What is the difference between test, [and [[?

[("test" command) and [[("new test" command) are used to evaluate expressions. [[works only in Bash, Zsh and the Korn shell, and is more powerful; [and `test` are available in POSIX shells. Here are some examples:

```
#POSIX
[ "$variable" ] || echo 'variable is unset or empty!' >&2
[ -f "$filename" ] || printf 'File does not exist or is not a regular file: %s\n' "$filename" >&2
```

```
if [[ ! -e $file ]]; then
    echo "File doesn't exist or is in an inaccessible directory or is a symlink to a file that
doesn't exist: $file" >&2
fi

if [[ $file0 -nt $file1 ]]; then
    printf 'file %s is newer than %s\n' "$file0" "$file1"
fi
```

To cut a long story short: `test` implements the old, portable syntax of the command. In almost all shells (the oldest Bourne shells are the exception), [is a synonym for `test` (but requires a final argument of]). Although all modern shells have built-in implementations of [, there usually still is an external executable of that name, e.g. `/bin/[`. POSIX defines a mandatory feature set for [, but almost every shell offers extensions to it. So, if you want portable code, you should be careful not to use any of those extensions.

[[is a new, improved version of it, and it is a keyword rather than a program. This makes it easier to use, as shown below. [[is understood by KornShell, Zsh and BASH (as of version 2.03), but not by other POSIX shell implementations (e.g. posh, yash, or dash) or the BourneShell.

Although [and [[have much in common and share many expression operators like "`-f`", "`-s`", "`-n`", and "`-z`", there are some notable differences. Here is a comparison list:

Feature	new test [[	old test [	Example
string comparison	>	\> (*)	`[[ a > b ]] || echo "a does not come after b"`
	<	\< (*)	`[[ az < za ]] && echo "az comes before za"`
	= (or ==)	=	`[[ a = a ]] && echo "a equals a"`
	!=	!=	`[[ a != b ]] && echo "a is not equal to b"`
integer comparison	-gt	-gt	`[[ 5 -gt 10 ]] || echo "5 is not bigger than 10"`
	-lt	-lt	`[[ 8 -lt 9 ]] && echo "8 is less than 9"`
	-ge	-ge	`[[ 3 -ge 3 ]] && echo "3 is greater than or equal to 3"`
	-le	-le	`[[ 3 -le 8 ]] && echo "3 is less than or equal to 8"`
	-eq	-eq	`[[ 5 -eq 05 ]] && echo "5 equals 05"`
	-ne	-ne	`[[ 6 -ne 20 ]] && echo "6 is not equal to 20"`
conditional evaluation	&&	-a (**)	`[[ -n $var && -f $var ]] && echo "$var is a file"`
	||	-o (**)	`[[ -b $var || -c $var ]] && echo "$var is a device"`
expression grouping	(...)	\(... \) (**)	`[[ $var = img* && ($var = *.png || $var = *.jpg) ]] && echo "$var starts with img and ends with .jpg or .png"`
Pattern matching	= (or ==)	(not available)	`[[ $name = a* ]] || echo "name does not start with an 'a': $name"`
RegularExpression matching	=~	(not available)	`[[ $(date) =~ ^Fri\ ...\ 13 ]] && echo "It's Friday the 13th!"`

(*) This is an extension to the POSIX standard; some shells may have it, others may not.

(**) The `-a` and `-o` operators, and `( ... )` grouping, are defined by POSIX but only for strictly limited cases, and are marked as deprecated. Use of these operators is discouraged; you should use multiple [commands instead:

- `if [ "$a" = a ] && [ "$b" = b ]; then ...`
- `if [ "$a" = a ] || { [ "$b" = b ] && [ "$c" = c ];}; then ...`

Special primitives that `[[` is defined to have, but `[` may be lacking (depending on the implementation):

Description	Primitive	Example
entry (file or directory) exists	`-e`	`[[ -e $config ]] && echo "config file exists: $config"`
file is newer/older than other file	`-nt` / `-ot`	`[[ $file0 -nt $file1 ]] && echo "$file0 is newer than $file1"`
two files are the same	`-ef`	`[[ $input -ef $output ]] && { echo "will not overwrite input file: $input"; exit 1; }`
negation	`!`	`[[ ! -u $file ]] && echo "$file is not a setuid file"`

But there are more subtle differences.

- No WordSplitting or glob expansion will be done for `[[` (and therefore many arguments need not be quoted):

  ```
  file="file name"
  [[ -f $file ]] && echo "$file is a regular file"
  ```

  will work even though `$file` is not quoted and contains whitespace. With `[` the variable needs to be quoted:

  ```
  file="file name"
  [ -f "$file" ] && echo "$file is a regular file"
  ```

  This makes `[[` easier to use and less error-prone.
- Parentheses in `[[` do not need to be escaped:

  ```
  [[ -f $file1 && ( -d $dir1 || -d $dir2 ) ]]
  [ -f "$file1" -a \( -d "$dir1" -o -d "$dir2" \) ]
  ```

- As of bash 4.1, string comparisons using < or > respect the current locale when done in `[[`, but **not** in `[` or `test`. In fact, `[` and `test` have *never* used locale collating order even though past man pages *said* they did. Bash versions prior to 4.1 do not use locale collating order for `[[` either.

