

# Bash

- [Bash Profiles](#)
- [Examples](#)

# Bash Profiles

The following block contains a list of files related to bash, and their location / use.

`/bin/bash`

□ The bash executable

`/etc/bash.bashrc`

□ The system-wide bashrc for interactive bash shells, invoked on any login to an interactive shell.

`/etc/skel/.bashrc`

□ Used as a template for new users when initializing a basic `.bashrc` in their home directory.

`/etc/profile`

□ The systemwide initialization file, executed for login shells

`/etc/bash.bash_logout`

□ The systemwide login shell cleanup file, executed when a login shell exits

`~/.bash_profile`

□ The personal initialization file, executed for login shells

`~/.bashrc`

□ The individual per-interactive-shell startup file

`~/.bash_aliases`

□ An optional file sourced by `.bashrc` by default

`~/.bash_logout`

□ The individual login shell cleanup file, executed when a login shell exits

For more help, you can refer to the references and examples in `/usr/share/doc/bash/`, or if you don't have these files you can download them on Ubuntu with `sudo apt install bash-doc` and see the `/usr/share/doc/bash-doc/` directory. To help you explore these files, consider installing the terminal file browser `ranger` with `sudo apt install ranger`. If it is a `.html` file, open it in a web browser to browse the file easily.

# Creating Shells

From within `man bash`, we can find the following explanation for the creation of an interactive bash shell -

“ When bash is invoked as an interactive login shell, or as a non-interactive shell with the `--login` option, it first reads and executes commands from the file `/etc/profile`, if that file exists. After reading that file, it looks for `~/.bash_profile`, `~/.bash_login`, and `~/.profile`, in that order, and reads and executes commands from the first one that exists and is readable. The `--noprofile` option may be used when the shell is started to inhibit this behavior.

What this means is when a bash session is started that allows the user to interact with it by reading *and* writing, it will read from the `/etc/profile`. After reading from this file, it will look for one of three files within the user's home directory and read the first one that exists. This means that we can use the `/etc/profile` file to set system-wide settings for all interactive terminal sessions. For me, this is useful to set the editor for all users to default to Vim with the exports to `EDITOR` and `VISUAL` below - if they want to override it, they can.

## System Profile

First, the default `/etc/profile/` can be seen in the code block below. I wrote some comments in the file to explain what the script is doing.

```
# /etc/profile: system-wide .profile file for the Bourne shell (sh(1))
# and Bourne compatible shells (bash(1), ksh(1), ash(1), ...).

# This line sets the system-wide default text editor to vim
export EDITOR='/usr/bin/vim'
export VISUAL='/usr/bin/vim'

if [ "${PS1-}" ]; then
  if [ "${BASH-}" ] && [ "$BASH" != "/bin/sh" ]; then
    # The file bash.bashrc already sets the default PS1.
    # PS1='\h:\w\$ '
    if [ -f /etc/bash.bashrc ]; then
      . /etc/bash.bashrc
    fi
  else
    if [ "`id -u`" -eq 0 ]; then
```

```

PS1='# '
# This block allows for configuring any user whos id == 0
# In other words, these settings will be applied to the root user only.
else
    PS1='$ '
    # These settings will apply in all other cases, system-wide
    # In other words, upon successful login to an authroized user who is not root, this block will be executed
fi
fi
fi

[]# If the directory /etc/profile.d/ exists, source every file within it
# + See this directory for system defaults for interactive login shells for various programs
if [ -d /etc/profile.d ]; then
    for i in /etc/profile.d/*.sh; do
        if [ -r $i ]; then
            . $i
        fi
    done
    unset i
fi

```

The above `/etc/profile` configuration will set the default editor to vim, system-wide, regardless of which user is logged in. *This includes the root user.* Users can choose to override this in their own `~/.bashrc`, but users won't be prompted to select their default editor since the system will now use Vim by default.

If you want to specify which user, or if you want to handle the root user independent from the rest of the system, take a closer look at the comments I've added in the above configuration file and modify as needed. You could specify a user ID here to source additional files, or you could just handle that sort of thing in that user's `~/.bashrc`.

If you are trying to use the default text editor for any command ran with `sudo`, be sure that you pass either the `-E` or `--preserve-env` argument. So, if we wanted to preserve our environment settings for the default text editor Vim when running `vigr` or `visudo` we would simply run `sudo -E vigr` or `sudo --preserve-env visudo` to ensure these settings are referred to when using `sudo`

