Bash c-style if statement and styling techniques

Asked 11 years, 5 months ago Modified 2 years, 9 months ago Viewed 11k times

9

3

From what I understand you can do C-style for and while loops in bash.

```
LIMIT=10

for ((a=1; a <= LIMIT ; a++))  # Double parentheses, and "LIMIT" with no "$".
do
  echo -n "$a "
done                           # A construct borrowed from 'ksh93'.
```

And even ternary operators.

```
(( var0 = var1<98?9:21 ))
```

How would you do this with the `if` statement?

Also why don't they implement braces like in C? What is the logic with using all of these keywords like `done`, `do`, `if`, and `fi`? I will be writing some scripts but bash appears very different.

Are there any bash styling techniques or bash alternatives/plugins? I would like to follow the standard, but coming from a C, Java and PHP background bash looks really weird. Maintainability and standards are important.

bash if-statement

Share Edit Follow Flag edited Feb 21, 2011 at 2:07 asked Feb 21, 2011 at 0:11

Lime
13.2k 11 51 88

2 If you want to use C then use C. But don't use csh. – Ignacio Vazquez-Abrams Feb 21, 2011 at 0:14

there are many languages besides bash you can use. If you want brackets, Perl may suit you. But IMO, brackets are not necessary at all in programming languages. Others include Python, and Ruby. IMO, these 2 have more "cleaner" syntax. – kurumi Feb 21, 2011 at 0:19

1 `[` and `]` are called "brackets" (or "square brackets"); `{` and `}` are braces (or "curly brackets", or "curly braces"); `(` and `)` are parentheses (also, "brackets", "round brackets", and "parens"). C uses braces for code blocks and brackets for arrays. – Dave Jarvis Feb 21, 2011 at 0:24

5 Answers

Sorted by: Highest score (default)

Trending sort available

17

There are some syntactic specialties of bash (and maybe other shells of the `sh` family) related to if:

- The basic if-statement has the syntax `if condition-command ; then list ; fi` (with the possibility to add `elif` and `else` for other branches).
  - The `condition-command` can be any command that bash supports (even another if, I think), is executed and the return value is then evaluated. (Actually, it can also be a list of commands, but this might confuse readers. The return value of the last command counts for the condition.)
  - Bash (and Unix shells in general) interpret result codes of programs in a bit unusual way (for C programmers): a result of 0 is interpreted as true (or "no error"), and any other result is interpreted as false (or "some error occurred"). (In C and some other languages it is the other way around: 0 is `false` and everything else is `true`.)
  - If the result of the condition-command is 0 (true), the `list` is executed (and the return value of the `if` is the result of the list). If it is not-0 (false), the list is not executed (but the `elif` or `else` branches, if there are any). (If none of the lists is executed, the return value of the if is 0.)
- Some useful commands as conditions (these work also as conditions in `while loops`):
  - `[ ... ]` is in fact another way to write `test ...` - this is simply a command which may return 0 (true) or 1 (false) depending on what parameters you are giving. (This is a buildin of the shell.)
  - `[[ ... ]]` is a conditional expression command. It supports some of the same arguments that `[` supports, and some more, like parentheses, `<`, `>`, and `( ... )` for condition nesting, and handles expansions inside a bit different, since it is a special syntactic construct instead of a buildin command. The whole thing is a command that returns 0 (true) or 1 (false).
  - `(( ... ))` is a wrapper for an arithmetic expression (or several of them) as a command. The result is 0 (true) when the result of the (last) expression is non-0, and 1 (false) when the result of the (last) expression is 0. (You could instead write `let ...`.)

    In arithmetic expressions the shell variables can be used without `$`, and everything is interpreted as integers (of a fixed width, but I didn't find out which). You can use these operators:

    - unary: `++`, `--`, `+`, `-`, `!`, `~` (like in C)
    - binary arithmetic: `**` (exponentation), `*`, `/`, `%`, `+`, `-`
    - bit-shift: `<<`, `>>`
    - comparison: `<=`, `>=`, `<`, `>`, `==`, `!=` (I think these return either 1 (true) or 0 (false))

(false))

- bitwise: `&` , `^` , `|`
- logical: `&&` , `||` , and of course the ternary operator: `... ? ... : ...`
- assignment: `=` , `*=` , `/=` , `%=` , `+=` , `-=` , `<<=` , `>>=` , `&=` , `^=` , `|=`
- comma: `,` (simply connects two expressions, returning the value of the last one)

I think in general this works a lot like C integer expressions. (These are roughly in order of precedence, but sometimes there are divisions inside of each group. Look in `info '(bash)Shell Arithmetic'` for details.)

- `true` is a buildin which does nothing and always returns 0.
- `false` is a buildin which does nothing and always returns 1.
- you can also call your external C (or anything else) programs as commands, their return values are interpreted the same way.

- Lists:
  - `( ... )` is a list of commands, which is executed in a subshell. Variable assignments in this subshell are not propagated back.
  - `{ ...; }` is a list of commands, which does not create a subshell.
  - `;` is a simple separator of commands in a list, but can also be replaced by a new-line.
  - `&&` is a separator of commands executing the second one only when the first returned 0.
  - `||` is a separator of commands executing the second one only when the first returned not-0.
  - `&` is a separator of commands, executing both commands in parallel (the first one in background).

