The Brainfuck Programming Language

Brainfuck is probably the craziest language I have ever had the pleasure of coming across. The language itself is a Turing-complete language created by Urban Müller. It consists of 8 operators, yet with the 8 operators, <>+-[],. You are capable of writing almost any program you can think of. To write programs in Brainfuck, I would suggest you get a few things first.

First, an interpretor. For Linux, you could try Beef or bf, and a quick Google should give you a variety of options to choose from for any operating system. Next, I would suggest an ASCII chart with all the ASCII chars and their decimal equivalent value. Next on the items is would be a calculator. Any will do. It will help you figure out the Greatest Common Factors for use in incrementing a memory block quickly.

THE BASICS

The idea behind Brainfuck is memory manipulation. Basically you are given an array of 30,000 1byte memory blocks. The array size is actually dependent upon the implementation used in the compiler or interpreter, but standard Brainfuck states 30,000. Within this array, you can increase the memory pointer; increase the value at the memory pointer, etc. Let me first present to you the 8 operators available to us.

BrainFuck	C++ equivalent	Explanation
+	a[p]++	Increases current cell value by 1
-	a[p]--	Decreases current cell value by 1
<	p--	Decreases cell pointer by 1
>	p++	Increases cell pointer by 1
[ ]	while(a[p] != 0)	Check if the current cell value is zero. If it is, jump to the matching], otherwise it continues
,	a[p] = (byte) getchar()	Reads a user input, stores ascii value to current cell
.	printf((char) a[p])	Prints current cell value as an ascii character

Some rules:

- Any arbitrary character besides the 8 listed above should be ignored by the compiler or interpreter. Characters besides the 8 operators should be considered comments.
- All memory blocks on the "array" are set to zero at the beginning of the program. And the memory pointer starts out on the very left most memory block.
- Loops may be nested as many times as you want. But all [must have a corresponding].

A few examples:

Let’s start with some examples of how to program in Brainfuck.
The simplest program in Brainfuck is:

```
[-]
```

Well, that's what they say anyway, I hardly consider that a program, because all it does is enter a loop that decreases the value stored at the current memory pointer until it reaches zero, then exits the loop. But since all memory blocks start out at zero, it will never enter that loop. So let’s write a real program.

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

This is equivalent in C to:

```
*p=+5;
while(*p != 0)
{
        *p--;
}
```

In that program we are incrementing the current memory pointers value to 5, then entering a loop that decreases the value located at the memory pointer till it is zero, then exits the loop.

```
>>>++
```

This will move the memory pointer to the fourth memory block, and increment the value stored there by 2. So it looks like

```
memory blocks
-------------
[0][0][0][2][0][0]...
^
memory pointer
```

As you can see in the 'k-rad' ASCII diagram, our memory pointer points to the fourth memory block and it increments the value there by 1. Since there was nothing there before, it now contains the value: 2. If we take that same program, and add more onto the end of it like:

```
>>>>++<<+>>+
```

At the end of our program, our memory layout will look like this:

```
memory blocks
-------------
[0][1][0][3][0][0]...
^
memory pointer
```

The pointer was moved to the fourth block, incremented the value by 2, moved back 2 blocks to the second block, incremented the valued stored there by 1, and then the pointer moved 2 blocks to the right again to the fourth block and incremented the value stored there by one. And at the end of the program the memory pointer lies back on the fourth memory block. That is fine and dandy, but we can't really see anything. So let’s write a program that will produce actual output.

A program to print “Hello World”

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

We must remember that we are working with numbers, so we must use a character's ASCII decimal number to represent it. Then when we print it, will print the value as an ASCII character. Let’s break this program down.

