

Bash programming from scratch

```
$ wget https://ftp.gnu.org/gnu/bash/bash-4.4.tar.gz
--2019-12-27 10:45:56-- https://ftp.gnu.org/gnu/bash/bash-4.4.tar.gz
Resolving ftp.gnu.org (ftp.gnu.org)... 209.51.188.20, 2001:470:142:3::b
Connecting to ftp.gnu.org (ftp.gnu.org)|209.51.188.20|:443... connected.
HTTP request sent, awaiting response... 200 OK
Length: 9377313 (8.9M) [application/x-gzip]
Saving to: 'bash-4.4.tar.gz'

bash-4.4.tar.gz      100%[=====>]      8.94M  1.95MB/s   in 5.6s

2019-12-27 10:46:02 (1.61 MB/s) - 'bash-4.4.tar.gz' saved [9377313/9377313]

$ tar xvf bash-4.4.tar.g
```

Ilya Shpigor

Bash programming from scratch

Ilya Shpigor

This book is for sale at <http://leanpub.com/bash-programming-from-scratch>

This version was published on 2022-04-15



Leanpub

This is a [Leanpub](#) book. Leanpub empowers authors and publishers with the Lean Publishing process. [Lean Publishing](#) is the act of publishing an in-progress ebook using lightweight tools and many iterations to get reader feedback, pivot until you have the right book and build traction once you do.

© 2020 - 2022 Ilya Shpigor

Contents

Introduction	1
General Information	3
Operating Systems	3
Computer Program	29
Bash Shell	44
Development Tools	44
Command Interpreter	48
Navigating the File System	51
Command Information	80
Actions on Files and Directories	86
Extra Bash Features	99
Bash Scripts	124
Development Tools	124
Why Do We Need Scripts?	131
Variables and Parameters	147
Conditional Statements	176
Arithmetic Expressions	199
Loop Constructs	233
Functions	259
Package Manager	278
Repository	278
Package Operating	278
Finalwords	282
Acknowledgements	285
Glossary	286
A	286
B	287
C	288
E	289

CONTENTS

F	290
G	290
H	291
I	291
L	291
M	292
N	292
O	292
P	293
Q	294
R	294
S	294
T	296
U	296
V	296
W	296
Solutions for Exercises	298
General Information	298
Bash Shell	298
Bash Scrips	304
Helpful Links	322
General Information	322
Bash	322
Unix Environment	323

Introduction

Learning to program is not easy. It is even more challenging if you start from scratch and learn it on your own without mentors. This task is feasible, but its result depends on you.

You are going to study a new and complex subject. It will require strong motivation. Therefore, consider your goals before you continue reading this book. What do you expect from your new skills? What tasks will your programs solve? Answers to these questions help you to choose an effective way to study.

If you firmly intend to become a professional programmer in the shortest possible time, you need a mentor. Enroll yourself in a full-time or online course. It accelerates your progress significantly. You will get a chance to communicate directly with the mentor, ask him questions, and clarify unclear topics.

You can learn programming without a mentor if you do it for curiosity. Do you want to get a new hobby or learn a new fancy thing? In this case, self-study with the book is the right way. It will bring you practical benefits for relatively small efforts. After all, basic programming skills are helpful for anyone who works with the computer. Perhaps this book would be a starting point for you. It will help you to choose a direction for your further development.

Today there are plenty of books available in online shops and libraries. A reader with any technical level can find appropriate material for him. You can ask about the reason for writing this particular book.

Programming is a practical domain. Yes, it has a lot of theory. But mathematics has a lot of theory too. However, just knowing formulas does not make you a mathematician. In the same way, knowing the fundamental principles of software development does not make you a programmer.

You have to write a lot of code on your own to become a programmer. At first, this code won't work. Then, it will contain errors. Gradually, you will learn how to anticipate and fix the mistakes in advance. Knowledge of a particular language does not indicate your progress. But an evaluation of your old code does it. When you read your code several months later and notice its disadvantages, it confirms your progress.

So what is wrong with existing books? Many of them focus on a specific language or technology. They consider the chosen theme in detail. In this case, the author does not pay enough attention to practical exercises. A beginner programmer does not need such a volume of specialized knowledge. It brings him the wrong idea of how to learn to program. It is rarely that somebody reads such books for specific technologies from cover to cover. More often, they are used as a reference manual when the specific practical question arises.

Books of another type teach you some programming language by examples. When you study such a book, your progress goes faster. But there is one problem here. Many readers do not find the

motivation to work with examples. The author offers you to read and understand a lot of code. Often, this code demonstrates a specific principle and does not have a practical use case. Therefore, such examples are not attractive to readers.

Modern **general-purpose languages**¹ are complex. It means that you should know a lot about the language before dealing with real-life tasks. Here we meet a vicious circle. Examples are not interesting because they are useless. But useful real-life programs are too hard to understand for a beginner.

This book follows another approach. It starts with a general theory about a computer. Here we pay attention to the reasons for the technical solutions of the past. These solutions have defined the basic features of a modern computer. Following the reasons helps you to learn the material quickly.

The general theory about a computer helps you when you start programming in a particular language. You will certainly meet problems. For example, your program runs too slow or ends up with an error all the time. Knowing the internals of a computer helps you to understand the reasons for such behavior.

The next part of this book introduces you to the Bash programming language. Contrary to popular belief, it is a complex domain-specific language. However, there are some practical tasks that it solves easily and laconically. We will use them as examples. This way, we will learn the basic concepts of programming.

Our first step is replacing the graphical user interface with the shell. It will teach you basic operations on files and directories using Bash commands. When we get this basic syntax, we apply it and write our first Bash programs.

Please do not consider learning Bash as a necessary but useless exercise. Every professional programmer faces tasks of automating some processes and executing commands on a Unix-like system. Bash knowledge is irreplaceable in both cases.

If you are not able to complete some example or exercise, do not be upset. It means that the book does not disclose material sufficiently. Please **write**² to me about this. We will consider it out together.

There is a glossary at the end of the book. There you can clarify an unknown term that you met while reading the book.

¹https://en.wikipedia.org/wiki/General-purpose_language

²<mailto:petrsum@gmail.com>

General Information

This chapter explains the basics of how a computer works. It starts with describing operating systems and their history. Then the chapter considers families of modern operating systems and their features. The last section explains a computer program and its execution.

Operating Systems

History of OS Origin

Most computer users understand why an **operating system**³ (OS) is needed. Before launching a new **application**⁴, you usually check its system requirements. These requirements specify the **hardware**⁵ and OS that you need to launch the application.

The system requirements bring us the idea that the OS is a software platform. The application requires this platform for correct working. But where did this requirement come from? Why can't you just buy a computer and launch the application without any OS?

Our questions seem meaningless at first blush. But let's consider them from the following point of view. Modern operating systems are multipurpose, and they offer users many features. Each specific user would not use most of these features. However, you cannot disable them. The OS utilizes the computer's resources intensively for maintaining its features. As a result, the user has much fewer computation resources for his applications. This overhead leads to the computer's slow work, hanging of the programs, and even rebooting the whole system.

Let's turn to history to find out the origins of operating systems. The first commercial OS was called **GM-NAA I/O**⁶. It appeared for the computer **IBM 704**⁷ in 1956. All earlier computers have worked without any OS. Why didn't they need it?

The main reason for having an OS is high computational speed. For example, let's consider the first **electromechanical computer**⁸. **Herman Hollerith**⁹ constructed it in 1890. This computer was called a tabulator. It does not require an OS and a **program**¹⁰ in the modern sense. The tabulator performs a limited set of arithmetic operations only. Its hardware design defines this set of operations.

³https://en.wikipedia.org/wiki/Operating_system

⁴https://en.wikipedia.org/wiki/Application_software

⁵https://en.wikipedia.org/wiki/Computer_hardware

⁶https://en.wikipedia.org/wiki/GM-NAA_I/O

⁷https://en.wikipedia.org/wiki/IBM_704

⁸https://en.wikipedia.org/wiki/Tabulating_machine#1890_census

⁹https://en.wikipedia.org/wiki/Herman_Hollerith

¹⁰https://en.wikipedia.org/wiki/Computer_program

The tabulator loads input data for computation from **punched cards**¹¹. These cards look like sheets of thick paper with punched holes. A human operator prepares these sheets manually and stacks them in special receivers. The sheets are threaded on the needles in the receivers. Then a short circuit happens in each punched hole. Each short circuit increases the mechanical counter, which is a rotating cylinder. The tabulator displays calculation results on dials that resemble watches.

Figure 1-1 shows the tabulator that Hermann Hollerith has constructed.



Figure 1-1. Hollerith tabulating machine

By modern standards, the tabulator works very slowly. There are several reasons for this. The first one is you need a lot of manual work. At the beginning, you should prepare input data. There was no way to punch the cards automatically. Then you manually load punched cards into the receivers. Both operations require significant efforts and time.

The second reason for low computation speed is mechanical parts. The tabulator contains many of them: needles to read data, rotating cylinders as counters, dials to output the result. All these mechanics work slowly. It takes about one second to perform one elementary operation. No automation can accelerate these processes.

If you look at how the tabulator works, you find no place for OS there. OS just does not have any tasks to do on such a kind of computer.

Tabulators used rotating cylinders for performing calculations. The next generation of computers replaced the cylinders with **relays**¹². A relay is a electromechanical element that changes its state due to an electric current.

The German engineer **Konrad Zuse**¹³ designed one of **the first relay computers**¹⁴ called Z2 in 1939. Then he improved this computer in 1941 and called it Z3. These machines perform a single

¹¹https://en.wikipedia.org/wiki/Punched_card

¹²<https://en.wikipedia.org/wiki/Relay>

¹³https://en.wikipedia.org/wiki/Konrad_Zuse

¹⁴[https://en.wikipedia.org/wiki/Z2_\(computer\)](https://en.wikipedia.org/wiki/Z2_(computer))

elementary operation in milliseconds. They are much faster than tabulators. This performance gain happens thanks to applying relays instead of rotating cylinders.

The increased computation speed is the first feature of Zuse's computers. The second feature is the concept of a computer program. The idea of the universal machine, which follows the **algorithm**¹⁵ you choose, was fundamentally new at that time.



An algorithm is a finite sequence of instructions to perform a particular calculation or task.

Z3 computer uses two input devices in parallel for supporting programs. The first one is a receiver for punched cards that resembles the tabulator's receiver. It reads the program to execute from the card. The second input device is the keyboard. It allows the user to type input data for the program.

The computers with the feature of changing their algorithms became known as **programmable**¹⁶ or **general-purpose**.

The invention of programmable computers was a milestone in the development of computer science. Until this moment, machines were able to perform highly specialized tasks only. The construction of these machines was too expensive and unprofitable. This was a reason why commercial investors did not join the projects to design new computers. Only governments invested money there. However, this situation has changed since programmable computers came.

The next step in computer design is the construction of the **ENIAC**¹⁷ (see Figure 1-2) computer in 1946 by **John Eckert**¹⁸ and **John Mauchly**¹⁹. ENIAC uses a new type of element for performing computations. **Vacuum tubes**²⁰ replaced relays there. The tube is a purely electronic device. It does not have any mechanical component as the relay has. Therefore, when the electrical signal comes, the tube's reaction time is much faster than the relay one. This change increased ENIAC performance by order of magnitude comparing to relay-based machines. The new computer performs single elementary operation in 200 microseconds instead of milliseconds.

¹⁵<https://en.wikipedia.org/wiki/Algorithm>

¹⁶https://en.wikipedia.org/wiki/General_purpose_computer

¹⁷<https://en.wikipedia.org/wiki/ENIAC>

¹⁸https://en.wikipedia.org/wiki/J._Presper_Eckert

¹⁹https://en.wikipedia.org/wiki/John_Mauchly

²⁰https://en.wikipedia.org/wiki/Vacuum_tube

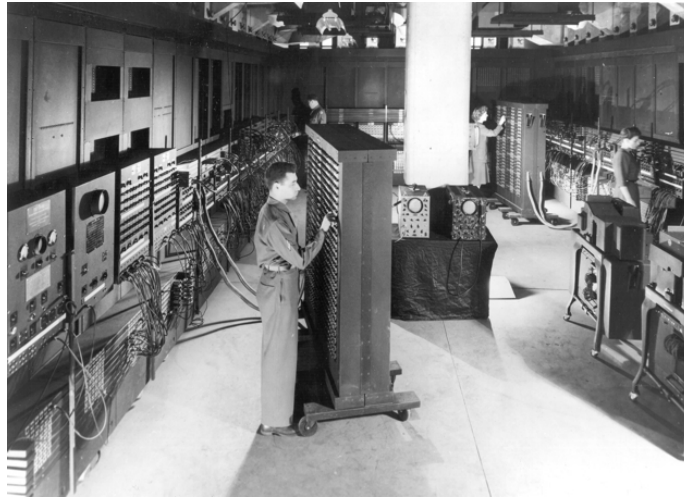


Figure 1-2. ENIAC

Most computer engineers were skeptical about vacuum tubes in the 1940s. The tubes were known for their low reliability and high power consumption. Nobody believed that a tube-based machine could work at all. ENIAC contains around 18,000 vacuum tubes. They burn out often. However, the computer performed the calculations efficiently between the failures. ENIAC was the first successful case of applying vacuum tubes. This case convinced many engineers that such an approach works well.