So, you **can** write your `if` -command with some braces and parentheses, but you still need your `then` and `fi` :

```
if (( i > 0 ))
then {
   echo "i > 0"
}
else {
   echo "i <= 0"
}
fi
```

The braces here are simply superfluous, since the commands between `then` , `else` and `fi` are (each) one list anyway. (Also note the need of new-lines or `;` after and before the closing braces here.)

braces here.)

But as the others said, **better use another language if you want to use a C-like syntax.**

Share Edit Follow Flag

edited Jun 27, 2013 at 19:05

answered Feb 21, 2011 at 1:10

Paŭlo Ebermann
71.3k 18 140 206

Wow thank for all of the descriptions ! Also I had know idea about the then{ } else{} fi syntax. – Lime Feb 21, 2011 at 1:33

I have nothing against bash's syntax. All syntax's take time to get used to. I was really just confused with all of the [], [[]], {}, $() and (()) business. It originally had looked incredibly ugly and not apparently useful. That list really helps but could you clarify the ((...)) syntax. What is "1 (false)" supposed to mean? – Lime Feb 21, 2011 at 1:41

The braces in these then and else blocks are simply superfluous, as the contents of those blocks are evaluated as lists anyways. – Bash (and unix shells in general) interpret result codes of programs in a bit unusual way (for C programmers): a result of 0 is interpreted as `true` (or "no error"), and any other result is interpreted as `false` (or "some error occurred"). Thus I added the true or false in parentheses to these numbers to help avoid confusion. (Clarifying of `((...))` comes tomorrow, I now should go to bed.) – Paŭlo Ebermann Feb 21, 2011 at 1:47

That makes sense. :) In c programs you return 0 for no error. Its just in javascript, 0(and empty strings) means false in expressions while any other value is true. I wish someone standardized all these syntax differences. – Lime Feb 21, 2011 at 1:54

So, I added a bit more about the arithmetic expressions (and incorporated my last comment in the answer). – Paŭlo Ebermann Feb 21, 2011 at 12:39

1

In bash, the if statement doesn't have brackets like `()` eg of some syntax in bash

```
if [[ ... ]];then
  ..
fi

if [ ... ];then
 ...
fi

if $(grep .. file) ;then  #note the () is part of $().
  ...
fi
```

I have nothing against Bash, but IMO, you are better off using a programming language which can provide you all the things you need in your scripting/programming endeavor, such as Python, Ruby or Perl. Bash has some limitations, like performing floating points maths,etc.Also, if your task is complex enough, excessive use of *nix tools may lead to "bloatedness" of your script, hence hindering performance. If you want "clean" syntax and maintainability, either Ruby/Python may suit you. IMO, having brackets or not is just language design considerations and I personally would not want brackets if i have the choice.

Share Edit Follow Flag

answered Feb 21, 2011 at 0:25
kurumi
24.3k 4 43 50

Bash uses double parentheses for numeric comparison: `a=3; if (( a > b )); then ...` .
– Dennis Williamson Feb 21, 2011 at 1:14

1

The reason for the current syntax is largely historical. Most of what you're asking about is based on the original Bourne shell (sh). Bourne chose to base the syntax on Algol instead of C.

To have a more C-like syntax (and not be csh), someone would have to start from scratch.

As others have said, for many purposes you can write scripts in other languages.

You might want to look at Tiny C Compiler which can be used to run C source as a shell script.

```
#!/usr/bin/tcc -run
#include <stdio.h>

int main()
{
    printf("Hello World\n");
    return 0;
}
```

Share Edit Follow Flag

answered Feb 21, 2011 at 1:36

Dennis Williamson

328k 88 368 430

1

Use `&&` and `||` command lists and *curly-braced* group commands

On examples below, `true`, `false`, or `test 0 -eq 0` could be replaced by **any** command, including arithmetic expressions like `(( i > 0 ))`

Multi-line style

```
true && {
  echo "true command returns zero (aka success)"
  # more commands here
  # and here
} || {
  echo "true command never returns non-zero"
  # more commands here
  # and here
}
```

Obfuscated *in-line* style

```
# I don't like this one so much
false && echo "false never returns zero" || echo "false returns always 1"
```

Split *in-line* style

```
# here the draw-back are back slashes
test 0 -eq 0 \
  && echo "Surprise, zero is equal to zero" \
  || echo "Apocalypse is near if zero is not zero"
```

NOTE: I can assert the techniques above work, but I don't know the performance drawbacks, if there are.

Share Edit Follow Flag

answered Oct 15, 2019 at 1:48
laconbass
15.7k 6 41 50

0

If you're not tied to bash in particular for some reason, I would encourage you to use a more modern structured scripting language such as Ruby, Python, or PHP, and simply use the appropriate script header to make them executable like a bash script. I personally am a huge fan of bash one-liners on the console, but find it to be completely obtuse when used as a scripting language.

Most modern scripting languages will give you easy access to an array like `ARGV` for command line arguments, and support familiar loops and control structures.

So, for instance, you'd write a Ruby script and preface it with `#!/usr/bin/ruby` , like so:

```
#!/usr/bin/ruby
for a in 1..10
  puts ( a >= 5 ) ? "a >= 5" : "a < 5"
end
```

Share Edit Follow Flag

answered Feb 21, 2011 at 0:31

Jack Senechal

1,560 2 17 20