## User Profiles

After reading from `/etc/profile/`, bash looks for one of three files - `~/.bash_profile`, `~/.bash_login`, and `~/.profile`, in that order. The first file that exists is sourced and bash stops looking. For evidence of this, notice the comments in the first few lines of the `~/.profile` file described by `man bash` that points out the file's order of execution. Just after, within the first condition of the file, it becomes

obvious where `~/.bashrc` comes into play and things start to come to an end -

```
# ~/.profile: executed by the command interpreter for login shells.
# This file is not read by bash(1), if ~/.bash_profile or ~/.bash_login
# exists.
# see /usr/share/doc/bash/examples/startup-files for examples.
# the files are located in the bash-doc package.

# the default umask is set in /etc/profile; for setting the umask
# for ssh logins, install and configure the libpam-umask package.
#umask 022

# if running bash
if [ -n "$BASH_VERSION" ]; then
    # Include ~/.bashrc if it exists
    if [ -f "$HOME/.bashrc" ]; then
        . "$HOME/.bashrc"
    fi
fi

# set PATH so it includes user's private bin if it exists
# + Any executables added to this directory will exist on your PATH
if [ -d "$HOME/bin" ] ; then
    PATH="$HOME/bin:$PATH"
fi

# set PATH so it includes user's private bin if it exists (Alternate path)
# + Any executables added to this directory will exist on your PATH
if [ -d "$HOME/.local/bin" ] ; then
    PATH="$HOME/.local/bin:$PATH"
fi
```

All my `~/.profile` is doing above is sourcing the `~/.bashrc` file if it exists, and then adding some default directories to my user's `PATH`, if they exist. On different systems this can be handled differently. For example, below is an example of the same thing happening in a `~/.bash_profile` -

```
if [ -f ~/.bashrc ]; then . ~/.bashrc; fi
```

So we know now that when you want to edit settings for certain users who invoke their own interactive shells, the `~/.bashrc` file should be created or reconfigured. The rest of the page below will show some basic syntax for editing the `~/.bashrc` file, along with some examples.

# Interactive Shells

An interactive shell is one that can read and write to the user's terminal. This means that bash can take input from the user and provide some input back to them as a result. As described on the [GNU Bash documentation](#), these shells often define the `PS1` variable which we will cover later. This variable describes how the user's bash prompt should appear within their session, and can often be fun or useful to customize. To start an interactive shell, you often use a **login shell** since you need to first authenticate with the system. On some more feature-rich systems though, you *can* start an interactive shell as a **non-login shell**, for example if you run a terminal application and you are already logged in - you are starting a new interactive shell without logging in, so you are in a non-login interactive shell.

## Non-interactive Shells

An example of a non-interactive shell is one which does not take input and often does not provide output. An example of these could be running a script, when we invoke the script we start a new shell that runs that script - this shell is non-interactive. These shells do require login, since they are invoked by users who are already logged in, so they are also considered to be a **non-login shell**.

# Skeleton Configurations

As stated in the first section, the `/etc/skel/` directory contains files that are distributed to each new user created on our system. This is useful to know, since we can directly modify these files to provide different default configurations provided when new users are created. This can be a nice way to ensure that all users start with the same aliases, or are shown a similar prompt. We can even specify other defaults here, like providing a default `.vimrc` to distribute to new users, or setting certain shell options.

## Customizing Bashrc

Once logged in as your bash user, you can adjust your personal bash settings by modifying `~/.bashrc`, or `/home/username/.bashrc`. If the file doesn't exist, you can just create it and follow along with no additional setup required. If this file exists, it can at first be a lot to look at, but some of the more important lines to consider are seen below -

### Bash prompt

```
# This controls how your prompt looks within terminals logged in as your user
if [ "$color_prompt" = yes ]; then

PS1='${debian_chroot:+($debian_chroot)}\[\033[01;32m\]\u@\h\[\033[00m\]:\[\033[01;34m\]\w\[\033[00m\]\$ '
else
```

```
PS1='${debian_chroot:+($debian_chroot)}\u@\h:\w\$ '
fi
```

## Alias / export customizations

```
# some more ls aliases
alias ll='ls -alF'
alias la='ls -A'
alias l='ls -CF'
```

## Additional files to source

```
# You may want to put all your additions into a separate file like
# ~/.bash_aliases, instead of adding them here directly.
# See /usr/share/doc/bash-doc/examples in the bash-doc package.
if [ -f ~/.bash_aliases ]; then
    . ~/.bash_aliases
fi
```

## Auto-completion

```
# enable programmable completion features (you don't need to enable
# this for each user, if it's already enabled in /etc/bash.bashrc and /etc/profile
if ! shopt -oq posix; then
    if [ -f /usr/share/bash-completion/bash_completion ]; then
        . /usr/share/bash-completion/bash_completion
    elif [ -f /etc/bash_completion ]; then
        . /etc/bash_completion
    fi
fi
```

# Environment Variables

**PS1:** Environment variable which contains the value of the default prompt. It changes the shell command prompt appearance.

```
kapper@kubuntu-vbox $ export PS1='[\u@\h \W]\$'
[kapper@kubuntu-vbox ~]$
```

**PS2:** Environment variable which contains the value the prompt used for a command continuation interpretation. You see it when you write a long command in many lines. In most cases, this is set to `>`, and is seen below after using the `\` character to break the command into several lines -

```
[kapper@kubuntu-vbox ~]$ export PS2='--> '  
[kapper@kubuntu-vbox ~]$ cp /some/really/long/system/path/fileOne \  
--> fileTwo
```