ENIAC is a programmable computer. You can set its algorithm using a combination of switches and jumpers on the control panels. This task requires considerable time and simultaneous work of several people. Figure 1-3 shows one of the panels for programming ENIAC.

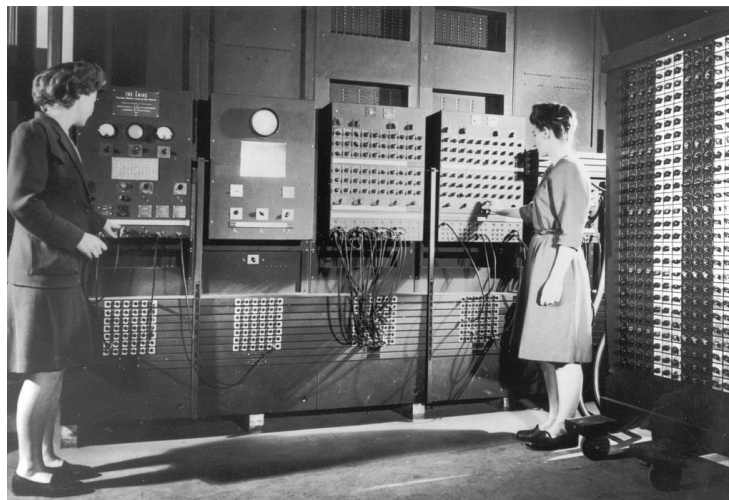


Figure 1-3. ENIAC control panel

ENIAC uses punched cards for reading input data and to output results. This is the same approach as previous models of computers have used. The new feature of ENIAC was storing intermediate results on the cards.

When the ENIAC should calculate a complex task, it can come to a leak of computation resources. However, you can split the task into several subtasks. Then the computer can solve each subtask separately. Storing the intermediate results on the cards helps you to move data from one subtask to another while you reprogram the computer.

Using ENIAC gave a new experience to engineers. It shows that all mechanical operations limit computer performance. These are the mechanical operations in ENIAC: manual reprogramming with switches and jumpers, reading and punching the cards. Each of these operations takes a significant time. Therefore, solving practical tasks with ENIAC is ineffective. The computer itself had an unprecedented performance at that time. However, it was idling most of the time and waiting for reprogramming or receiving input data. These findings initiated the development of new devices for both data input and output.

The next step in computer design is replacing vacuum tubes with [transistors](#)²¹. It again increased the computation speed by an order of magnitude. Transistors came together with new [input/output](#)²² devices. It allowed to increase the loading of the computers and reprogram them more often.

When the epoch of transistors started, computers spread beyond government and military projects. Banks and corporations began to exploit machines too. It increases the number and variety of programs running on computers significantly.

Commercial usage of computers brings new requirements. When the computer works in a company, it should execute programs one after another without any delays. Otherwise, the machine does not justify the money spent on it.

New solutions were needed to meet new requirements. The most time-consuming task was switching between programs. Therefore, engineers of General Motors and North American Aviation came to an idea to automate it. They created the first commercial operating system [GM-NAA I/O](#)²³. The primary goal of the system was managing the execution of programs.

The heavy load of computers and the variety of their programs brought another new task. When the computer of that time loads a program, the program defines the hardware's available features. For example, if the program includes a code to control an output device, you can use it. Otherwise, this device is unavailable.

Suppose that you are a company. You use a particular computer model all the time. The hardware always stays the same. Therefore, you do not change the code that controls your hardware. You just copy this code from one program to another. It takes extra efforts.

Copying hardware-specific code brought engineers to an idea of a special service program. The service program loads together with the user's application and provides the hardware support. These service programs became part of the first-generation operating systems after a while.

Now it is time to come back to the question, why do you need an OS. We found out that applications could work without them in general. Such applications are still in use today. For example, they are

²¹<https://en.wikipedia.org/wiki/Transistor>

²²<https://en.wikipedia.org/wiki/Input/output>

²³https://en.wikipedia.org/wiki/GM-NAA_I/O

utilities²⁴ for memory checks and disk partitioning, antiviruses, recovery tools. However, developing such applications takes more time and efforts. They should include the code for supporting all required hardware.

OS usually provides the code for supporting hardware. Why would you not use it? Most modern developers choose this way. It reduces the amount of work and speeds up the release of the applications.

However, a modern OS is a vast complex system. It provides much more features than just hardware support. Let's consider them too.

OS Features

Why did we start studying programming by considering the OS? OS features are the basis for the application. Let's consider how it works.

Figure 1-4 demonstrates the interaction between the OS, an application and hardware. Applications are programs that solve practical user tasks. Examples are text editor, calculator, browser. Hardware is all electronic and mechanical components of a computer. For example, these are keyboard, monitor, central processor, video card.

²⁴https://en.wikipedia.org/wiki/Utility_software

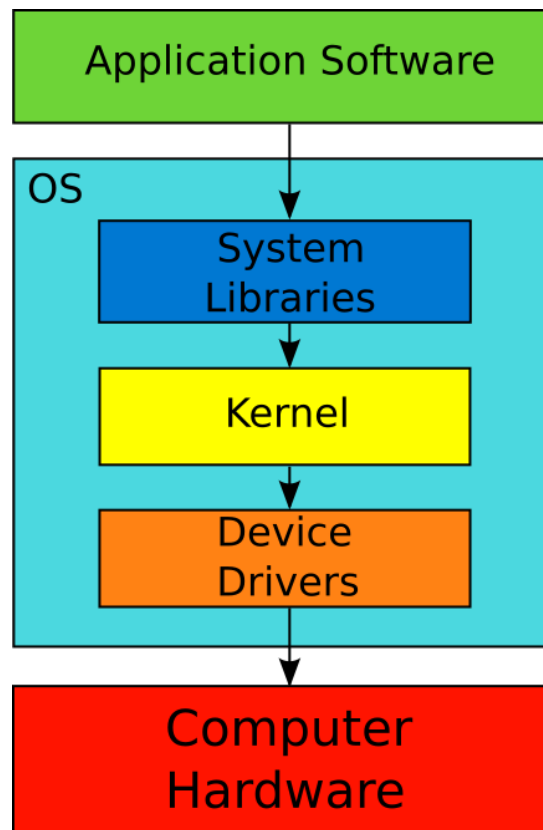


Figure 1-4. The interaction between the OS, an application and hardware

According to Figure 1-4, the application does not interact with hardware directly. The program does it through **system libraries**²⁵. The system libraries are part of the OS. There are rules to access the system libraries. Each application should follow them.

Application Programming Interface²⁶ or API is the interface that the OS provides to an application to interact with system libraries. In general, the API term means a set of agreements for interacting components of the information system. These agreements become a well-known standard often. For example, the POSIX standard describes the portable API for a family of OSes. The standard guarantees the compatibility of the OS and applications.

OS **kernel**²⁷ and **device drivers**²⁸ are part of OS. They dictate which hardware features the application can access. When the application interacts with system libraries, the libraries request capabilities of kernel and drivers. If you need the hardware feature and OS does not support it, you cannot use it.

When the application accesses the system library, it calls a library's **function**²⁹. A function is a program fragment or an independent block of code that performs a certain task. You can imagine

²⁵[https://en.wikipedia.org/wiki/Library_\(computing\)](https://en.wikipedia.org/wiki/Library_(computing))

²⁶<https://en.wikipedia.org/wiki/API>

²⁷[https://en.wikipedia.org/wiki/Kernel_\(operating_system\)](https://en.wikipedia.org/wiki/Kernel_(operating_system))

²⁸https://en.wikipedia.org/wiki/Device_driver

²⁹<https://en.wikipedia.org/wiki/Subroutine>

the API as a list of all available functions that the application can call. Besides that, API describes the following aspects of the interaction between the OS and applications:

1. What action does the OS perform when the application calls the specific system function?
2. What data does the function receive as input?
3. What data does the function return as a result?

Both the OS and application should follow the API agreements. It guarantees the compatibility of their current versions and future modifications. Such compatibility is impossible without a well-documented and standardized interface.

We have discovered that some applications work without an OS. They are called **bare-metal software**³⁰. This approach works well in some cases. However, the OS provides ready-made solutions for interaction with the computer's hardware. Without these solutions, developers should take responsibility for managing hardware. It requires significant efforts. Imagine the variety of devices of a modern computer. The application should support all popular models of each device (for example, video cards). Without such support, the application would not work for all users.

Let's consider the features that the OS provides via the API. We can treat the computer's hardware as resources. The application uses these resources for performing calculations. The API reflects the list of hardware features that the program can use. Also, the API dictates the order of interaction between several applications and the hardware.

There is an example. Two programs cannot write data to the same area of the **hard disk**³¹ simultaneously. There are two reasons for that:

1. A single magnetic head records data on the hard disk. The head can do one task at a time.
2. One program can overwrite data of another program in the same memory area. It leads to losing data.

You should place all requests to write data on the disk in a queue because of these two problems. Then each request should be performed separately. The OS takes care of this task.

The kernel (see Figure 1-4) of the OS provides a mechanism for managing access to the hard drive. This mechanism is called **file system**³². Similarly, the OS manages access to all **peripheral**³³ and internal devices of the computer. Device drivers provide access to the hardware for OS and applications.

We have mentioned peripheral and internal devices. What is the difference between them? Peripherals are all devices that are responsible for inputting, outputting, and storing data permanently. Here are the examples:

³⁰<https://www.quora.com/What-is-bare-metal-programming-in-Embedded-systems>

³¹https://en.wikipedia.org/wiki/Hard_disk_drive#Magnetic_recording

³²https://en.wikipedia.org/wiki/File_system

³³<https://en.wikipedia.org/wiki/Peripheral>

- Keyboard
- Mouse
- Microphone
- Monitor
- Speakers
- Hard drive

Internal devices are responsible for processing data, i.e. for executing programs. These are internal devices:

- **Central Processing Unit**³⁴ (CPU)
- **Random-Access Memory**³⁵ (RAM)
- **Video Card**³⁶ (graphics processing unit or GPU).

The OS provides access to the computer's hardware. At the same time, the OS has something besides the hardware management to share with user's applications. The system libraries have grown from the program modules to serve the devices. However, some libraries of modern OSes provide complex algorithms for processing data. Let's consider an example.

There is the Windows OS component called **Graphics Device Interface**³⁷ (GDI). It provides algorithms for manipulating graphic objects. GDI allows you to create a user interface for your application with minimal efforts. Then you can use the monitor to display this interface.

The system libraries with useful algorithms (like GDI) are software resources of the OS. These resources are already installed on your computer. You just need to know how to use them. Besides that, the OS also provides access to third-party libraries and their algorithms. You can install these libraries separately and use them in your applications.

The OS manages hardware and software resources. Also, it organizes the joint work of running programs. The OS performs several non-trivial tasks to launch an application. Then the OS tracks its work. If the application violates some agreements (like memory usage), the OS terminates it. We will consider launching and executing the program in detail in the next section.

If the OS is multi-user, it controls access to the data. It is an important security feature. This feature allows the user to access the following file system objects:

- Files and directories that the user owns.
- Files and directories that somebody has shared with the user.

It allows several persons to use the same computer safely.

Here is the summary. The modern OS has all following features:

³⁴https://en.wikipedia.org/wiki/Central_processing_unit

³⁵https://en.wikipedia.org/wiki/Random-access_memory

³⁶https://en.wikipedia.org/wiki/Video_card

³⁷https://en.wikipedia.org/wiki/Graphics_Device_Interface

1. It provides and manages access to hardware resources of the computer.
2. It provides its own software resources.
3. It launches applications.
4. It organizes the interaction of applications with each other.
5. It controls access to users' data.

You can guess that it is impossible to launch several applications in parallel without the OS. You are right. When you develop an application, you have no idea how a user will launch it. The user can launch your application together with another one. You cannot foresee this use case. However, the OS responds for launching all applications. It means that the OS has enough information to allocate computer resources properly.

Modern OSes

We have reviewed the OS features in general. Now we will consider modern operating systems. Today you can pick any OS and get very similar features. However, their developers follow different approaches. This leads to implementation difference that can be important for some users.

There is the **software architecture**³⁸ term. It means the implementation aspects of the specific software and the solutions that led to them.

All modern OSes have two features that determine the way how users interact with them. These features are multitasking and the graphical user interface. Let's take a closer look at them.