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

Let’s break this part down farther using our diagrams.

```
>
```

First you can see that we increment the memory pointer to the next memory block leaving the first memory block at zero.

```
memory blocks
-------------
[0][0][0][0][0][0]...
^
memory pointer
```

We then increase the value at our current memory block to 9.

```
+++++++++
```

Leaving our diagram like this:

```
memory blocks
-------------
[0][9][0][0][0][0]...
^
memory pointer
```

Since the block we are on contains a non-zero value, we then enter the loop.

```
[
```

Now we are in the loop. Then we move the memory pointer one block to the left.

```
<
```

Which gives us:

```
memory blocks
-------------
[0][9][0][0][0][0]...
^
memory pointer
```

And we increment the memory blocks stored value by 8.

```
++++++++
```

So our diagram looks like:

```
memory blocks

-------------
[8][9][0][0][0][0]...
^
memory pointer
```

Then we move the memory pointer one block to the right, to the second memory block again, and decrease the value stored there from 9 to 8.

```
>-
```

Diagram:

```
memory blocks
-------------
[8][8][0][0][0][0]...
^
memory pointer
```

We then hit the end of our loop.

```
]
```

It checks to see if the memory block the pointer currently points to contains the value zero, but current memory block's stored value is not zero, so the loop starts over. Moving the pointer to the left, increasing it by 8, and moving the pointer to the right and decreasing it by 1. After the 2nd pass of all that, our diagram now looks like:

```
memory blocks
-------------
[16][7][0][0][0][0]...
^
memory pointer
```

It will continue this process over and over until the value stored at the second memory block is zero. It then exits the loop. Once we have exited the loop. The program moves the pointer back to the first memory block one final time, and prints the value stored there. If you followed that, you would see that we increased the first memory blocks stored value by 8, 9 times. We know that 8*9=72 and 72 is the ASCII decimal value for 'H'.

```
<.
```

And the diagram:

```
memory blocks
-------------
[72][0][0][0][0][0]...
^
memory pointer
```

Call the print function and 'H' is printed to the console.

I'm going to leave it up to you to figure out how the rest of that is printing out "Hello World!" But from that you should have the basics of memory pointer and value manipulation.

Wow...that was a lot of freaking work just to print one single character. Why you may ask would you want to waste your time programming is this horribly inefficient programming language?!? Well, because some hackers that actually like to do fun and challenging things to expand their minds and make them think.
If we were to write that in C, it would be like:

```
++p;
*p=+9;
while(*p != 0){
--p;
*p=+8;
--p;
--*p;
}
--p;
putchar(*p);
```

I'm going to leave it up to you to figure out how the rest of that is printing out "Hello World!" But from that you should have the basics of memory pointer and value manipulation.

INPUT/OUTPUT

Input in Brainfuck is controlled by the ',' operator. It will get a character and store its ASCII decimal value to the current memory block that the memory pointer points to. Let’s experiment with it a bit.
Remember, when you use the input operator, you are actually storing the decimal ASCII value of the character you press on the keyboard. So pressing 2 for input isn't actually storing 2. Its storing the decimal value of the ASCII char '2', which is decimal 50.

```
,.,.,.
```

This will take in 3 characters and print them out. Let’s write something more complex.

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

This is a program that will act like the UNIX cat command. It will read in from STDIN and output to STDOUT. Let’s break it down.

```
>,
```

Move the memory pointer the second memory block leaving the first block with a value of zero. Input a value and store it at the current memory pointer location which is the second memory block.

```
[>,]
```

Begin a loop that will move the pointer up a memory block, and Input a value and store it there. This will repeat until it encounters a NULL character (\0 or decimal value of zero);

```
<[<]
```

Rewind. Once we've made it to this point in the program, it means that we have encountered a NULL character. So in order to start our loop, we need to move the memory pointer one memory block backwards so that we have a non-zero value stored there. Once there, the loops starts, and moves the memory pointer one block to the left until we reach the first memory block, which we left with a value of zero at the beginning of the program. Once it reaches the first memory block with the value of zero, the loop exits.

```
>[.>]
```

Now we move our memory pointer to the right one space, so we are now on a memory block containing a non-zero value. We enter a loop and proceed to print the current value stored, then move the memory pointer to the right. We continue to do this until we come to a memory block containing a NULL character (zero) and then the loop exits. This program in C would be like:

```
++p;
*p=getchar();
while(*p != 0){
++p;
*p=getchar();
}
--p;
while(*p != 0) --p;
++p;
while(*p != 0)
{
        putchar(*p);
        ++p;
}
```

TRICKS

There are many little tricks you can use in Brainfuck to make it easier. I will try to cover ones I have figured out.