**PS3:** Environment variable which contains the value of the prompt for the select operator inside the shell script.

**PS4:** Environment variable which contains the value of the prompt used to show script lines during the execution of a bash script in debug mode. This could be used to show the line number at the current point of execution -

```
# $0 is the current file being executed, $LINENO is the current line number  
[kapper@kubuntu-vbox ~]$ export PS4='$0:$LINENO'  
[kapper@kubuntu-vbox ~]$ bash -x fix-vbox.sh  
fix-vbox.sh:5grep 'VBoxClient --draganddrop'  
fix-vbox.sh:6awk '{print $2}'  
fix-vbox.sh:7xargs kill  
fix-vbox.sh:8ps aux www
```

**PROMPT\_COMMAND:** Environment variable which contains command(s) to run before printing the prompt within the terminal.

```
[kapper@kubuntu-vbox ~]$export PROMPT_COMMAND='echo -n "$(date): " && pwd'  
Sun 12 Sep 2021 05:00:55 PM EDT: /home/kapper  
[kapper@kubuntu-vbox ~]$ls  
Desktop  Music  Pictures  Videos  
Code     Public  Documents  Downloads  
Sun 12 Sep 2021 05:01:02 PM EDT: /home/kapper
```

## Bash Aliases

Create a list of aliases within your home directory inside a file named `.bash_aliases`, and add any custom aliases or `PATH` modifications there. The file may not exist, and if it doesn't just create one and start listing aliases or settings. This way when you want to adjust something like your `PATH` or aliases, you don't have to dig through all the contents of `.bashrc`. For example, some of the contents of my `~/.bash_aliases` is seen below. This file will automatically be sourced by bash when logging into our user, in addition to the contents of the `~/.bashrc`.

```
# Alias / export customizations  
alias gitkapp='git config --global user.name "Shaun Reed" && git config --global user.email  
"shaunrd0@gmail.com"'
```



```
# colored GCC warnings and errors
export GCC_COLORS='error=01;31:warning=01;35:note=01;36:caret=01;32:locus=01:quote=01'

# some more ls aliases
alias ll='ls -aF'
alias la='ls -A'
alias l='ls -CF'
```

The `gitkapp` alias above is a quick way of telling git who I am when logged in as a new user. You could imagine having more versions of this alias to switch to different git users quickly. Alternatively, you could use the `git config --local ...` command within the alias to automate configuring a specific repository for a certain user in a single command without modifying your global git user. Aliases even automatically show up using auto completion -

```
[user@host ~]$git
git          git-shell      git-upload-pack
git-receive-pack  git-upload-archive  gitkapp
[user@host ~]$gitkapp
```

## Identifying Unicode Symbols for use in .bashrc

### Character search engine

If you don't have access to a terminal, you can [search up a symbol to get UTF8](#), see the below character and the corresponding UTF8 format as an example. ☐ = 0xE0 0xB4 0xBD

To output this symbol in a bash terminal using this hex value, we can test with `echo` -

```
echo -e "\xe0\xb4\xbd"
```

Note that these hexadecimal values are not case sensitive.

### Hexdump unicode symbol

Most linux systems already have `hexdump` installed, so we could also run `echo ✓ | hexdump -C` to see the following output. Note that the `-C` option displays character data in hexadecimal ascii format -

```
[kapper@kubuntu-vbox ~]$echo ✓ | hexdump -C
00000000 e2 9c 93 0a                |....|
00000004
```

From this output, we can see that the UTF8 hexadecimal format of our symbol is `e2 9c 93`. Using this information, we can test the character with the `echo` statement below.

```
echo -e '\xe2\x9c\x93'
```

This will output our ✓ symbol, colored green. `\001\033[1;32m\002` Begins the green color, and `\001\033[0m\002` returns to

```
echo -e '\001\033[1;32m\002\xe2\x9c\x93\001\033[0m\002'
```

## Unicode.vim

Worth mentioning that if you are using vim, an easy to use plugin that is useful for identifying characters is `unicode.vim`. See my notes on [Unicode vim plugin](#) for more information, or check out the [official Unicode vim repository](#).

In any case, when using special characters and symbols in an assignment to `PS1`, you need to tell bash to interpret these values with a `$` before opening your single-quotes, as in `export PS1='${\xe2\x9c\x93}'`

# Bash Prompt

Your bash prompt is seen before you type a command -

```
user@host: ~/$
```

The prompt above, `user@host: ~/$`, is defined by the `PS1` variable within your `~/.bashrc` where `\u` is your username `user`, and `\h` is the hostname `host` in the prompt above. The `\w` in the prompt is what places our current directory `~/` before the final `$` within the prompt -

```
# Bash prompt settings

if [ "$color_prompt" = yes ]; then

PS1='${debian_chroot:+($debian_chroot)}\[\033[01;32m\]\u@\h\[\033[00m\]:\[\033[01;34m\]\w\[\033[00m\]\$ '
else
    PS1='${debian_chroot:+($debian_chroot)}\u@\h:\w\$ '
fi
```

By default within your `.bashrc` there's two settings to configure, the first block includes color, the second does not. When first learning about the prompt and all the available options like `\u`, `\h`, and `\w`, it might be easier to look at the second prompt without the escape sequences for adding color. As we will see later, care must be taken to properly escape non-printing characters within your prompt, specifically color codes. That is the meaning of character sequences like `\[\033[01;32m\]` or `\001\033[01;32m\002`. Later we will cover the meaning of these symbols and how to

properly organize them within your prompt.