Multitasking

All modern OSes support **multitasking**³⁹. It means that they can execute multiple programs in parallel. The systems with this feature displaced OSes without it. Why does multitasking so important?

The challenge of efficient usage of computers came in the 1960s. Computers were expensive at that time. Only large companies and universities were able to buy them. These organizations counted every minute of working with their machines. They did not accept any idle time of the hardware because of its huge cost.

Early operating systems executed programs one after another without delays. This approach saves time for switching computer tasks. If you use such an OS, you should prepare several programs and their input data in advance. Then you should write them on the storage device (e.g., magnetic tape). You load the tape to the computer's reading device. Afterward, the computer executes the programs sequentially and prints their results to an output device (e.g., a printer). This mode of operation is called **batch processing**⁴⁰.

³⁸https://en.wikipedia.org/wiki/Software_architecture

³⁹https://en.wikipedia.org/wiki/Computer_multitasking

⁴⁰https://en.wikipedia.org/wiki/Batch_processing

Batch processing increased the efficiency of using computers in the 1960s. This approach has automated program loading. The human operators became unnecessary for this task. However, the computers still had the **bottleneck**⁴¹. The computational power of processors was increasing significantly every year. The speed of peripherals has remained almost the same. It led to CPU idles all the time while waiting for input/output.



A bottleneck is a component or resource of an information system that limits its overall performance or bandwidth.

Why does CPU idle and wait for peripheral devices? Here is an example. Suppose that the computer from the 1960s runs programs one by one. It reads data from a magnetic tape and prints the results on the printer. The OS loads each program and executes its instructions. Then it loads the next one, and so on.

The problem happens when reading data and printing the results. The time for reading data on the magnetic tape is huge on the CPU scale. This time is enough for the processor to perform many calculations. However, it does not do that. The reason for that is the currently loaded program occupies all computer resources. The same CPU idle happens when printing the results. The printer is a slow electromechanical device.

The CPU idle led OS developers to the concept of **multiprogramming**⁴². This concept implies that OS loads several programs into the computer memory at the same time. The first program works as long as all resources it needs are available. It stops executing once a required resource is busy. Then OS switches to another program.

Here is an example. Suppose that your application wants to read data from your hard disk. While the disk controller reads the data, it is busy. It cannot process the following requests from the program. Thus, the application waits for the controller. In this case, OS stops executing it and switches to the second program. The computer executes it to the end or until it has all the required resources. When the second program finishes or stops, OS switches tasks again.

Multiprogramming differs from multitasking that modern OSes have. Multiprogramming fits the batch processing mode very well. However, this load balancing concept is not suitable for **interactive systems**⁴³. An interactive system considers each user action as an event (for example, a keystroke). The system should process events immediately when they happen. Otherwise, the user cannot work with the system.

Here is an example of workflow with an interactive system. Suppose that you are typing text in the MS Office document. You press a key and expect to see this symbol on the screen. If the computer requires several seconds to process your keystroke and display the symbol, you cannot work like that. Most of the time, you will wait and check if the computer has processed your keystroke or not. This is inefficient.

⁴¹[https://en.wikipedia.org/wiki/Bottleneck_\(production\)](https://en.wikipedia.org/wiki/Bottleneck_(production))

⁴²https://en.wikipedia.org/wiki/Computer_multitasking#Multiprogramming

⁴³https://en.wikipedia.org/wiki/Interactivity#Computing_science

Multiprogramming cannot handle events in the interactive system. It happens because you cannot predict when task switching happens next time. OS switches tasks when a program finishes or when it requires a busy resource. Suppose that your text editor is not an active program now. Then you do not know when it can process your keystroke. It can happen in a second or in several minutes. This is unacceptable for a good user interface.

The multitasking concept solves the task of processing events in interactive systems. There were several versions of multitasking before it comes to the current state. Modern OSes use **displacing multitasking**⁴⁴ with pseudo-parallel tasks processing. The idea behind it is to allow the OS to choose an appropriate task for executing at the moment. The OS takes into account the priorities of the running programs. Therefore, a higher priority program receives hardware resources more often than a lower priority one. OS kernel provides this task-switching mechanism. It is called **task scheduler**⁴⁵.

Pseudo-parallel processing means that the computer executes one task only at any given time. The OS switches tasks so quickly that you can suppose the processing of several programs at once. This concept allows the OS to react immediately to any event. Even though every program and OS component uses hardware resources at strictly defined moments.

There are computers with multiple processors or with **multi-core processors**⁴⁶. Only these computers can execute several programs at once. The number of the running programs equals the number of cores of all processors approximately. The preemptive multitasking mechanism with constant task switching works on such systems well. It is a universal approach that balances the load regardless of the number of cores. This way, the computer responds to the user's actions quickly. The number of processor cores does not matter.

User Interface

Modern OSes are able to solve a wide range of tasks. These tasks depend on the computer type where you run the OS. Here are the main types of computers:

- **Personal computers**⁴⁷ (PC) and notebooks.
- Smartphones and tablets.
- Servers.
- **Embedded systems**⁴⁸.

We will consider OSes for PCs and notebooks only. Apart from multitasking, they provide **graphic user interface**⁴⁹ (GUI). This interface means the way to interact with the system. You launch applications, configure computer devices and OS components via the user interface. Let's take a look at its history and find how it has reached the current state.

⁴⁴[https://en.wikipedia.org/wiki/Preemption_\(computing\)](https://en.wikipedia.org/wiki/Preemption_(computing))

⁴⁵[https://en.wikipedia.org/wiki/Scheduling_\(computing\)](https://en.wikipedia.org/wiki/Scheduling_(computing))

⁴⁶https://en.wikipedia.org/wiki/Multi-core_processor

⁴⁷https://en.wikipedia.org/wiki/Personal_computer

⁴⁸https://en.wikipedia.org/wiki/Embedded_system

⁴⁹https://en.wikipedia.org/wiki/Graphical_user_interface

Nobody works with commercial computers interactively before 1960. [Digital Equipment Corporation](#)⁵⁰ implemented the interactive mode for their [minicomputer](#)⁵¹ PDP-1⁵² in 1959. It was a fundamentally new approach. Before that, IBM computers dominated the market in the 1950s. They worked in batch processing mode only. This mode automated program loading and provided high performance for calculation tasks.

The idea of interactive work with the computer appeared first in the SAGE military project. The US Air Force was its customer. The goal of the project was to develop an automated air defense system to detect Soviet bombers.

When working on the SAGE project, engineers faced the problem. The user of the system should analyze data from radars in real-time. If he detects a threat, he should react as quickly as possible and command to intercept the bombers. However, the existed methods of interaction with the computer did not fit this task. They did not allow showing information to the user in real-time and receive his input at any moment.

Engineers came to the idea of the interactive mode. They implemented it in the first interactive computer [AN/FSQ-7](#)⁵³ in 1955 (see Figure 1-5). The computer used the monitor with a [cathode-ray tube](#)⁵⁴ to display information from radars. The [light pen](#)⁵⁵ allowed the user to command the system.

⁵⁰https://en.wikipedia.org/wiki/Digital_Equipment_Corporation

⁵¹<https://en.wikipedia.org/wiki/Minicomputer>

⁵²<https://en.wikipedia.org/wiki/PDP-1>

⁵³https://en.wikipedia.org/wiki/AN/FSQ-7_Combat_Direction_Central

⁵⁴https://en.wikipedia.org/wiki/Cathode-ray_tube

⁵⁵https://en.wikipedia.org/wiki/Light_pen

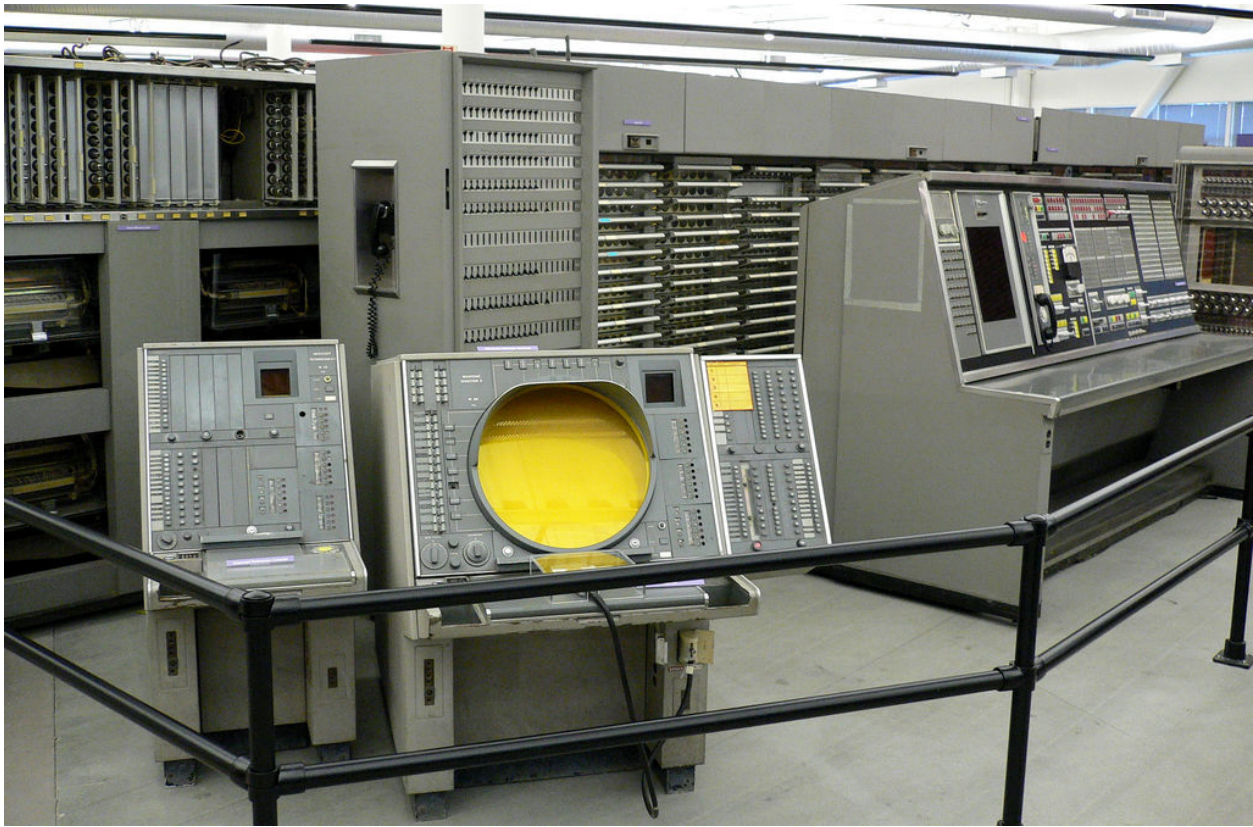


Figure 1-5. Computer AN/FSQ-7

The new way of interaction with computers became known in scientific circles. It gained popularity quickly. The existing batch processing mode coped with program execution well. However, this mode was inconvenient for development and **debugging**⁵⁶ applications.



Debugging a program is searching and eliminating its errors.

Suppose that you are writing a program for the computer with batch processing. You should prepare your code and write it to the storage device. When the device is ready, you put it in a queue. The computer's operator takes devices from the queue and loads them to the computer one by one. Therefore, your task can wait in the queue for several hours. Now assume that an error happened in your program. In this case, you should fix it, prepare the new version of the program and write it to the device. You put it in the queue again and wait for hours. Because of this workflow, you need several days to make even a small program working.

The software development workflow differs when you use an interactive mode. You prepare your program and load it to the computer. Then it launches the program and shows you results. Immediately, you can analyze a possible error, fix it and relaunch the program. This workflow

⁵⁶<https://en.wikipedia.org/wiki/Debugging>

accelerates software development and debugging tasks significantly. Now you spend few hours on the task that requires days with batch processing mode.

The interactive mode brought new challenges for computer engineers. This mode requires a system that reacts to user actions immediately. A short reaction time required a new load-balancing mechanism. The multitasking concept became a solution for this issue.

There are alternative solutions for providing interactive mode. For example, there are interactive single-tasking OSes like [MS-DOS](#)⁵⁷. MS-DOS was the system for cheap PCs of the 1980s.

However, it was inadvisable to apply single-tasking in the 1960s when computers were too expensive. These computers executed many programs in parallel. Such a mode was called [time-sharing](#)⁵⁸. It allows sharing expensive hardware resources among several users. The single-tasking approach does not fit such a use case because it is not compatible with time-sharing.

When the first relatively cheap personal computers appeared in the 1980s, they used single-tasking OSes. Such systems require fewer hardware resources than their analogs with multitasking. Despite its simplicity, single-tasking OSes support interactive mode for the running program. This mode became especially attractive for PC users.

When interactive mode became more and more popular, computer engineers meet a new challenge. The existing devices for interacting with computers turned out to be inconvenient. Magnetic tapes and printers were widespread through the 1950s and early 1960s. They did not fit interactive mode absolutely.