As a rule of thumb, `[[` is used for strings and files. If you want to compare numbers, use an ArithmeticExpression, e.g.

```
# Bash
i=0
while (( i < 10 )); do ...
```

When should the new test command `[[` be used, and when the old one `[`? If portability/conformance to POSIX or the BourneShell is a concern, the old syntax should be used. If on the other hand the script requires BASH, Zsh, or KornShell, the new syntax is usually more flexible, but not necessarily backwards compatible.

For reasons explained in the theory section below, any problem with an operator used with `[[` is an unhandleable parse-time error that will cause bash to terminate, even if the command is never evaluated.

```
# Example of improper [[ usage.
# Notice that isSet is never even called.

 $ bash-3.2 <<\EOF
if ((BASH_VERSINFO[0] > 4 || (BASH_VERSINFO[0] == 4 && BASH_VERSINFO[1] >= 2))); then
  isSet() { [[ -v $1 ]]; }
else
  isSet() { [[ ${1+_} ]]; }
fi
EOF
bash-3.2: line 2: conditional binary operator expected
bash-3.2: line 2: syntax error near `$1'
bash-3.2: line 2: `  isSet() { [[ -v $1 ]]; }'
```

If backwards-compatibility were desired then `[ -v` should have been used instead. The only other alternatives would be to use an alias to conditionally expand during the function definition, or `eval` to defer parsing until the command is actually reached at runtime.

See the Tests and Conditionals chapter in the BashGuide.

Theory

The theory behind all of this is that `[` is a simple command, whereas `[[` is a compound command. `[` receives its arguments as any other command would, but most compound commands introduce a special parsing context which is performed before any

other processing. Typically this step looks for special `reserved words` or `control operators` specific to each compound command which split it into parts or affect control-flow. The Bash test expression's logical and/or operators can short-circuit because they are special in this way (as are e.g. `;;`, `elif`, and `else`). Contrast with ArithmeticExpression, where all expansions are performed left-to-right in the usual way, with the resulting string being subject to interpretation as arithmetic.

- The arithmetic compound command has no special operators. It has only one evaluation context - a single arithmetic expression. Arithmetic expressions have operators too, some of which affect control flow during the arithmetic evaluation step (which happens last).

```
# Bash
(( 1 + 1 == 2 ? 1 : $(echo "This doesn't do what you think..." >&2; echo 1) ))
```

- Test expressions on the other hand do have "operators" as part of their syntax, which lie on the other end of the spectrum (evaluated first).

```
# Bash
[[ '1 + 1' -eq 2 && $(echo "...but this probably does what you expect." >&2) ]]
```

- Old-style tests have no way of controlling evaluation because its arguments aren't special.

```
# Bash
[ $((1 + 1)) -eq 2 -o $(echo 'No short-circuit' >&2) ]
```

- Different error handling is made possible by searching for special compound command tokens before performing expansions. `[[` can detect the presence of expansions that don't result in a word yet still throw an error if none are specified. Ordinary commands can't.

```
# Bash
( set -- $(echo 'Unquoted null expansions do not result in "null" parameters.' >&2); echo $# )
[[ -z $(:) ]] && echo "-z was supplied an arg and evaluated empty."
[ -z ] && echo "-z wasn't supplied an arg, and no errors are reported. There's no possible way
Bash could enforce specifying an argument here."
[[ -z ]] # This will cause an error that ordinary commands can't detect.
```

- For the very same reason, because `[`'s operators are just "arguments", unlike `[[`, you *can* specify operators as parameters to an ordinary `test` command. This might be seen as a limitation of `[[`, but the downsides outweigh the good almost always.

```
# ksh93

args=(0 -gt 1)

(( $(print '0 > 1') )) # Valid command, Exit status is 1 as expected.
[ "${args[@]}" ]       # Also exit 1.
[[ ${args[@]} ]]       # Valid command, but is misleading. Exit status 0. set -x reveals the
resulting command is [[ -n '0 -gt 1' ]]
```

- Do keep in mind which operators belong to which shell constructs. Order of expansions can cause surprising results especially when mixing and nesting different evaluation contexts!

```
# ksh93
typeset -i x=0

( print "$(( ++x, ${ x+=1; print $x >&2;}1, x ))"      ) # Prints 1, 2
( print "$(( $((++x)), ${ x+=1; print $x >&2;}1, x ))" ) # Prints 2, 2 - because expansions are
performed first.
```

CategoryShell