You can change this prompt using the variety of settings below. Test your prompts with `export PS1='<YOUR_PROMPT_HERE>'` and after you've got a good export working, paste it into the `~/.bashrc` to apply your changes each time you login. If you do not put the `PS1` assignment within your `~/.bashrc` and log out of your terminal with an export applied, when you login it will be overwritten by the code above.

The `${debian_chroot:+($debian_chroot)}` portion of `PS1` above only impacts our shell when we are using a `chroot`, which is a way of changing the root directory of the system into a smaller virtualized environment that exists within the system. So if we are using a `chroot`, then we will see the following prompt -

```
(chroot-name)user@host:~/$
```

You can remove this `${debian_chroot:+($debian_chroot)}` portion or leave it, entirely up to you.

## Prompt Options

When setting your bash prompt, we have the following options available to use. Options are useful for getting information from the current bash session dynamically. For example, `\u` can be used to place the current username in the prompt, and `\h` will print the hostname. So the prompt `export PS1='\u@\h:'` will make our prompt `username@hostname:`

- `\a` The ASCII bell character (you can also type `\007`)
- `\d` Date in "Sat Sep 04" format
- `\e` ASCII escape character (you can also type `\033` or `\x1B`)
- `\h` First part of hostname (such as "mybox")
- `\H` Full hostname (such as "mybox.mydomain.com")
- `\j` The number of processes you've suspended in this shell by hitting `^Z`
- `\l` The name of the shell's terminal device (such as "tty4")
- `\n` Newline
- `\r` Carriage return
- `\s` The name of the shell executable (such as "bash")
- `\t` Time in 24-hour format (such as "23:59:59")
- `\T` Time in 12-hour format (such as "11:59:59")
- `\@` Time in 12-hour format with am/pm
- `\u` Your username
- `\v` Version of bash (such as 2.04)
- `\V` Bash version, including patchlevel
- `\w` Current working directory (such as "/home/kapper")
- `\W` The "basename" of the current working directory (such as "kapper")
- `\!` Current command's position in the history buffer

\# Command number (this will count up at each prompt, as long as you type something)

\\$ If you are not root, inserts a “\$”; if you are root, you get a “#”

\xxx Inserts an ASCII character based on three-digit number xxx (replace unused digits with zeros, such as “\007”)

\\ A backslash

[ This sequence should appear before a sequence of characters that don’t move the cursor (like color escape sequences). This allows bash to calculate word wrapping correctly.

] Same as \002, This sequence should appear after a sequence of non-printing characters.

□

\001 can be used directly in place of [ and is recommended as a more portable option

\002 can be used directly in place of ] and is recommended as a more portable option

## Background color codes

This section will cover using escape sequences to change the background color used within your bash prompt. This will have the effect of 'highlighting' the text in a certain color.

The following sequences can be used to set attributes that impact the background color of text print within a bash terminal. Notice that each color has a corresponding light color by changing the leading 4 to a 10. For example, in the color sequence [42m and [102m for green and light green background colors, respectively -

Default color	\001\033[0;49m\002		
Black	\001\033[0;40m\002	White	\001\033[0;107m\002
Light Gray	\001\033[0;47m\002	Dark Gray	\001\033[0;100m\002
Red	\001\033[0;41m\002	Light Red	\001\033[0;101m\002
Green	\001\033[0;42m\002	Light Green	\001\033[0;102m\002
Yellow	\001\033[0;43m\002	Light Yellow	\001\033[0;103m\002
Blue	\001\033[0;44m\002	Light Blue	\001\033[0;104m\002
Magenta	\001\033[0;45m\002	Light Magenta	\001\033[0;105m\002
Cyan	\001\033[0;46m\002	Light Cyan	\001\033[0;106m\002

## Foreground color codes

This section will cover using escape sequences to change the font color used within your bash prompt

Using the appropriate bash syntax and the codes below, the `\001\033[32m\002` escape code will colorize everything green after until output is reset with `\001\033[0m\002`. Technically, the color code is only the `[32m` portion, but it needs to be enclosed in `\001\033` and `\002`. `\001\033` is the more portable option for `\[e`, and `\002` is the more portable option for `\]`.

So `\001\003[32m\002` is both technically equivalent to and more portable than `\[e[32m\]`

Also, the next section covers attributes, which make up the `0` in `\[e[0;32m\]`. So any attribute can be applied to any color by changing this leading value, or the `0;` can be removed entirely if normal text is used, as in `\[e[32m\]`.