[Teletype](#)⁵⁹ (TTY) became a prototype of a device for interactive work with a computer. The original idea behind this device was to connect two of them via wires. It allows users on both sides to send each other text messages. One user types the message on the keyboard. Then his device transmits the keystrokes to the receiver. When the teletype on the other side receives data, it prints the text on paper.

Figure 1-6 shows the Model 33 teletype. It was one of the most popular devices in the 1960s.

⁵⁷<https://en.wikipedia.org/wiki/MS-DOS>

⁵⁸<https://en.wikipedia.org/wiki/Time-sharing>

⁵⁹<https://en.wikipedia.org/wiki/Teleprinter>



Figure 1-6. A Teletype Model 33

Computer engineers connected the teletype to the computer. This solution allowed the user to send text commands to the machine and receive results. Such a workflow became known as a **command-line interface**⁶⁰ (CLI).

Teletype uses the printer as an output device. It works very slow and requires around 10 seconds to print a single line. The next step of developing the user interface was replacing the printer with the monitor. This increased the data output speed several times. The new device with a keyboard and monitor was called the **terminal**⁶¹. It replaced teletypes in the 1970s.

Figure 1-7 shows a modern command-line interface. You can see the **terminal emulator**⁶² application there. This application emulates the terminal device for the sake of compatibility. It allows you to run programs that work with the real terminal only. The emulator application in Figure 1-7 is called **Terminator**⁶³. The Bash command-line interpreter is running in the Terminator window. The window displays the results of the ping and ls programs.

⁶⁰https://en.wikipedia.org/wiki/Command-line_interface

⁶¹https://en.wikipedia.org/wiki/Computer_terminal

⁶²https://en.wikipedia.org/wiki/Terminal_emulator

⁶³[https://en.wikipedia.org/wiki/Terminator_\(terminal_emulator\)](https://en.wikipedia.org/wiki/Terminator_(terminal_emulator))

```

root@hamamelis: ~
root@hamamelis: ~ 94x30
root@hamamelis:~# ping google.com
PING google.com (172.217.16.206) 56(84) bytes of data:
64 bytes from fra16s08-in-f14.1e100.net (172.217.16.206): icmp_seq=1 ttl=53 time=22.9 ms
64 bytes from fra16s08-in-f14.1e100.net (172.217.16.206): icmp_seq=2 ttl=53 time=23.1 ms
^C
--- google.com ping statistics ---
2 packets transmitted, 2 received, 0% packet loss, time 1001ms
rtt min/avg/max/mdev = 22.948/23.070/23.192/0.122 ms
root@hamamelis:~# ls -la
total 64
drwx----- 9 root root 4096 Sep 28 11:20 .
drwxr-xr-x 24 root root 4096 Oct 29 2017 ..
-rw----- 1 root root 4197 Jul 25 22:34 .bash_history
-rw-r--r-- 1 root root 3106 Oct 22 2015 .bashrc
drwx----- 7 root root 4096 May 7 21:29 .cache
drwx----- 6 root root 4096 May 7 21:29 .config
drwx----- 3 root root 4096 Jun 5 2017 .dbus
drwx----- 2 root root 4096 Nov 2 10:20 .gconf
drwxr-xr-x 4 root root 4096 Sep 28 11:19 .java
drwx----- 3 root root 4096 Mar 19 2017 .local
drwxr-xr-x 4 root root 4096 Sep 28 11:20 .pdfstudio2019
-rw-r--r-- 1 root root 148 Aug 17 2015 .profile
-rw-r--r-- 1 root root 75 Mar 19 2017 .selected_editor
-rw----- 1 root root 6713 Jul 25 2018 .viminfo
root@hamamelis:~#

```

Figure 1-7. Command-line interface

The command-line interface is still in demand today. It has several advantages over the graphical interface. For example, CLI does not require as many resources as GUI. CLI runs reliably on low-performance embedded computers as well as on powerful servers. If you use CLI to access the computer remotely, you can use a low bandwidth communication channel. The server will receive your commands in this case.

The command-line interface has some disadvantages. Learning to use CLI effectively is hard and takes time. You have to remember around a hundred commands. Each command has several modes that change its behavior. Therefore, you should keep in mind these modes too. It takes some time to remember at least a minimum set of commands for daily work.

There is an option to make the command-line interface more user-friendly. You can give a hint to the user about available commands. It was done in the **text-based interface**⁶⁴ (TUI). The interface uses **box-drawing characters**⁶⁵ along with alphanumeric symbols. The box-drawing characters display the graphic primitives on the screen. Primitives are lines, rectangles, triangles, etc. They guide the user about the available actions he can do with the application.

Figure 1-8 shows a typical text-based interface. There is an output of system resource usage by the `htop` program.

⁶⁴https://en.wikipedia.org/wiki/Text-based_user_interface

⁶⁵https://en.wikipedia.org/wiki/Box-drawing_character

```

root@hamamelis: ~
root@hamamelis: ~ 129x35

 1  [|||||] 11.4% 5 [|||||] 6.5%
 2  [|||||] 9.3% 6 [|||||] 5.9%
 3  [|||||] 14.3% 7 [|||||] 6.5%
 4  [|||||] 14.3% 8 [|||||] 2.7%
Mem [|||||] 2.75G/15.5G Tasks: 135, 592 thr; 5 running
Swp [|||||] 109M/7.45G Load average: 0.93 0.97 1.76
Uptime: 10:36:22

  PID USER  PRI  NI  VIRT  RES  SHR  S  CPU%  MEM%  TIME+  Command
 3258 elly   20   0 9350M 510M 181M R 55.9 3.2 1:24.51 /usr/lib/firefox/firefox -contentproc -childID 14 -isForBrowser -p
32296 elly   20   0 3427M 562M 210M S 12.5 3.6 2:58.67 /usr/lib/firefox/firefox
 2908 elly   20   0 471M 10460 7632 R 5.3 0.1 7:36.28 /usr/bin/pulseaudio --start --log-target=syslog
 3364 elly   20   0 9350M 510M 181M S 4.6 3.2 0:03.14 /usr/lib/firefox/firefox -contentproc -childID 14 -isForBrowser -p
 3365 elly   20   0 9350M 510M 181M S 4.6 3.2 0:03.05 /usr/lib/firefox/firefox -contentproc -childID 14 -isForBrowser -p
 3361 elly   20   0 9350M 510M 181M S 4.6 3.2 0:03.10 /usr/lib/firefox/firefox -contentproc -childID 14 -isForBrowser -p
 3362 elly   20   0 9350M 510M 181M S 4.6 3.2 0:03.01 /usr/lib/firefox/firefox -contentproc -childID 14 -isForBrowser -p
 3350 elly   21   1 9350M 510M 181M R 3.9 3.2 0:03.64 /usr/lib/firefox/firefox -contentproc -childID 14 -isForBrowser -p
 3377 elly   20   0 9350M 510M 181M S 3.9 3.2 0:02.50 /usr/lib/firefox/firefox -contentproc -childID 14 -isForBrowser -p
 2991 elly   20   0 709M 5496 4492 S 2.6 0.0 9:37.15 conky -q
 2930 elly   20   0 471M 10460 7632 R 2.6 0.1 3:19.73 /usr/bin/pulseaudio --start --log-target=syslog
32317 elly   20   0 3427M 562M 210M S 2.6 3.6 0:02.95 /usr/lib/firefox/firefox
 2112 root    20   0 288M 88896 70272 S 2.0 0.5 1h01:25 /usr/lib/xorg/Xorg -core :0 -seat seat0 -auth /var/run/lightdm/root
32316 elly   20   0 3427M 562M 210M S 2.0 3.6 0:03.16 /usr/lib/firefox/firefox
 3500 root    20   0 25292 4300 3228 R 2.0 0.0 0:00.28 htop
 3378 elly   20   0 9350M 510M 181M S 2.0 3.2 0:01.10 /usr/lib/firefox/firefox -contentproc -childID 14 -isForBrowser -p
26698 elly   20   0 841M 34484 25284 S 2.0 0.2 1:14.95 pavucontrol
32399 elly   20   0 20.6G 269M 98208 S 2.0 1.7 0:41.83 /usr/lib/firefox/firefox -contentproc -childID 2 -isForBrowser -pr
 3372 elly   20   0 9350M 510M 181M S 1.3 3.2 0:00.83 /usr/lib/firefox/firefox -contentproc -childID 14 -isForBrowser -p
 3367 elly   20   0 9350M 510M 181M S 1.3 3.2 0:01.15 /usr/lib/firefox/firefox -contentproc -childID 14 -isForBrowser -p
 2932 elly   20   0 471M 10460 7632 S 1.3 0.1 1:10.69 /usr/bin/pulseaudio --start --log-target=syslog
 3363 elly   20   0 9350M 510M 181M S 1.3 3.2 0:01.20 /usr/lib/firefox/firefox -contentproc -childID 14 -isForBrowser -p
 3268 elly   20   0 9350M 510M 181M S 1.3 3.2 0:00.40 /usr/lib/firefox/firefox -contentproc -childID 14 -isForBrowser -p
 3007 elly   20   0 709M 5496 4492 S 0.7 0.0 4:06.44 conky -q

F1 Help F2 Setup F3 Search F4 Filter F5 Tree F6 SortBy F7 Nice F8 Nice F9 Kill F10 Quit

```

Figure 1-8. Text-based user interface

The further performance gain of computers allowed OS developers to replace box-drawing characters with real graphic elements. There are examples of such elements: windows, icons, buttons, etc. It was a moment when the full-fledged graphical interface came. Modern OSes provide this kind of interface.

The first GUI appeared in the [Xerox Alto](https://en.wikipedia.org/wiki/Xerox_Alto)⁶⁶ minicomputer (see Figure 1-10). It was developed in the research center [Xerox PARC](https://en.wikipedia.org/wiki/PARC_(company))⁶⁷ in 1973. However, the graphical interface did not spread widely until the 1980s. It happens because GUI requires a lot of memory and high computer performance. PCs with such features were too expensive for ordinary users at that time.

Apple produced the first relatively cheap PC with GUI in 1983. It was called Lisa.

⁶⁶https://en.wikipedia.org/wiki/Xerox_Alto

⁶⁷[https://en.wikipedia.org/wiki/PARC_\(company\)](https://en.wikipedia.org/wiki/PARC_(company))



Figure 1-10. Minicomputer Xerox Alto

Figure 1-9 demonstrates the Windows GUI. You can see the desktop. There are windows of three applications there. The applications are Explorer, Notepad and Calculator. They work in parallel.

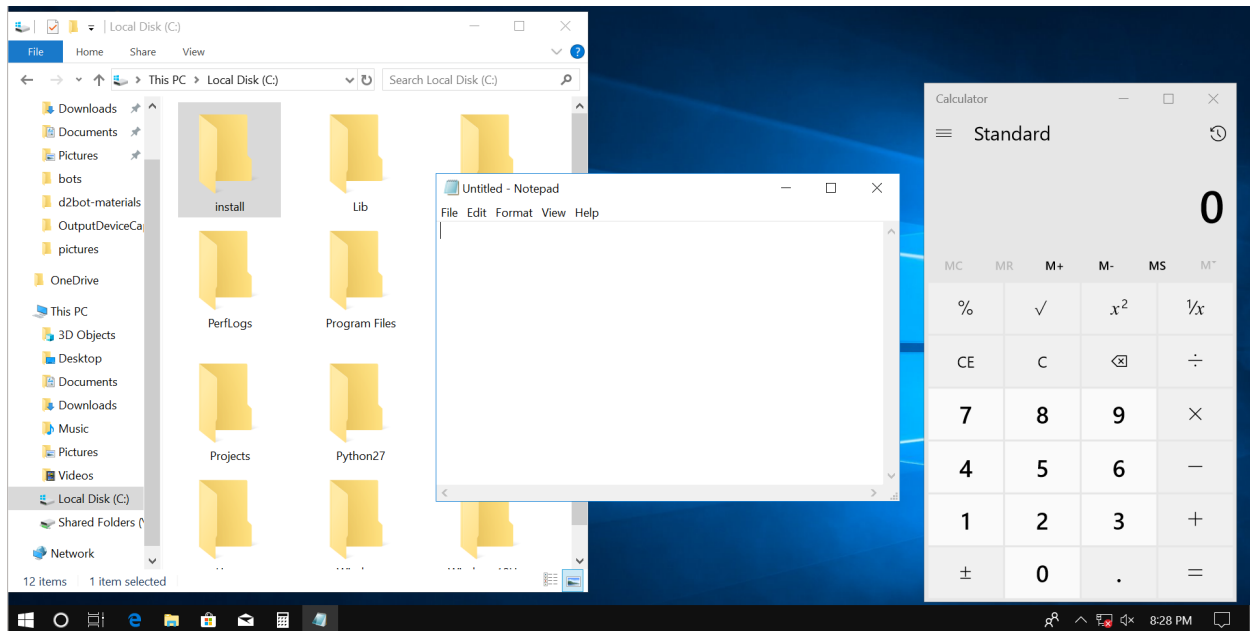


Figure 1-9. Windows GUI

Families of OSes

There are three [families of OSes](#)⁶⁸ that dominate the market today. Here are these families:

- [Windows](#)⁶⁹
- [Linux](#)⁷⁰
- [macOS](#)⁷¹

The term “family” means several OS versions that follow the same architectural solutions. Therefore, most functions in these versions are implemented in the same way.

