy the divisor in (3)
<<<]                              Move the stack pointer to (0) and go
                                  the start of the main loop
>[-]>>>>[-<<<<<+>>>>>]            Destructively copy the quotient in (5
                                  (not necessary; but cleaner)
<<<<++++++[-<++++++++>]<.         Add 48 and print result
```

Portability issues

Partly because Urban Müller did not write a thorough language specification, the many subsequent brainfuck

interpreters and compilers have come to use slightly different dialects of brainfuck.

Cell size

In the classic distribution (http://wuarchive.wustl.edu/pub/aminet/dev/lang/brainfuck-2.lha) , the cells are of 8-bit size (cells are bytes), and this is still the most common size. However, to read non-textual data, a brainfuck program may need to distinguish an end-of-file condition from any possible byte value; thus 16-bit cells have also been used. Some implementations have used 32-bit cells, 64-bit cells, or bignum cells with practically unlimited range, but programs that use this extra range are likely to be slow, since storing the value n into a cell requires $\Omega(n)$ time as a cell's value may only be changed by incrementing and decrementing.

In all these variants, the `,` and `.` commands still read and write data in bytes. In most of them, the cells wrap around, i.e. incrementing a cell which holds its maximal value (with the `+` command) will bring it to its minimal value and vice versa. The exceptions are implementations which are distant from the underlying hardware, implementations that use bignums, and implementations that try to enforce portability.

Fortunately, it is usually easy to write brainfuck programs that do not ever cause integer wraparound or overflow, and therefore don't depend on cell size. Generally this means avoiding increment of +255 (unsigned 8-bit wraparound), or avoiding overstepping the boundaries of [-128, +127] (signed 8-bit wraparound) (since there are no comparison operators, a program cannot distinguish between a signed and unsigned two's complement fixed-bit-size cell and negativeness of numbers is a matter of interpretation). For more details on integer wraparound, see the Integer overflow article.

Array size

In the classic distribution, the array has 30,000 cells, and the pointer begins at the leftmost cell. Even more cells are needed to store things like the millionth Fibonacci number, and the easiest way to make the language Turing-complete is to make the array unlimited on the right.

A few implementations[7] extend the array to the left as well; this is an uncommon feature, and therefore portable brainfuck programs do not depend on it.

When the pointer moves outside the bounds of the array, some implementations will give an error message, some will try to extend the array dynamically, some will not notice and will produce undefined behavior, and a few will move the pointer to the opposite end of the array. Some tradeoffs are involved: expanding the array dynamically to the right is the most user-friendly approach and is good for memory-hungry programs, but it carries a speed penalty. If a fixed-size array is used it is helpful to make it very large, or better yet let the user set the size. Giving an error message for bounds violations is very useful for debugging but even that carries a speed penalty unless it can be handled by the operating system's memory protections.

End-of-line code

Different operating systems (and sometimes different programming environments) use subtly different versions of ASCII. The most important difference is in the code used for the end of a line of text. MS-DOS and Microsoft Windows use a CRLF, i.e. a 13 followed by a 10, in most contexts. UNIX and its descendants, including Linux and Mac OS X, use just 10, and older Macs use just 13. It would be unfortunate if brainfuck programs had to be rewritten for different operating systems. Fortunately, a unified standard is easy to find. Urban Müller's compiler and his example programs use 10, on both input and output; so do a large majority of existing brainfuck programs; and 10 is also more convenient to use than CRLF. Thus, brainfuck implementations should make sure that brainfuck programs that assume newline=10 will run properly; many do so, but some do not.

This assumption is also consistent with most of the world's sample code for C and other languages, in that they use '\n', or 10, for their newlines. On systems that use CRLF line endings, the C standard library transparently remaps "\n" to "\r\n" on output and "\r\n" to "\n" on input for streams not opened in binary mode.