The following sequences can be used to set attributes that impact the color of text in a bash terminal. Notice that each color has a corresponding light color by changing the leading `3` to a `9`. For example, in the color sequence `[32m` and `[92m` for green and light green, respectively -

Default color	\001\033[0;39m\002		
Black	\001\033[0;30m\002	White	\001\033[0;97m\002
Light Gray	\001\033[0;37m\002	Dark Gray	\001\033[0;90m\002
Red	\001\033[0;31m\002	Light Red	\001\033[0;91m\002
Green	\001\033[0;32m\002	Light Green	\001\033[0;92m\002
Yellow	\001\033[0;33m\002	Light Yellow	\001\033[0;93m\002
Blue	\001\033[0;34m\002	Light Blue	\001\033[0;94m\002
Magenta	\001\033[0;35m\002	Light Magenta	\001\033[0;95m\002
Cyan	\001\033[0;36m\002	Light Cyan	\001\033[0;96m\002

## Reset attributes

The following sequences can be used to reset attributes that impact the appearance of text in a bash terminal, returning them to normal after the attribute was previously set. Note that the reset is technically only `[0m` but these also need to be wrapped in `\001\033` and `\002` -

Reset all attributes	\001\033[0m\002
----------------------	-----------------

Reset bold and bright	\001\033[21m\002
Reset dim	\001\033[22m\002
Reset underline	\001\033[24m\002
Reset blink	\001\033[25m\002
Reset reverse	\001\033[27m\002
Reset hidden	\001\033[28m\002

## Set attributes

Any attribute can be applied to any color by changing the leading `0;`, or the attribute value can be removed entirely and the current attribute settings are used, as in `\[\e[32m\]`.

The following sequences can be used to set attributes that impact the appearance of text in a bash terminal. Note that the set is technically only `[1m` but these also need to be wrapped in `\001\033` and `\002` -

Set bold and bright	\001\033[1m\002
Set dim	\001\033[2m\002
Set underline	\001\033[4m\002
Set blink	\001\033[5m\002
Set reverse	\001\033[7m\002
Set hidden	\001\033[8m\002

## Prompt Examples

Any of the below exports can be pasted directly into the terminal to be tested. Once the terminal is closed, these settings will be lost, so no worries about getting back to default. This is a good way to test what would happen if you changed the PS1 within your `~/.bashrc`, without actually doing so. If you mess up too bad, just close your terminal and open a new one. If you are logged in via `ssh`, you'll have to either `source ~/.bashrc` or log out and back into the server.

Note that when using special characters and symbols, you need to tell bash to interpret these values with a `$` before opening your single-quotes, as in `export PS1=$'\xe2\x9c\x93'`

Note that we *do not* need to escape hexadecimal characters that will be interpreted. See the below for examples

```
# Ok
echo -e '\001\033[1;32m\002\xde\x90\x0a\001\033[0m\002'

# Wrong, no need to wrap symbol hex value with '\001' and '\002'
echo -e '\001\033[1;32m\002\001\xde\x90\x0a\002\001\033[0m\002'

# Wrong, hexadecimal symbol is wrapped within '\001' and '\002'
echo -e '\001\033[1;32m\xde\x90\x0a\033[0m\002'
```

When writing custom prompts, this can become a lot to take in all at once. The following prompt doesn't even use color codes yet, and already it is quite the line -

```
export
PS1=$'\xe2\x94\x8c\xe2\x94\x80\xe2\x94\x80u@h\xe2\x94\x80[\W]\n\xe2\x94\x94\xe2\x94\x80\xe2\x95\xbc$'
```

What I like to do is split the prompt between several append statements to `PS1` **within my** `.bashrc`. An example of this prompt split across multiple lines shows it is much more readable and easier to adjust -

```
# Printing ┐
PS1=""
PS1+=$'\xe2\x94\x8c'
PS1+=$'\xe2\x94\x80'
PS1+=$'\xe2\x94\x80'

# Printing kapper@kubuntu-vbox-[~]
PS1+='\u@h'
PS1+=$'\xe2\x94\x80'
PS1+='[\W]'

# Move to next line
PS1+=$'\n'

# Printing ┐─$
PS1+=$'\xe2\x94\x94'
PS1+=$'\xe2\x94\x80'
PS1+=$'\xe2\x95\xbc'
PS1+='\$'
```

Alternatively, for practice or playing around, we can create a new file called `.practice_prompt` with the following contents. Then, we can just save the file and run `source ~/.practice_prompt` from a different terminal to enable the custom prompt and see the changes -

```
# Printing ─
export PS1=""
export PS1+=${'\xe2\x94\x8c'}
export PS1+=${'\xe2\x94\x80'}
export PS1+=${'\xe2\x94\x80'}

# Printing kapper@kubuntu-vbox-[~]
export PS1+='\u@\h'
export PS1+=${'\xe2\x94\x80'}
export PS1+='\[W]'

# Move to next line
export PS1+=${'\n'}

# Printing └─$
export PS1+=${'\xe2\x94\x94'}
export PS1+=${'\xe2\x94\x80'}
export PS1+=${'\xe2\x95\xbc'}
export PS1+=`${'$'}
```

Splitting your PS1 assignment up not only makes it easier to read, but it suddenly becomes easy to comment out specific sections of the prompt when debugging issues with character spacing or adjusting the final appearance.