The developers of the OS family follow the same architecture. They do not offer something fundamentally new in the upcoming versions of their product. Why?

Actually, changes in modern OSes happen gradually and slowly. The reason for this is a **backward compatibility**⁷² problem. This compatibility means that newer OS versions provide the features of older versions. Most existing programs require these features for their work. You can suppose that backward compatibility is an optional requirement. However, it is a severe limitation for software development. Let’s find out why it is.

Imagine that you wrote a program for Windows and sell it. Sometimes users meet errors in the program. You receive bug reports and fix them. Also, you add new features from time to time.

⁶⁸https://en.wikipedia.org/wiki/Category:Operating_system_families

⁶⁹https://en.wikipedia.org/wiki/Microsoft_Windows

⁷⁰<https://en.wikipedia.org/wiki/Linux>

⁷¹<https://en.wikipedia.org/wiki/MacOS>

⁷²https://en.wikipedia.org/wiki/Backward_compatibility

Your business goes well until the new Windows version comes. Let's assume that Microsoft has changed its architecture completely. Therefore, your program does not work on the new OS version. This leads users of your program to the following choice:

- Update Windows and wait for the new version of your program that works there.
- Do not update Windows and continue to use your program.

If users need your program for daily work, they refuse the Windows update. Using the program is more important than getting new OS features.

We know that Microsoft has changed the Windows architecture completely. It means that you should rewrite your program from scratch. Now count all the time that you have spent fixing bugs and adding new features. You should repeat all this work as soon as possible. Most likely, you will give up this idea. Then your users should stay with the old Windows version.

Windows is a very popular and widespread OS. It means that there are many programs like yours. Their developers will come to the same decision as you. As a result, nobody updates to the new Windows version. This situation demonstrates the backward compatibility problem. This problem forces OS developers to be careful with changing their products. The best solution for them is to make a family of similar OSes.

There is a significant influence of user applications on OS development. For example, Windows and IBM computers owe their success to a table processor [Lotus 1-2-3](https://en.wikipedia.org/wiki/Lotus_1-2-3)⁷³. You need both IBM PC and Windows to launch Lotus 1-2-3. For the sake of Lotus 1-2-3, users bought both the computer and OS. The specific combination of the hardware and software is called the [computing platform](https://en.wikipedia.org/wiki/Computing_platform)⁷⁴. The popular application, which brings the platform to the broad market, is called [killer application](https://en.wikipedia.org/wiki/Killer_application)⁷⁵.

The tabular processor [VisiCalc](https://en.wikipedia.org/wiki/VisiCalc)⁷⁶ was another killer application. It promoted the distribution of the [Apple II](https://en.wikipedia.org/wiki/Apple_II_series)⁷⁷ computers. In the same way, free compilers for C, Fortran and Pascal languages help Unix OS to become popular in university circles.

There was the killer application behind each of the modern OS families. This application gave these OSes the initial success. Further distribution of the OS happened thanks to the [network effect](https://en.wikipedia.org/wiki/Network_effect)⁷⁸. This effect means that developers tend to choose the most widespread computing platforms for their new applications.

What are the differences between the OS families? Windows and Linux are remarkable because they do not depend on the hardware. It means that you can install them on any modern PC or laptop. macOS runs on Apple computers only. If you want to use macOS on different hardware, you would need the unofficial [modified version](https://en.wikipedia.org/wiki/Hackintosh)⁷⁹ of OS.

⁷³https://en.wikipedia.org/wiki/Lotus_1-2-3

⁷⁴https://en.wikipedia.org/wiki/Computing_platform

⁷⁵https://en.wikipedia.org/wiki/Killer_application

⁷⁶<https://en.wikipedia.org/wiki/VisiCalc>

⁷⁷https://en.wikipedia.org/wiki/Apple_II_series

⁷⁸https://en.wikipedia.org/wiki/Network_effect

⁷⁹<https://en.wikipedia.org/wiki/Hackintosh>

Hardware compatibility is an example of the design decision of the OS development. There are many such decisions. Together they define the features and design of each OS family.

There is one more important point for software development besides the OS design. OS dictates the infrastructure for the programmer. The infrastructure means development tools. Examples of these tools are IDE, compiler, build system. Tools together with OS API impose some design decisions for the applications. It leads to a specific culture for program development. Please keep in mind that you should develop applications differently for each OS. Take it into account when you design your programs.

Let's consider the origins of software development cultures for Windows and Linux.

Windows

Windows is [proprietary software](#)⁸⁰. The source code of such software is unavailable for users. You cannot read or modify it as you want. In other words, there is no legal way to know about proprietary software more than its documentation tells you.

If you want to install Windows on your computer, you should buy it from Microsoft. However, manufacturers of computers pre-install Windows on their devices often. In this case, the final cost of the computer includes the price of the OS.

The target devices for Windows are relatively cheap PCs and laptops. Many people can buy such a device. Therefore, there is a huge market of potential users. Microsoft tends to keep its competitive edge in this market. The best way to reach it is to prevent appearing of Windows analogs with the same features from other companies. For reaching this goal, Microsoft takes care of protecting its intellectual property. The company does it in both technical and legal ways. An example of legal protection is the user agreement. It prohibits you to explore the internals of the OS.

The Windows OS family has a long history. Also, it is popular and widespread. It leads many developers to choose this OS as the target for their applications. However, the Microsoft company has developed the first Windows applications on its own. An example is the package of office programs [Microsoft Office](#)⁸¹. Such applications became a standard to follow for other developers.

Microsoft followed the same principle when developing both Windows and applications for it. It is a secrecy principle:

- Source codes are not available to users.
- Data formats are undocumented.
- Third-party utilities do not have access to software features.

The goal of these decisions is to protect intellectual property.

Other software developers have followed the example of Microsoft. They stuck with the same philosophy of secrecy. As a result, their applications are self-contained and independent of each other. The formats of their data are encoded and undocumented.

⁸⁰https://en.wikipedia.org/wiki/Proprietary_software

⁸¹https://en.wikipedia.org/wiki/Microsoft_Office

If you are an experienced computer user, you immediately recognize a typical Windows application. It has a window with **control elements**⁸² like buttons, input fields, tabs, etc. You manipulate some data using these control elements. Examples of data are text, image, sound record, video. When you are done, you save your results on the hard disk. You can open it again in the same application. If you write your own Windows program, it will look and work similarly. This succession of solutions is called the development culture.

Linux

Linux has borrowed most of the ideas and solutions from the **Unix**⁸³. Both OSes follow the set of standards that is called **POSIX**⁸⁴ (Portable Operating System Interface). POSIX defines interfaces between applications and the OS. Linux and Unix got the same design because they follow the same standard. We should have a look at the Unix origins to get this design.

The Unix appeared in the late 1960s. Two engineers from the Bell Labs company have developed it. Unix was a hobby project of **Ken Thompson**⁸⁵ and **Dennis Ritchie**⁸⁶. In their daily work, they developed the **Multics**⁸⁷ OS. It was a joint project of the Massachusetts Institute of Technology (MIT), General Electric (GE) and Bell Labs. General Electric planned to use Multics for its new computer GE-645. Figure 1-11 demonstrates this machine.

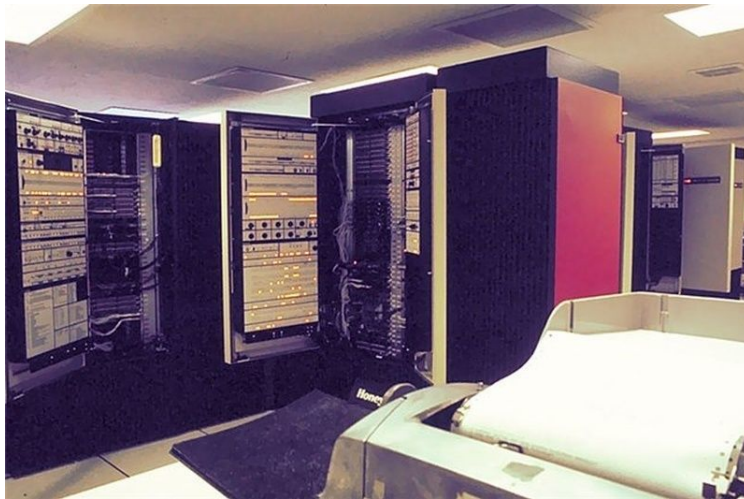


Figure 1-11. Mainframe GE-645

The Multics developers have invented several innovative solutions. One of them was time-sharing. It allows several users to work with the computer at the same time. Multics uses the multitasking concept to share resources among all users.

Because of many innovations and high requirements, Multics turned out to be too complicated. The

⁸²https://en.wikipedia.org/wiki/Graphical_widget

⁸³<https://en.wikipedia.org/wiki/Unix>

⁸⁴<https://en.wikipedia.org/wiki/POSIX>

⁸⁵https://en.wikipedia.org/wiki/Ken_Thompson

⁸⁶https://en.wikipedia.org/wiki/Dennis_Ritchie

⁸⁷<https://en.wikipedia.org/wiki/Multics>

project consumed more time and money than it was planned. This was a reason why Bell Labs left the project.

The Multics project was interesting from a technical point of view. Therefore, many Bell Labs engineers wanted to continue working on it. Ken Thompson was one of them. He decided to create his own operating system for the computer GE-645. Thompson started to write the system kernel and duplicated some Multics mechanisms. However, General Electric demanded the return of its GE-645 soon. Bell Labs has received this computer on loan only. As a result, Thompson lost a hardware platform for his development.

When working on the Multics analog, Thompson had a pet project to create a computer game. It was called [Space Travel](#)⁸⁸. He launched the game on the past generation computer GE-635 from General Electric. It had the [GECOS](#)⁸⁹ OS. GE-635 consisted of several modules. Each module was a cabinet with electronics. The overall computer cost was about \$7500000. Bell Labs engineers actively used this machine. Therefore, Thompson was rarely able to work with it to develop his game.

The limited access to the GE-635 machine was a problem. Therefore, Thompson decided to port his game to a relatively inexpensive and rarely used computer [PDP-7](#)⁹⁰ (see Figure 1-12). Its cost was about \$72000. When doing that, Thompson met one problem. Space Travel used the features of the GECOS OS. The software of PDP-7 did not provide such features. Thompson was joined by his colleague Dennis Ritchie. They implemented GECOS features for PDP-7 together. It was a set of libraries and subsystems. Over time, these modules were developed into a self-sufficient OS. It was called Unix.



Figure 1-12. Minicomputer PDP-7

Thompson and Ritchie were not going to sell Unix. Therefore, they never had a goal to protect their intellectual property. They developed Unix for their own needs. Afterward, they distributed it for free with the source code. Everyone could copy and use this OS. It was reasonable because the first Unix users were Bell Labs employees only.

⁸⁸[https://en.wikipedia.org/wiki/Space_Travel_\(video_game\)](https://en.wikipedia.org/wiki/Space_Travel_(video_game))

⁸⁹https://en.wikipedia.org/wiki/General_Comprehensive_Operating_System

⁹⁰<https://en.wikipedia.org/wiki/PDP-7>

Unix became popular among Bell Labs employees. Thompson and Ritchie presented the OS at the Symposium on Operating Systems Principles conference. Then they got a lot of requests for the system. However, Bell Labs belonged to AT&T company. Therefore, Bell Labs did not have the right to distribute any software on its own.

AT&T noticed the new perspective OS. The company started to sell its source code to universities for \$20000. Thus, university circles got a chance to improve and develop Unix.

[Linus Torvalds](#)⁹¹ met Unix when he had studied at the University of Helsinki. Unix encouraged him to create his own OS called Linux. It was not a pet project for fun. Torvalds met a practical problem. He needed a Unix-compatible OS for his PC to do university tasks at home. Such OS was not available at that moment.

At the University of Helsinki, students performed study assignments using the MicroVAX computer running Unix. Many of them had PCs at home. However, there was no Unix version for PC. The only Unix alternative for students was [Minix](#)⁹² OS.

Andrew Tanenbaum developed Minix for IBM PCs with Intel 80268 processors in 1987. He created Minix for educational purposes only. This was a reason why he refused to apply changes to his OS for supporting modern PCs. Tanenbaum was afraid that his system becomes too complicated. Then he cannot use it for teaching students.

