char*

```
char* my_string = "hello";
my_string + 1; // points to " -
ell o"
my_string + 2; // points to " -
llo "
```

A pointer to a character, typically used to point to the start of a string.

Strings in C are just arrays of characters ending with a null terminator \0.

Adding a number to the pointer moves it forward by that many characters.

sleep

```
sleep 1 # pause for 1 second
sleep 30 # pause for 30 seconds
sleep 0.5 # pause for half a
second
```

Pauses the script for a specified number of seconds.

Used to simulate delays, timeouts, or hung processes in testing.

/dev/null

```
echo "discard this" > /dev/null
# output is discarded
command 2> /dev/null # discard
error output
```

A special file on Linux that discards anything written to it.

Commonly used to suppress unwanted output.

Array Declaration ()

```
my_array=("first" "second"
"third")
echo ${my_a rra y[0]} # prints
" fir st"
echo ${my_a rra y[1]} # prints
" sec ond "
echo ${my_a rra y[2]} # prints
" thi rd"
```

Parentheses are used to declare an array in bash.

$@

```
#!/usr/bin/env bash
echo " $@" # prints all
arguments
some_c ommand " $@" # passes all
arguments to another command
```

Represents all arguments passed to a script or function.

Quoting it with "$@" preserves arguments that contain spaces as single units.

Commonly used to pass all arguments along to another command or store them.

Length Operator

```
my_string="hello"
${#my_ string} # 5 (number of
charac ters)
my_arr ay= ("a" " b" " c")
${#my_ arr ay[@]} # 3 (number of
elements)
```

When used inside ${}, the # returns the length of a variable or array.

For strings it returns the number of characters, for arrays the number of elements.

;

```
command1 ; command2 # command2
runs regardless
command1 && command2 # command2
only runs if command1 succeeded
rm -f " $sr c" ; exit 0 # delete
file then exit
```

Separates two commands on the same line, running them sequentially.

The second command always runs regardless of whether the first succeeded or failed.

Different from && which only runs the second command if the first succeeded.

exit

```
exit 0 # success
exit 1 # generic failure
exit 23 # custom failure code
```

""

```
"${MY_VARIABLE}" # preserved as
single unit
some_c ommand " $@" # preserves
arguments with spaces
```

Quotes preserve a value as a single unit if it contains spaces.

Without quotes, spaces would split the value into separate arguments.

${}

```
${MY_VARIABLE}
echo " Hello ${NAME }wo rld " #
without braces NAME and world
would merge
```

Curly braces denote a variable expansion in bash.

Used to explicitly mark the boundaries of a variable name.

Required when combining a variable with other characters.

Array Indexing

```
my_array=("first" "second"
"third")
# [0] [1] [2]
${my_a rra y[0]} # " fir st"
${my_a rra y[2]} # " thi rd"
(last element)
${my_a rra y[$ {#m y_a rra y[@]}
- 1]} # " thi rd" (last element
dynami cally)
${my_a rra y[$ {#m y_a rra y[@]}
- 2]} # " sec ond " (second to
last dynami cally)
```

Arrays in bash start at index 0, not 1.

The last element is always at index length - 1.

The second to last element is at index length - 2.

$(())

Each space-separated value becomes its own element, accessible by index starting at 0.

"$@" can be used to populate an array with all script arguments.

case ... in ... esac

```
case "${MY_VARIABLE}" in
        <pa tte rn1 >) do_som -
ething ;;
        <pa tte rn2 >) do_som -
eth ing _else ;;
        <*>) defaul t_a ction ;;
esac
```

Bash's switch statement, matching a value against a set of patterns.
esac closes the case block (it's "case" spelled backwards).
Each pattern ends with) and each case ends with ;;.

Note: *) is the catch-all pattern in a case statement, matching anything not already matched. Always placed last so specific patterns are checked first.

Exits the script with a return code.

0 means success, any non-zero value means failure.

The exit code can be checked by the calling process to determine if the script succeeded.

```
$(( 5 + 2 )) # 7
$(( 10 - 3 )) # 7
$(( 4 * 2 )) # 8
```

Performs arithmetic in bash.

The result of the calculation is used as a value.

>>

```
echo "new line" >>
existing_file.txt # appends to
file
echo "new line" > existi ng_ -
fil e.txt # overwrites file
```

Appends output to a file without overwriting existing content.

Use > instead to overwrite the file completely.

>&2

```
echo "something went wrong" >&2
# sends to stderr
echo " normal output " # sends
to stdout
```

Redirects output to stderr (standard error) instead of stdout (standard output).

Used for error messages so they can be separated from normal output.

:-

```
${MY_VARIABLE:-default_value} #
uses default_value if
MY_VARIABLE is unset
${MY_V ARI ABL E:- /so me/ path}
# uses /some/path if MY_VAR -
IABLE is unset
```

Provides a default value if the variable is unset or empty.

The value after :- is used as a fallback.

rm

```
rm file.txt # removes file,
errors if not found
rm -f file.txt # removes file,
no error if not found
rm -rf directory/ # removes
directory and all contents
forcefully
```

Removes a file from the filesystem.

Without flags it will prompt for confirmation on protected files and error on missing files.

A `-f` flag forces deletion — suppresses all prompts and ignores missing files, making it safe to use in scripts where the file may or may not exist.