## Simple Prompt

We can create a bare-minimum and simple export like the below, before adding any color

```
# Example of what the prompt will look like
[kapper@kubuntu-vbox ~]$

# Export to use this prompt
export PS1='[\u@\h \W]\$'
```

## Colorized Prompt

Adding color to the prompt makes things look a bit more complicated, but if we stick to the rules outlined in the sections above we shouldn't have too much of an issue. Remember, if the prompt



gets too long feel free to split it up between multiple appending statements within a file, then source that file. An example of this is shown in the earlier sections.

```
# Example of what the prompt will look like
[kapper@kubuntu-vbox ~]$

# Export to use this prompt
export PS1='\001\033[1;32m\002[\u@\h\001\033[0m\002 \W\001\033[1;32m\002]\$\001\033[0m\002'
```

## Symbols in Prompt

Let's take the colors out for now, and use some symbols to create a more interesting prompt. This prompt is based on the default prompt from the Parrot linux distribution. This prompt will use special symbols, so to begin we use `echo └─┐ | hexdump -C` to get the below output.

```
[kapper@kubuntu-vbox ~]$echo └─┐ | hexdump -C
00000000 e2 94 8c 20 e2 94 94 20 e2 94 80 20 e2 95 bc 0a |... ..|
00000010
[kapper@kubuntu-vbox ~]$
```

Notice we passed three symbols with spaces between them. If we run the command `ascii` to see the ascii table, we can see that the value of the hexadecimal column for the space character is `20`. This is seen in the above output and helps to separate the hexadecimal values of our symbols so we can easily see where one begins and ends. We see that `└` is `e2 94 8c` followed by a space `20`, then `─` which is `e2 94 94`, and another space value of `20`. Next, the `┐` symbol is `e2 94 80`, followed by one more `20` and the final `|` symbol's hex value of `e2 95 bc`. We will need to place the hexadecimal values of our special characters in the position we want the symbol to appear within our `PS1` export.

Below, we use this information to *correctly* use symbols in our bash prompt. Note that while pasting the raw symbol will appear to work, it *will cause bugs in your prompt*. The method below requires more effort, but it will not cause character spacing issues within your prompt.

```
# Example of what the prompt will look like
└─┐kapper@kubuntu-vbox-[~]
└─┐$

# Export to use this prompt
export
PS1='${\xe2\x94\x8c\xe2\x94\x80\xe2\x94\x80\u@\h\xe2\x94\x80[\W]\n\xe2\x94\x94\xe2\x94\x80\xe2\x95\xbc}$'
```

## Symbols and colors in Prompt

Here's everthing together in one prompt.

```
# Example of what the prompt will look like
```

```
# NOTE: Color is lost here, but there will be color within your terminal
```

```
└─kapper@kubuntu-vbox-[~]
```

```
└─$
```

```
# Export to use this prompt
```

```
export
```

```
PS1='${001\033[1;31m\002\ue2\x94\x8c\ue2\x94\x80\ue2\x94\x80\001\033[1;32m\002\u@\h\001\033[1;31m\002\ue2\x94\x80\001\033[0m\002\W\001\033[1;31m\002]\n\ue2\x94\x94\ue2\x94\x80\ue2\x95\xbc\001\033[1;32m\002\$\001\033[0;39m\002'
```

# Examples

## Read the manual page for bash!

If needed, check out my not-so-brief [Introduction to Manual Pages](#) to learn how to reference these manual pages more efficiently.

I would also recommend the book [Bash Pocket Reference by Arnold Robbins](#), it is a pretty dense read but worth looking over. There are a lot of good examples, and it has actually been a pretty useful reference for me to keep close by. It isn't a book, but rather a collection of examples and concepts that are common in bash scripting.

## Redirecting Output

Knowing how to control your output streams in bash can help to make you much more effective at writing commands. For example, say you want to run `clion` on the CWD and fork the process to the background.

```
clion . &
[1] 178345
2021-12-18 16:13:28,384 [ 3869] WARN - I.NotificationGroupManagerImpl - Notification group CodeWithMe is
already registered (group=com.intellij.notification.NotificationGroup@68d6e24d). Plugin descriptor:
PluginDescriptor(name=Code With Me, id=com.jetbrains.codeWithMe, descriptorPath=plugin.xml,
path=~/.local/share/JetBrains/Toolbox/apps/CLion/ch-0/213.5744.254/plugins/cwm-plugin,
version=213.5744.254, package=null, isBundled=true)
2021-12-18 16:13:29,647 [ 5132] WARN - pl.local.NativeFileWatcherImpl - Watcher terminated with exit code
130
```

Woah! We've inherited the output from the process we forked to the background, and we've also lost immediate control of the process so `CTRL+C` won't interrupt the program and stop the output as it normally would. To fix this, run `fg` to bring the process back to the foreground, and then try pressing `CTRL+C` again. The process will terminate and the output will stop.