Torvalds had a goal to write a Unix-compatible OS for his new IBM computer with Intel 80386 processor. He used Minix OS for development but did not import any part of it to Linux. Like the Unix creators, Torvalds had no commercial interests and was not going to sell his software. He developed the OS for his own needs and wanted to share it with everyone. Linux became free in this way. Torvalds decided to distribute it with source code for free via the Internet. This decision made Linux well known and popular.

Torvalds developed the kernel of OS only. The kernel provides memory management, file system, peripherals drivers and processor time scheduler. However, users needed an interface to access the kernel's features. It means that the Linux OS was not ready for use as it is.

The solution to the problem came from the [GNU software project](#)⁹³. [Richard Stallman](#)⁹⁴ started this project at MIT in 1983. His idea was to develop the most necessary software for computers and make it free. The major products of the GNU project are the GCC compiler, glibc system library, system utilities and Bash shell. Torvalds included these products in his project and released the first [Linux distribution](#)⁹⁵ in 1991.

The first versions of Linux did not have a graphical interface. The only way to interact with the system was a command-line shell. Some complex applications had a text interface, but they were the minority. Linux got a GUI in the middle of the 1990s. This interface was based on [X Window System](#)⁹⁶ free software. X Window allowed developers to create graphical applications.

⁹¹https://en.wikipedia.org/wiki/Linus_Torvalds

⁹²<https://en.wikipedia.org/wiki/MINIX>

⁹³https://en.wikipedia.org/wiki/GNU_Project

⁹⁴https://en.wikipedia.org/wiki/Richard_Stallman

⁹⁵https://en.wikipedia.org/wiki/Linux_distribution

⁹⁶https://en.wikipedia.org/wiki/X_Window_System

Unix and Linux evolved in very specific conditions. They differ from a typical cycle of commercial software development. These conditions made a unique development culture. Both systems developed in university circles. Computer science teachers and students used the OSes in daily work. They understood this software well. Therefore, they fixed software errors and added new features there willingly.

Let's have a look at what is the Unix development culture. Unix users prefer to use highly specialized command-line utilities. You can find a tool almost for each specific task. Such tools are well written, tested many times and worked as efficiently as possible. However, all features of one utility are focused on one specific task. The utility is not universal software to cover most of your needs.

When you meet a complex task, there is no single utility to solve it. However, you can easily combine several utilities and solve your task this way. Such an interaction becomes available thanks to a clear data format. Most Unix utilities use the [text data](#)⁹⁷ format. It is simple and self-explained. You can start working with it immediately.

The Linux development culture follows the Unix traditions. It differs from the standards that are adopted in Windows. Every application is monolithic and performs all its tasks by itself in the Windows world. The application does not rely on third-party utilities. The reason is the most software for Windows costs money and can be unavailable to the user. Therefore, each developer relies on himself. He cannot force the user to buy something extra to make the specific application working.

The software dependency looks different in Linux. Most of the utilities are free, interchangeable and accessible via the Internet. Therefore, it is natural that one program requires you to download and install a missing system component or another program.

The interaction of programs is crucial in Linux. Even monolithic graphical Linux applications usually provide a command-line interface. This way, they fit smoothly into the ecosystem. It leads that you can integrate them with other utilities and applications.

Suppose that you are solving a complex task in Linux. You should assemble a single computing process from a combination of highly specialized utilities. It means that you make a computation algorithm that can be complex by itself. Linux provides a tool for this specific task. The tool is called [shell](#)⁹⁸. Using the shell, you type commands and the system performs them. The first Unix shell appeared in 1979. It was called [Bourne shell](#)⁹⁹. Now it is deprecated. The [Bash](#)¹⁰⁰ shell has replaced it in most Linux distributions. We will consider Bash in this book.

We have considered Linux and Windows cultures. You cannot give a preference to one or another. Comparing them causes endless disputes on the Internet. Each culture has its advantages and disadvantages. For example, the Window-style monolithic applications cope well the tasks that require intensive calculations. When you combine specialized Linux utilities for the same task, you get an overhead. The overhead happens because of launching many utilities and transferring data between them. This requires extra time. As a result, you wait longer to complete your task.

⁹⁷https://en.wikipedia.org/wiki/Plain_text

⁹⁸https://en.wikipedia.org/wiki/Unix_shell

⁹⁹https://en.wikipedia.org/wiki/Bourne_shell

¹⁰⁰[https://en.wikipedia.org/wiki/Bash_\(Unix_shell\)](https://en.wikipedia.org/wiki/Bash_(Unix_shell))

Today, we observe a synthesis of Windows and Linux cultures. Microsoft started to contribute to open-source projects: Linux kernel, the [samba](#)¹⁰¹ network protocol, [PyTorch](#)¹⁰² machine learning library, the [Chromium](#)¹⁰³ web browser, etc. The company has released some of its projects with a free license: [.NET Framework](#)¹⁰⁴, PowerShell, VS Code IDE, etc.

On the other side, more and more commercial applications get ported to Linux: browsers, development tools, games, messengers, etc. However, their developers are not ready for changes that the Linux culture dictates. Such changes require too much time and effort. They also make it more challenging to maintain the product. Instead of one application, there are two: each platform has a different version. Therefore, developers port their applications without significant changes. As a result, you find more and more Windows-style applications on Linux.

One can argue about the pros and cons of this process. However, the more applications run on the specific OS, the more popular it becomes, thanks to the network effect.



Read more about the development culture in Unix and Linux in Eric Raymond's book "[The Art of Programming in Unix](#)"¹⁰⁵.

Computer Program

We got acquainted with operating systems. They are responsible for starting and running computer programs. The program or application solves the user's specific task. For example, a text editor allows you to write and edit documents.

A program is a set of elementary steps. They are called instructions. The computer performs these steps sequentially. It follows the strict order of actions and copes with complex tasks. Let's consider how the computer launches and executes the program in detail.

Computer Memory

A hard disk stores all instructions of the program. If the program is relatively small and simple it fits a single file. Complex applications occupy several files.

Suppose that you have a single file program. When you launch it, the OS loads the file into the computer memory called RAM. Then the OS allocates a part of processor time for the new task. This way, the processor performs the program's instructions at specified intervals.

The first step of launching a program is to load it into RAM. We should consider the computer memory internals to understand this step better.

¹⁰¹[https://en.wikipedia.org/wiki/Samba_\(software\)](https://en.wikipedia.org/wiki/Samba_(software))

¹⁰²<https://en.wikipedia.org/wiki/PyTorch>

¹⁰³[https://en.wikipedia.org/wiki/Chromium_\(web_browser\)](https://en.wikipedia.org/wiki/Chromium_(web_browser))

¹⁰⁴https://en.wikipedia.org/wiki/.NET_Framework

¹⁰⁵https://en.wikipedia.org/wiki/Unix_philosophy#Eric_Raymond's_17_Unix_Rules

The single unit of the computer memory is **byte**¹⁰⁶. The byte is the minimum amount of information that the processor can reference and load into its memory. However, the CPU can handle smaller amounts of data if you apply special techniques. You operate bits in this case. A **bit**¹⁰⁷ is the smallest amount of information you cannot divide. You can imagine the bit as a single logical state. It has one of two possible values. There are several ways to interpret them:

- 0 or 1
- True or False
- Yes or No
- + or —
- On or Off.

Another way to imagine one bit is to compare it with a lamp switch. It has two possible states:

- The switch closes the circuit. Then the lamp is on.
- The switch opens the circuit. Then the lamp is off.

A byte is a memory block of eight bits. Here you can ask why do we need this packaging? CPU can operate a single bit, right? Then it should be able to refer to a specific bit in memory.

CPU cannot refer to a single bit. There are technical reasons for that. The primary task of the first computers was arithmetic calculations. For example, these computers calculated the **ballistic tables**¹⁰⁸. You should operate integers and fractional numbers to solve such a task. The computer does the same. However, a single bit is not enough to store a number in memory. Therefore, the computer needs memory blocks to store numbers. The bytes became such blocks.

Introducing bytes affected the architecture of processors. Engineers have expected that the CPU performs most operations over numbers. Therefore, they added a feature to load and process all bits of the number at once. This solution increased computers' performance by order of magnitude. At the same time, loading of the single bit in the CPU happens rarely. Supporting this feature in hardware brings significant overhead. Therefore, engineers have excluded it from modern processors.

There is one more question. Why does a byte consist of eight bits? It was not always this way. The byte was equal to **six bits**¹⁰⁹ in the first computers. Such a memory block was enough to encode all the English alphabet characters in upper case, numbers, punctuation marks and mathematical operations.

Six-bits encodings were insufficient for representing control and box-drawing characters. Therefore, these encodings were extended to seven bits in the early 1960s. The **ASCII encoding**¹¹⁰ appeared at that moment. It became the standard for encoding characters. ASCII defines characters for codes from 0 to 127. The maximum seven-bit number 127 limits this range.

¹⁰⁶<https://en.wikipedia.org/wiki/Byte>

¹⁰⁷<https://en.wikipedia.org/wiki/Bit>

¹⁰⁸https://www.wikiwand.com/en/Ballistic_table

¹⁰⁹https://en.wikipedia.org/wiki/Six-bit_character_code

¹¹⁰<https://en.wikipedia.org/wiki/ASCII>

Then IBM released the computer [IBM System/360](#)¹¹¹ in 1964. The size of a byte was eight bits in this computer. IBM chose this size for supporting old character encodings from the past projects. The IBM System/360 computer was popular and widespread. It led that eight-bit packaging became the industry standard.

Table 1-1 shows frequently used [units of information](#)¹¹² besides bits and bytes.

Table 1-1. Units of information

Title	Abbreviation ¹¹³	Number of bytes	Number of bits
kilobyte	KB	1000	8000
megabyte	MB	1000000	8000000
gigabyte	GB	1000000000	8000000000
terabyte	TB	10000000000	8000000000000

Table 1-2 shows standard storage devices and their capacity. You can compare them using Table 1-1.

Table 1-2. Storage devices

Storage device	Capacity
Floppy disk 3.5 ¹¹⁴	1.44 MB
Compact disk ¹¹⁵	700 MB
DVD ¹¹⁶	up to 17 GB
USB flash drive ¹¹⁷	up to 2 TB
Hard disk drive ¹¹⁸	up to 16 TB
Solid State Drive ¹¹⁹	up to 100 TB

We got acquainted with units of information. Now let's get back to the execution of the program. Why does the OS load it into RAM? In theory, the processor can read the program instructions directly from the hard disk drive, right?

A modern computer has four levels of the [memory hierarchy](#)¹²⁰. Each level matches the red rectangle in Figure 1-13. Each rectangle match a separate device. The only exception is the CPU chip. It contains both registers and a memory cache. These are separate modules of the chip.

You see the arrows in Figure 1-13. They represent data flows. Data transfer occurs between adjacent memory levels.

Suppose that you want to process some data on the CPU. Then you should load these data to its registers. This is the only place where the processor can take data for calculations. If the CPU needs something from the disk drive, the following data transfers happen:

¹¹¹https://en.wikipedia.org/wiki/IBM_System/360

¹¹²https://en.wikipedia.org/wiki/Units_of_information

¹¹⁴https://en.wikipedia.org/wiki/Floppy_disk

¹¹⁵https://en.wikipedia.org/wiki/Compact_disc

¹¹⁶<https://en.wikipedia.org/wiki/DVD>

¹¹⁷https://en.wikipedia.org/wiki/USB_flash_drive

¹¹⁸https://en.wikipedia.org/wiki/Hard_disk_drive

¹¹⁹https://en.wikipedia.org/wiki/Solid-state_drive

¹²⁰https://en.wikipedia.org/wiki/Memory_hierarchy

1. Disk drive -> RAM
2. RAM -> Processor cache
3. Processor cache -> Processor registers

When the CPU writes data back to the disk, it happens in the reverse order of steps.

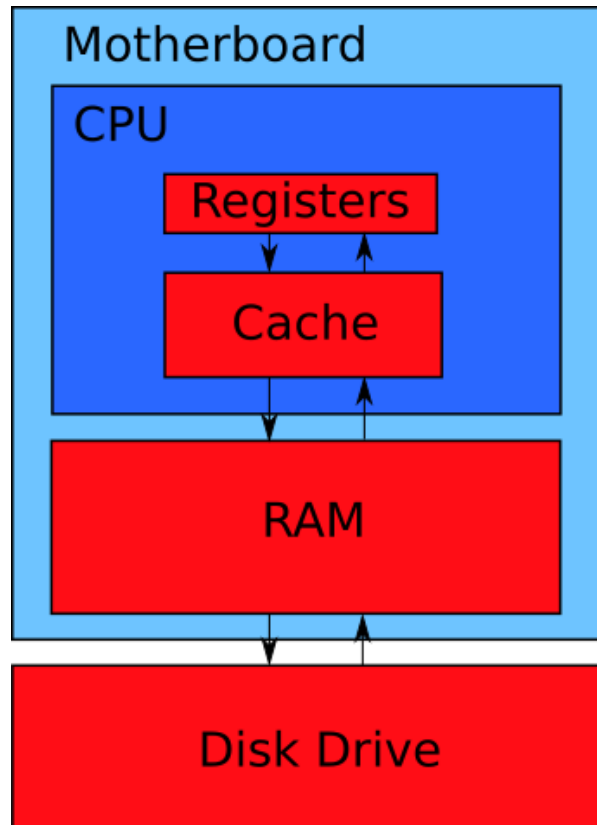


Figure 1-13. Memory hierarchy

Data storage devices have the following parameters:

1. **Access speed** defines the amount of data that you can read or write to the device per unit of time. Units of measure are bytes per second (Bps).
2. **Capacity** is the maximum amount of data that a device can store. The units are bytes.
3. **Cost** is a price of a device concerning its capacity. The units are dollars or cents per byte or bit.
4. **Access time** is the time between the moment when the processor needs some data from the device and when it receives them. Units are **clock signals**¹²¹ of the CPU.