How to MOVE or shift a value from one memory block to another:

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

This will set the first memory block to the value of 5. It then starts a loop that will copy the value stored in the first block, to the third memory block. Leaving the first memory block empty again.

How to COPY from one memory block to another:

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

This little program sets the first memory block to the value of 5. Then it goes and copies that value to the 3rd memory block and 4th memory block, leaving the first memory block empty. It then moves the value from the 4th memory block back to the first one, leaving the 4th block empty.

ADDITION of 2 memory blocks and easily be done as well.

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

We increment the first block to 5. Move the pointer the right one block, and then increment that block by three. We want to add the second memory block to the first one. So we enter a loop that will move the pointer to the left one block, add one, then move it to the right 1 block and subtract one.

SUBTRACTION of one block from another is just as easy.

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

We increment the first block to 7, move to the right one block, increment it by 5, then we begin a loop that will move the pointer to the left and subtract one then move the pointer back to the right and decrease the value stored there. Doing this until we have subtracted 5 from 7.

MULTIPLICATION we have covered before in our hello world program, but I will go over it again right here.
'

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

We just incremented the first blocks value to 3, then started a loop that will move the pointer to the right one block, add 5, then move the pointer back to the left one block and subtract one. This will accomplish multiplying 5 by 3 and leave the value stored at the second memory block at 15.

IF Statements

Say we want to input into memory block 1. Then we would like to test if the input value (x) was equal to 5, and if so, set y to 3. There are two ways to do this, one is the destructive flow control, where it diminishes the value you are test. The other obviously non-destructive flow control where you variable stays intact. Here is non-destructive way:

In C:

```
x=getchar;
if(x == 5)
        y = 3;
```

In Brainfuck it would look like:

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

Once again, let’s break that down to hopefully explain that better. Run though this twice. Once as we go along assuming the value 6 was entered, and once assuming the value 5 was entered. Also, remember, Brainfuck will _only_ enter as loop if the value in the block that the pointer is currently on is non-zero. If the value at the block is zero, then it will skip over that loop and ignore it. And the same goes while in a loop. If when it reaches the other end of that loop ( ]), if the value stored at the block where the pointer is currently at is zero, it will exit the loop and continue on with the program.

Input into x

```
1 2 3 4 5 6
[x][y][0][0][0][0]...
^
memory pointer
```

Copy from block 1, to block 3, using block 4 as a temp storage. We end on block number 4.

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

```
1 2 3 4 5 6
[x][y][x][0][0][0]
^
memory pointer
```

Set block 5 to 1. This will be our block to test for true of false. Then move the pointer back to 3.

```
>+<<
```

```
1 2 3 4 5 6
[x][y][x][0][1][0]
^
memory pointer
```

Now subtract 5 and if x was 5 set y to 3 and then move the pointer back over to block 5 and set back to zero so that the loop will only run once. If x was not equal to 5, then the pointer will end up resting on memory block 6.

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

```
if x was 5:
1 2 3 4 5 6
[x][3][0][0][0][0]...
^
memory pointer
```

```
if x was not 5
1 2 3 4 5 6
[x][y][x][0][1][0]...
^
memory pointer
```

That was the non destructive way to do an if statement. The destructive way would be to just subtract from the input variable directly instead of copying it. This will lead to much shorter code. Lets test x for the value 5 again and set y to 3 if it is.

Destructive way:

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