If we want to fork this process to the background and redirect all of its output so we don't need to see it, we can run the following command. Note that in this case, we are redirecting to `/dev/null` to throw away the output. If we wanted to, we could instead redirect to a file and log the output.

```
clion . 2>/dev/null 1>&2 &
```

Or, a shorter version, which is shorthand for redirecting all output.

```
clion . &>/dev/null &
```

If we want to redirect only standard output

```
clion 1>/dev/null &
```

And if we want to redirect only standard error

```
clion. 2>/dev/null
```

## Creating Scripts

Bash scripting is much like interacting with the bash terminal - the similarity can be easily seen in how we would split a bash command to multiple lines...

```
kapak@base:~$ \\  
> s -la  
total 76  
drwxr-xr-x 9 username username 4096 Jul 28 01:24 .  
drwxr-xr-x 3 root root 4096 Jul 6 09:49 ..  
-rw----- 1 username username 5423 Jul 20 18:10 .bash_history  
-rw-r--r-- 1 username username 220 Jul 6 09:49 .bash_logout  
-rw-r--r-- 1 username username 3771 Jul 6 09:49 .bashrc  
...( Reduced Output ) ...  
kapak@base:~$ ls -la  
total 76  
drwxr-xr-x 9 username username 4096 Jul 28 01:24 .  
drwxr-xr-x 3 root root 4096 Jul 6 09:49 ..  
-rw----- 1 username username 5423 Jul 20 18:10 .bash_history  
-rw-r--r-- 1 username username 220 Jul 6 09:49 .bash_logout  
-rw-r--r-- 1 username username 3771 Jul 6 09:49 .bashrc  
...( Reduced Output ) ...
```

In a bash script, we would handle splitting `ls -la` across multiple lines much the same. Create the file below, name it test.sh -

```
#!/bin/bash  
ls -la  
\br/>s -la
```

Now make the file executable and run the script, you should see the output of `ls -la` twice, since this script is a simple example of splitting commands across lines.

```
# Make the script executable
sudo chmod a+x test.sh

# Run the script
./test.sh
```

This of course isn't a common use case, but it shows how you can use `\` to effectively escape a newline and continue your command on the next line.

## Printf Formatting

This is just one of many commands in bash, but you will use it a lot so getting to know the syntax well will make your life a lot easier.

```
#!/bin/bash

## Author: Shaun Reed | Contact: shaunrd0@gmail.com | URL: www.shaunreed.com ##
## A custom bash script to configure vim with my preferred settings      ##
## Run as user with sudo within directory to store / stash .vimrc configs  ##
#####

#####

# Example of easy colorization using printf
GREEN=$(tput setaf 2)
RED=$(tput setaf 1)
UNDERLINE=$(tput smul)
NORMAL=$(tput sgr0)

# Script Reduced, lines removed

# Example of creating an array of strings to be passed to printf
welcome=( "\nEnter 1 to configure vim with the Klips repository, any other value to exit." \
"The up-to-date .vimrc config can be found here: https://github.com/shaunrd0/klips/tree/master/configs" \
"${RED}Configuring Vim with this tool will update / upgrade your packages${NORMAL}\n\n")

# Create a printf format and pass the entire array to it
# Will iterate through array, filling format provided with array contents
# Useful for printing / formatting lists, instructions, etc
printf '%b\n' "${welcome[@]}"

read cChoice

# Script Reduced, lines removed
```

[Full script](#)

Using the above method, you could easily create a single array containing multiple responses to related paths the script could take for a related option, and refer to the appropriate index of the array directly, instead of passing all of the contents of the array to the same format.

For more advanced formatting, read the below script carefully, and you will have a basic understanding of how printf can be used dynamically within scripts to provide consistent formatting.

```
#!/bin/bash

divider=====
divider=$divider$divider

header="\n %-10s %8s %10s %11s\n"
format=" %-10s %08d %10s %11.2f\n"

width=43

printf "$header" "ITEM NAME" "ITEM ID" "COLOR" "PRICE"

printf "%$width.${width}s\n" "$divider"

printf "$format" \
Triangle 13  red 20 \
Oval 204449 "dark blue" 65.656 \
Square 3145 orange .7

# https://linuxconfig.org/bash-printf-syntax-basics-with-examples
```

## String Manipulation

In bash, there are useful features to handle manipulating strings. These strings may be in any format, but the examples below will use strings that refer to directories and files as examples, since this is a common scenario.

```
#!/bin/bash

local teststring="/home/kapper/Code/"

echo "${teststring#/home/}"    # Remove shortest subtring from left matching pattern `/home/` (outputs
kapper/Code)

echo "${teststring##*kapper/}" # Remove longest subtring from left matching pattern `*/` (outputs Code/)
echo "${teststring%/*}"        # Remove shortest subtring from right matching pattern `/*` (outputs
/home/kapper/Code)
```

```

echo "${teststring%%kapper/*}" # Remove longest subtring from right matching pattern `kapper/*` (outputs
/home/)

# Can be used to obtain file name
local file="/home/kapper/.vimrc"
echo ${file##*/} # (outputs .vimrc)

# Can be used to obtain or strip file extensions
local other_file="/home/kapper/image.jpg"
echo ${other_file##*/} # (outputs image.jpg)
echo ${other_file%.*} # (outputs .jpg)

# Can piggy-back string manipulation, though it gets hard to read quickly
echo ${${other_file##*/}%.*} # (outputs image)

# Can remove up to first appearance of a space character (or tab, newline, other whitespace)
local white_space="sometextbeforeaspace /system/path/"
echo ${white_space%%[:space:]} # (outputs /system/path)