These parameters vary for devices on each level of the memory hierarchy. Table 1-3 shows the ratio of parameters for modern storage devices.

¹²¹https://en.wikipedia.org/wiki/Clock_signal

Table 1-3. Memory levels

Level	Device	Capacity	Access speed	Access time	Cost
1	CPU registers ¹²²	up to 1000 bytes	-	1 tick	-
2	CPU cache ¹²³	from one KB to several MB	from 700 to 100 GBps	from 2 to 100 cycles	-
3	RAM	dozens of GB	10 GBps	up to 1000 clock cycles	\$10 ⁻⁹ /byte
4	Disk drive (hard drive ¹²⁴ or solid drive ¹²⁵)	several TB	2000 Mbps	up to 10000000 cycles	\$10 ⁻¹² /byte

Table 1-3 raises questions. You can read the data from the disk drive at high speed. Why is there no way to read these data to the CPU registers directly? Actually, you can do it, but it leads to a significant performance drawback.

The high speed of accessing a memory storage is not so crucial for performance in practice. The critical parameter here is the access time. It measures the idle time of the CPU until it receives the required data. You can measure this idle time in clock signals or cycles. Such a signal synchronizes all operations of the processor. The CPU requires roughly from 1 to 10 clock cycles to execute one instruction of the program.

High access time can cause serious performance issues. For example, suppose that the CPU reads the program instructions directly from the hard disk. The problem happens because CPU registers have a small capacity. There is no chance to load the whole program from the hard disk to the registers. Therefore, when the CPU did one part of the program, it should load the next one. This loading operation takes up to 10000000 clock cycles. It means that loading data from the disk takes a much longer time than processing them. The CPU spends most of the time idling. The memory hierarchy solves exactly this problem.

Let's consider data flow between memory levels by example. Suppose that you launch a simple program. It reads a file from the hard disk and displays its contents on the screen. Reading data from the disk happens in several steps. The hardware does them.

The first step is reading data from the hard disk into the RAM according to Figure 1-13. The next step is loading data from RAM to the CPU cache. There is a sophisticated **caching mechanism**. It guesses the data from RAM that the CPU requires next. This mechanism reduces the access time to the data and decreases the idle time of the CPU.

When data comes to the CPU chip, it manages them on its own. The processor reads the required data from the cache to registers and manipulates them. The program instructions reach the CPU the

¹²²https://en.wikipedia.org/wiki/Processor_register

¹²³https://en.wikipedia.org/wiki/CPU_cache

¹²⁴https://en.wikipedia.org/wiki/Hard_disk_drive

¹²⁵https://en.wikipedia.org/wiki/Solid-state_drive

same way as the data.

The program displays data on the screen in our example. It should call the corresponding API function for that. Then the system library changes the screen picture. The CPU does the actual work here. It loads the instructions of the system library and the video card driver. Then the CPU applies these instructions to the data in its registers. This way, the video card driver displays the data on the screen.

The required data may be absent in the specific memory level. Here are few examples. Suppose that the CPU needs data to process them in the video driver code. If these data are in the CPU cache but not in the registers, the processor waits for 2-100 clock cycles to get them. If data are in the RAM, the CPU's waiting time increases by order of magnitude up to 1000 cycles.

Our program can display both small and big files. Some big file does not fit the RAM. Then the RAM contains only part of it. The CPU can require the missing file part. In this case, the CPU idle time increases by four orders of magnitude up to 10000000 clock cycles. For comparison, the processor could execute about 1000000 program instructions instead of this idle time. This is really a lot.

Both CPU and disk drives use hardware caching mechanisms. The idea of [caching for disk drives](#)¹²⁶ is to store some data in the small and fast access memory. It speeds up reading and writing blocks of data. There are caching mechanisms on the software level too. They are parts of the OS in most cases.

All caching mechanisms increase a computer's performance significantly. When such a mechanism makes a mistake, it leads to the CPU idle. This mistake is called **cache miss**. Any cache miss is expensive from the performance point of view. Therefore, remember the memory hierarchy and caching mechanisms. Consider them when developing algorithms. Some algorithms and data structures cause more cache misses than others.

The storage devices with lower access times are placed closer to the CPU. Figure 1-14 demonstrates this principle. For example, registers and cache is the internal CPU memory. They are part of the processor's chip.

Both CPU and RAM are two chips that are plugged into the [motherboard](#)¹²⁷ near each other. The high-frequency [data bus](#)¹²⁸ connects them. This data bus provides low access time.

The motherboard is the [printed circuit board](#)¹²⁹ that connects computer components. You can plug in some chips right into the motherboard. However, there are devices that you should connect via cables there. The disk drive is an example of such a device. It connects to the motherboard via a relatively slow interface. There are several standards for such an interface: ATA, SATA, SCSI, PCI Express.

The old motherboard models have an embed chip that transfers data between CPU and RAM. This chip is called [northbridge](#)¹³⁰. Thanks to improving technologies for chip manufacturing, the CPUs take northbridge's functions since 2011.

¹²⁶https://en.wikipedia.org/wiki/Disk_buffer

¹²⁷<https://en.wikipedia.org/wiki/Motherboard>

¹²⁸[https://en.wikipedia.org/wiki/Bus_\(computing\)](https://en.wikipedia.org/wiki/Bus_(computing))

¹²⁹https://en.wikipedia.org/wiki/Printed_circuit_board

¹³⁰[https://en.wikipedia.org/wiki/Northbridge_\(computing\)](https://en.wikipedia.org/wiki/Northbridge_(computing))

The **southbridge**¹³¹ is another motherboard's chip. It presents there nowadays. The southbridge transfers data between RAM and devices that are connected via slow interfaces like PCI, USB, SATA, etc.

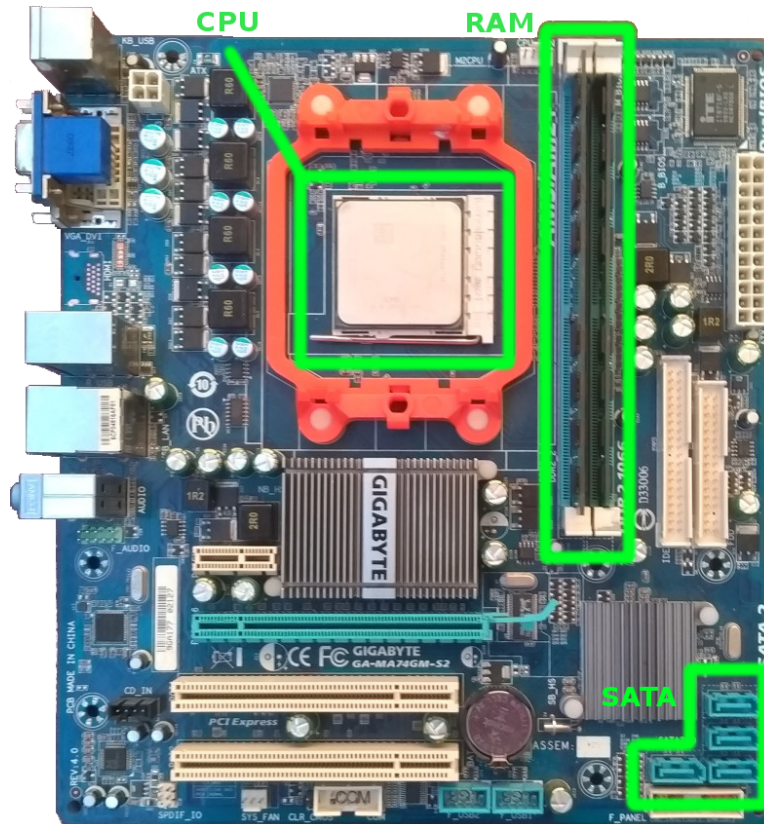


Figure 1-14. PC motherboard

Machine code

Suppose that the OS has loaded the contents of an executable file into RAM. This file contains both instructions and data of the program. Examples of data are text strings, box-drawing characters, predefined constants, etc.

Program instructions are written in **machine code**¹³². This code consists of commands that processor performs one by one. A single instruction is an elementary operation on the data from the CPU registers.

The CPU has logical blocks for executing each type of instruction. The available blocks determine the operations that the CPU can perform. If the processor does not have an appropriate logical block to accomplish a specific instruction, it combines several blocks to do this job. The execution takes more clock cycles in this case.

¹³¹[https://en.wikipedia.org/wiki/Southbridge_\(computing\)](https://en.wikipedia.org/wiki/Southbridge_(computing))

¹³²https://en.wikipedia.org/wiki/Machine_code

When the OS has loaded the program instructions and its data into RAM, it allocates the CPU time slots for that. The program becomes a **computing process**¹³³ or process since this moment. The process means the running program and the resources it uses. Examples of the resources are memory area and OS objects.

How do the program instructions look like? You can see them using the special program for reading and editing executable files. Such a program is called **hex editor**¹³⁴. The editor represents the program's machine instructions in **hexadecimal numeral system**¹³⁵. The actual contents of the executable file is **binary code**¹³⁶. This code is a sequence of zeros and ones. The CPU receives program instructions and data in this format. The hex editor makes them easy to read for humans.

There are advanced programs to read machine code. They are called **disassemblers**¹³⁷. Disassembler translates the program instructions into **assembly language**¹³⁸. This language is more convenient for a human to read. You can get a better representation of the program using the disassembler than the hex editor.

If you take a specific number, it looks different in various numeral systems. The numeral system determines the symbols and their order to write a number. For example, binary allows 0 and 1 symbols only.

Table 1-4 shows matching of numbers in binary (BIN), decimal (DEC), and hexadecimal (HEX).

Table 1-4. Numbers in decimal, hexadecimal and binary

Decimal	Hexadecimal	Binary
0	0	0000
1	1	0001
2	2	0010
3	3	0011
4	4	0100
5	5	0101
6	6	0110
7	7	0111
8	8	1000
9	9	1001
10	A	1010
11	B	1011
12	C	1100
13	D	1101
14	E	1110
15	F	1111

¹³³[https://en.wikipedia.org/wiki/Process_\(computing\)](https://en.wikipedia.org/wiki/Process_(computing))

¹³⁴https://en.wikipedia.org/wiki/Hex_editor

¹³⁵<https://en.wikipedia.org/wiki/Hexadecimal>

¹³⁶https://en.wikipedia.org/wiki/Binary_code

¹³⁷<https://en.wikipedia.org/wiki/Disassembler>

¹³⁸https://en.wikipedia.org/wiki/Assembly_language



Most calculator applications can convert numeral systems. If you use Windows, convert numbers with the standard Windows calculator in the **programmer mode**¹³⁹.

Why do programmers use both binary and hexadecimal numeral systems? It is more convenient to use only one of them, right? We should know more about computer hardware to answer this question.

The binary numeral system and **Boolean algebra**¹⁴⁰ are the basis of **digital electronics**¹⁴¹. An electrical **signal**¹⁴² is the smallest portion of the information there. When you work with such signals, you need a way to encode them. Encoding means associating specific numbers with the signal states. The signal has two states only. It can be present or absent. Therefore, the simplest way to encode the signal is to take the first two integers: zero and one. Then you use zero when there is no signal. Otherwise, you use the number one. Such encoding is very compact. You can use one bit to encode one signal.

The basic element of digital electronics is a **logic gate**¹⁴³. It converts electrical signals. You can implement the logic gate using various physical devices. Examples of such devices are an electromagnetic relay, vacuum tube and transistor. Each device has its own physics in the background. However, they work in the same way in terms of signal processing. This processing contains two steps:

1. Receive one or more signals on the input.
2. Transmit the resulting signal to the output.

The signal processing follows the rules of Boolean algebra. This algebra has an elementary operation for each type of logic gate. When you combine logic gates together, you can calculate the result using a Boolean expression. Combining logic gates provides a complex behavior. You need it if your digital device follows some non-trivial logic. For example, the CPU is a huge network of logic gates.