End-of-file behavior

The behavior of the , command when an end-of-file condition has been encountered varies. Some implementations set the cell at the pointer to 0, some set it to the C constant EOF (in practice this is usually -1), some leave the cell's value unchanged. There is no real consensus; arguments for the three behaviors are as follows.

Setting the cell to 0 avoids the use of negative numbers, and makes it marginally more concise to write a loop that reads characters until EOF occurs. This is a language extension devised by Panu Kalliokoski.

Setting the cell to -1 allows EOF to be distinguished from any byte value (if the cells are larger than bytes), which is necessary for reading non-textual data; also, it is the behavior of the C translation of , given in Müller's readme file. However, it is not obvious that those C translations are to be taken as normative.

Leaving the cell's value unchanged is the behavior of Urban Müller's brainfuck compiler. This behavior can easily coexist with either of the others; for instance, a program that assumes EOF=0 can set the cell to 0 before each , command, and will then work correctly on implementations that do either EOF=0 or EOF="no change". It is so easy to accommodate the "no change" behavior that any brainfuck programmer interested in portability should do so.

Miscellaneous dialects

Many people have modified brainfuck in order to produce their own languages, often by adding commands, occasionally by removing them. Many of the resulting languages are included in the brainfuck derivatives (http://esoteric.voxelperfect.net/wiki/Category:Brainfuck_derivatives) list on the Esoteric Languages wiki. However, there are also unnamed minor variants, formed possibly as a result of inattention, of which some of the more common are:

- forbidding, rather than ignoring, any non-command characters in brainfuck programs
- introducing a comment marker which comments out the rest of the line
- various alterations of the loop semantics, sometimes destroying Turing completeness
- requiring a special character to mark the end of the program

References

1. ^ Urban Müller's classic distribution at aminet.net (http://aminet.net/dev/lang/brainfuck-2.readme)
2. ^ The Brainfuck Programming Language (http://www.muppetlabs.com/~breadbox/bf/)
3. ^ Wouter's False page (http://strlen.com/false/index.html)
4. ^ BF is Turing-complete (http://www.iwriteiam.nl/Ha_bf_Turing.html)
5. ^ The Brainfuck Archive (http://esoteric.sange.fi/brainfuck/bf-source/prog/)
6. ^ Brainfuck interpreter written in brainfuck (http://www.iwriteiam.nl/Ha_bf_inter.html)
7. ^ Beef brainfuck interpreter (http://kiyuko.org/software/beef)

External links

- The Brainfuck Archive (http://esoteric.sange.fi/brainfuck/)
- Brainfuck (http://esolangs.org/wiki/Brainfuck) on the Esolang (Esoteric Languages) wiki (http://esolangs.org/wiki/Main_Page)

- Visual brainfuck (https://sites.google.com/site/visualbf) , a brainfuck IDE compatible with Windows 7 x86 and x64.
- Brainfuck Developer (http://4mhz.de/bfdev.html) , Brainfuck IDE for Windows (also works with WINE under Linux)
- Brainfuck (http://www.dmoz.org/Computers/Programming/Languages/Brainfuck/) at the Open Directory Project
- bf2c (http://code.google.com/p/junkdrawer/downloads/) , a Brainfuck to C translator.
- ookie (http://rubygems.org/gems/ookie) , a Brainfuck and Ook! language interpreter
- BrainForce (http://www.beco.cc/compiler/brainforce/brainforce.html) , compiler (wrap gcc) and C translator (has lots of options to control wrapping values, cell sizes, etc.)
- esolang (http://www.davenicholas.me.uk/blog/view_post/31/esolang-esoteric-programming-for-iphone) , a Brainfuck interpreter for iPhone written in objective c.
- Le Brainfuck (http://copy.freeunix.net/brainfuck/) , a Javascript based optimizing interpreter. Also has many options, including memory dumping.

Retrieved from "http://en.wikipedia.org/wiki/Brainfuck"
Categories: Non-English-based programming languages | Esoteric programming languages