```

## Arrays

Arrays can be declared and initialized with the syntax below. This script will output the word `collection`

```

#!/bin/bash
local arr=(a collection of single words speparated by spaces will automatically be split into indexed array)
echo ${arr[1]}

```

Alternatively, arrays of strings that include spaces can be declared without splitting by simply using double-quotes. This script will output `This array has two elements`

```

#!/bin/bash
local arr=("This will not be split" "This array has two elements")
echo ${arr[1]}

```

Arrays can also be associative, if we also define names for each index when initializing. The script below will output `example`

```

#!/bin/bash
local arr=([zero]=some [one]=example [two]=array)

```

```
echo ${arr[one]}
```

In the script below, we use an array to search for the package coretemp sensor of our CPU.

```
#!/bin/bash
## Author: Shaun Reed | Contact: shaunrd0@gmail.com | URL: www.shaunreed.com ##
##                                     ##
## A script to find and return the CPU package temp sensor          ##
#####
#####
# bash.sh

for i in /sys/class/hwmon/hwmon*/temp*_input; do
    # Append each sensor to an array
    sensors+=("${(<$(dirname $i)/name): $(cat ${i%_*}_label 2>/dev/null || echo $(basename ${i%_*}))
$(readlink -f $i)");
done

# Loop through initialized array of hardware temp sensors
for i in "${sensors[@]}"
do
    # If the sensor is for the CPU core package temp, export to env variable
    if [[ $i =~ ^coretemp:.Package.* ]]
    then
        export CPU_SENSOR=${i#*0}
    fi
done

# Convert from C to F using our exported CPU_SENSOR variable
echo "scale=2;((9/5) * $(cat $CPU_SENSOR)/1000) + 32"|bc
```

## While Loops

While loop using conditionals within bash to automate cmake builds -

```
#!/bin/bash
## Author: Shaun Reed | Contact: shaunrd0@gmail.com | URL: www.shaunreed.com ##
## A custom bash script for building cmake projects.                ##
## Intended to be ran in root directory of the project alongside CMakeLists ##
#####
```



```
#####
```

```
# Infinite while loop - break on conditions
```

```
while true
```

```
do
```

```
    printf "\nEnter 1 to build, 2 to cleanup previous build, 0 to exit.\n"
```

```
    read bChoice
```

```
    # Build loop
```

```
    # If input read is == 1
```

```
    if [ $bChoice -eq 1 ]
```

```
    then
```

```
        mkdir build
```

```
        # Move to a different directory within a subshell and build the project
```

```
        # The '(' and ')' here preserves our working directory
```

```
        (cd build && cmake .. && cmake --build .)
```

```
    fi
```

```
    # Clean-up loop
```

```
    # If input read is == 2
```

```
    if [ $bChoice -eq 2 ]
```

```
    then
```

```
        printf "test\n"
```

```
        rm -Rv build/*
```

```
    fi
```

```
    # Exit loops, all other input -
```

```
    # If input read is >= 3, exit
```

```
    if [ $bChoice -ge 3 ]
```

```
    then
```

```
        break
```

```
    fi
```

```
    # If input read is <= 0, exit
```

```
    if [ $bChoice -le 0 ]
```

```
    then
```

```
        break
```

```
    fi
```

```
# Bash will print an error if symbol or character input
```

```
done
```

## Subshells

Sometimes, you may want to stay in your active directory but perform some action in another. To do this, we can do something in a subshell (a background shell) -

```
$ (cd /var/log && cp -- *.log ~/Desktop)
```

## Examples

Check out my snippet repository for more up-to-date scripts, configurations - [/shaunrd0/klips](#)

Basic script example of adding a user to a Linux system. This script uses parameters, basic formatting, and can either be ran as root or as a user without sudo, depending on the need. If needed, see the Knoats Book [Adding Linux Users](#) for more explanation on the below.

```
#!/bin/bash
## Author: Shaun Reed | Contact: shaunrd0@gmail.com | URL: www.shaunreed.com ##
## A custom bash script for creating new linux users. ##
## Syntax: ./adduser.sh <username> <userID> ##
#####
#####

if [ "$#" -ne 2 ]; then
    printf "Illegal number of parameters."
    printf "\nUsage: sudo ./adduser.sh <username> <groupid>"
    printf "\n\nAvailable groupd IDs:"
    printf "\n60001.....61183  Unused | 65520.....65533  Unused"
    printf "\n65536.....524287  Unused | 1879048191.....2147483647  Unused\n"
    exit
fi

sudo adduser $1 --gecos "First Last,RoomNumber,WorkPhone,HomePhone" --disabled-password --uid $2

printf "\nEnter 1 if $1 should have sudo privileges. Any other value will continue and make no changes\n"
read choice
if [ $choice -eq 1 ]; then
    printf "\nConfiguring sudo for $1...\n"
```

```
    sudo usermod -G sudo $1
fi

printf "\nEnter 1 to set a password for $1, any other value will exit with no password set\n"
read choice

if [ $choice -eq 1 ] ; then
    printf "\nChanging password for $1...\n"
    sudo passwd $1
fi
```