When you deal with digital electronics, you should apply the binary numeral system. This way, you convert signal states to logical values that Boolean algebra operates. Then you can calculate the resulting signals. The level of signals and logic gates is close to the computer hardware. You have to work on this level sometimes when programming. We can say that the hardware design dictates you to use the binary numeral system.

The binary numeral system is a language of hardware. Why do you need the hexadecimal system then? When you develop a program, you need decimal and binary systems only. Decimal is convenient for writing a general logic of the program. For example, you count repeating the same action in decimal.

You need the binary system when your program deals with computer hardware. For example, you send data to some device in binary format. There is one problem with binary. It is hard to write,

¹³⁹https://en.wikipedia.org/wiki/Windows_Calculator#Features

¹⁴⁰https://en.wikipedia.org/wiki/Boolean_algebra

¹⁴¹https://en.wikipedia.org/wiki/Digital_electronics

¹⁴²https://en.wikipedia.org/wiki/Signal#Digital_signal

¹⁴³https://en.wikipedia.org/wiki/Logic_gate

read, memorize and pronounce the numbers in this system for humans. Conversion from decimal to binary takes effort. The hexadecimal system solves both problems of representing and converting numbers. It is a compact and convenient as the decimal system. At the same time, you can convert a number from hexadecimal to binary form easily.

Follow these steps to convert a number from binary to hexadecimal form:

1. Split the number into groups of four digits. Start the splitting from the end of the number.
2. If the last group is less than four digits, add zeros to the left side of the group.
3. Use Table 1-4 to replace each four-digit group with one hexadecimal number.

Here is an example of converting the binary number 110010011010111:

1 110010011010111 = 110 0100 1101 0111 = 0110 0100 1101 0111 = 6 4 D 7 = 64D7

Exercise 1-1. Numbers conversion from binary to hexadecimal

Convert the following numbers from binary to hexadecimal:

- * 10100110100110
 - * 1011000111010100010011
 - * 111110111000100101010011000000110101101
-

Exercise 1-2. Numbers conversion from hexadecimal to binary

Convert the following numbers from hexadecimal to binary:

- * FF00AB02
 - * 7854AC1
 - * 1E5340ACB38
-

There are the answers for all exercises in the last section of the book. Check yourself there if you are unsure about your results.

Let's come back to executing the program. The OS loads its executable file from the disk drive into RAM. Then the OS loads all libraries that the program requires. The special OS component does both these tasks. It is called **loader**¹⁴⁴. Thanks to preloading libraries, the CPU does not idle too much when the program accesses them. The instructions of the required library are already in RAM. Therefore, the CPU waits for a few hundred clock cycles to access them. When the loader finishes his job, the program becomes a process. The CPU executes it, starting from the first instruction.

Each machine code instruction is called **opcode**¹⁴⁵. The opcode dictates the CPU which logic gates it should apply for data in the specific registers. When the operation is done, the opcode specifies the register for storing the result. Opcodes have a binary format that is a natural language of the CPU.

While the program is running, its instructions, resources and required libraries occupy the RAM area. The OS clears this memory area when the program finishes. Then other applications can use it.

¹⁴⁴[https://en.wikipedia.org/wiki/Loader_\(computing\)](https://en.wikipedia.org/wiki/Loader_(computing))

¹⁴⁵<https://en.wikipedia.org/wiki/Opcode>

Source Code

Machine code is a low-level language for the representation of a program. This format is convenient for the processor. However, it is hard for a human to write a program in machine code. Software engineers developed the first programs this way. It was possible for early computers because of their simplicity. Modern computers are much more powerful and complex devices. Their programs are huge and have a lot of functions.

Computer engineers invented two types of special applications. They solve the problem of the machine code complexity. These applications are **compilers**¹⁴⁶ and **interpreters**¹⁴⁷. They translate the program from a human-readable language into machine code. Compilers and interpreters solve this task differently.

Software developers use **programming languages**¹⁴⁸ in their work nowadays. Compilers and interpreters take programs written in such languages and produce the corresponding machine instructions.

Humans use one of **natural languages**¹⁴⁹ to communicate with each other. Programming languages are different from them. They are formal and very limited. Using a programming language, you can express only actions that a computer can perform. There are strict rules on how you should write these actions. For example, you can use a small set of words and combine them in specific orders. **Source code**¹⁵⁰ is a text of the program you write in a programming language.

The compiler and interpreter process source code differently. The compiler reads the entire program text, generates machine instructions and saves them on a disk drive. The compiler does not execute the resulting program on its own. The interpreter reads the source code in parts, generates machine instructions and executes them immediately. The interpreter stores its results in RAM temporarily. When the program finishes, you lose these results.

Let's consider how the compiler works step by step. Suppose that you have written the program. You saved its source code to a file on the hard disk. Then you run a compiler that fits the language you have used. Each programming language has the corresponding compiler or interpreter. The compiler reads your file, processes it and writes the resulting machine instructions in the executable file on a disk. Now you have two files: one with source code and one with machine instructions. Every time you change the source code file, you should generate the new executable file. You can run the executable file to launch your program.

Figure 1-15 shows how the compiler processes a program written in C and C++ languages.

¹⁴⁶<https://en.wikipedia.org/wiki/Compiler>

¹⁴⁷[https://en.wikipedia.org/wiki/Interpreter_\(computing\)](https://en.wikipedia.org/wiki/Interpreter_(computing))

¹⁴⁸https://en.wikipedia.org/wiki/Programming_language

¹⁴⁹https://en.wikipedia.org/wiki/Natural_language

¹⁵⁰https://en.wikipedia.org/wiki/Source_code

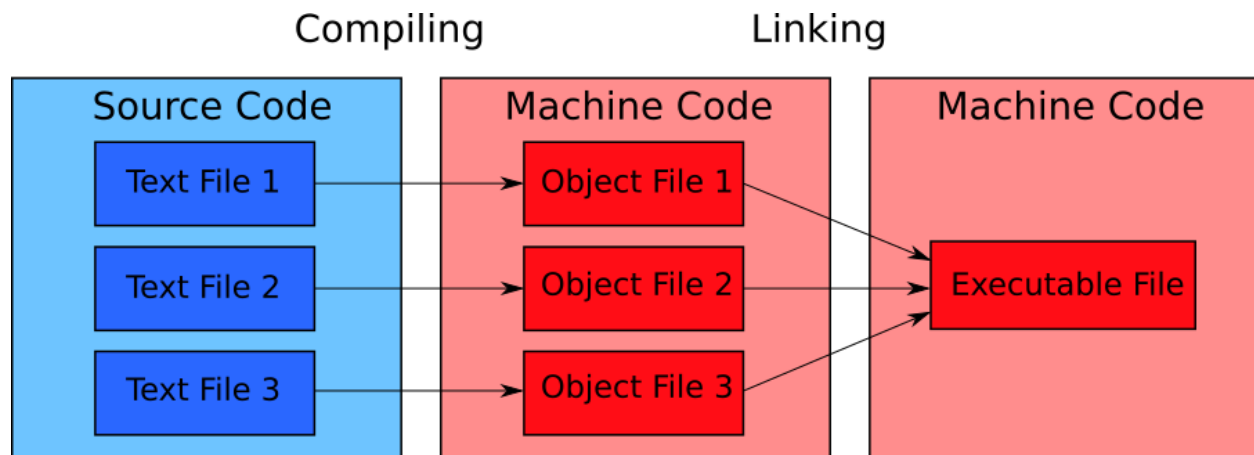


Figure 1-15. The compilation process

The compilation process consists of two steps. The compiler does the first step. The second step is called **linking**. The special program called **linker**¹⁵¹ performs it.

The compiler produces intermediate **object files**. The linker takes them and converts them to one executable file.

Why do you need two steps for compiling the source code? In theory, you can combine the compiler and linker into a single application. However, such a solution has several problems.

The limited RAM size causes the first problem. There is a common practice to split source code into several files. Each file matches a separate part of the program that solves the specific task. This way, you simplify your work with the source code. The compiler processes these files separately. It produces an object file for each source code file. They store the intermediate results of compilation.

If you combine the compiler and linker into one application, storing the intermediate results on the disk becomes a controversial decision. If you do it, you get the bottleneck because of slow disk operations. In theory, you can keep all data in RAM and get better performance. However, you cannot do it. When you deal with a big program, the compilation process consumes all your RAM and crashes.

Suppose that you have a combined compiler-linker that stores temporary files on the disk. In this case, storing temporary files brings the performance overhead. At the same time, you do not get any benefits from it. You avoided the RAM limitation this way. However, you can get the benefit by splitting the compiler and linker. Then you simplify both applications and make it cheaper to support them. The developers of compilers chose this way.

The second problem of the compiler-linker application is resolving dependencies. There are blocks of commands that call each other in the source code. Such references are called **dependencies**. Tracking them is the linker task.

When the compiler produces the object files, they contain machine code but not the source code. It is simpler for the linker to track dependencies in the machine code.

¹⁵¹[https://en.wikipedia.org/wiki/Linker_\(computing\)](https://en.wikipedia.org/wiki/Linker_(computing))

If you combine compiler and linker, you need extra passes through the whole program source code for resolving dependencies. The compiler needs much more time for a single pass over the source code than the linker for processing the machine code. Therefore, when you have the compiler and linker separated, you speed up the overall compilation process.

The program can call blocks of commands from the library. The linker process the library file together with the object files of your program in this case. The compiler cannot process the library. Its file contains machine code but not the source code. Therefore, the compiler does not understand it. Splitting the compilation into two steps resolves the task of using libraries too.

We have considered the basics of how the compiler works. Now suppose that you choose an interpreter instead to execute your program. You have the file with its source code on the disk drive. The file is ready for execution. When you run it, the OS loads the interpreter first. Then the interpreter reads your source code file into RAM and executes it line by line. The translation of source code commands to machine instructions happens in RAM. Some interpreters save files with an intermediate representation of the program to the disk drive. It speeds up the program execution if you restart it. However, you always need to run an interpreter first for executing your program.

Figure 1-16 shows the process of interpreting the program.

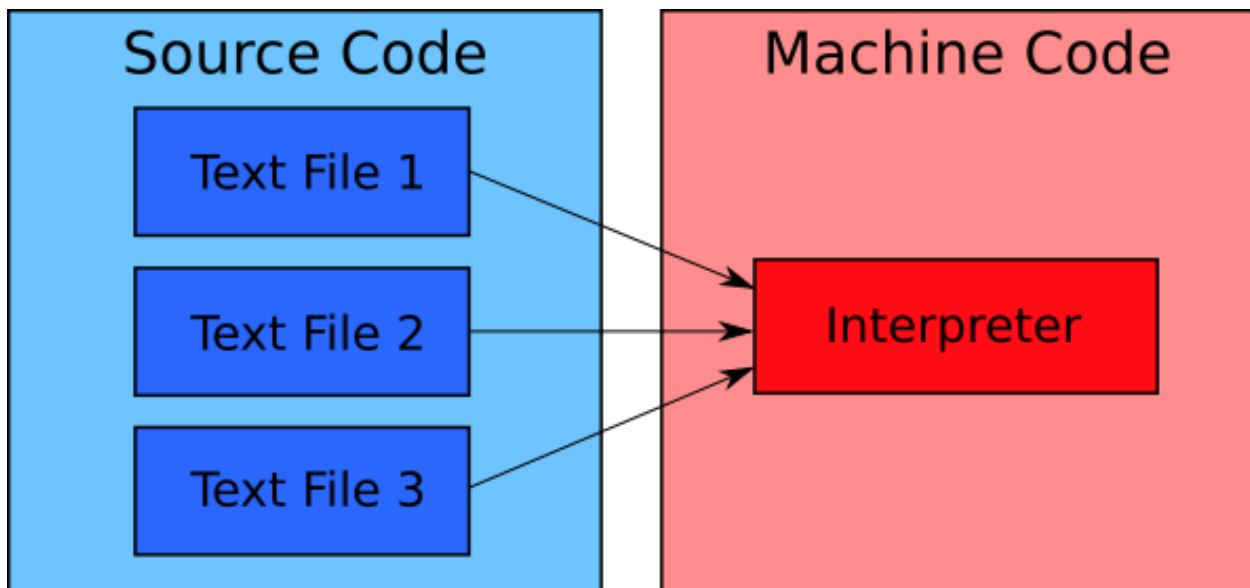


Figure 1-16. Interpreting the program

Figure 1-16 can give an idea that the interpreter works the same way as the compiler and linker combined into one application. The interpreter loads source code files into RAM and translates them into machine code. Why are there no problems with the RAM overflow and dependency resolution?

The interpreter avoids problems because it processes the source code differently than the compiler does. The interpreter processes and executes the program code line by line. Therefore, it does not store the machine code of the whole program in memory. The interpreter processes the parts of the source code file that it requires at the moment. When the interpreter executes them, it unloads these

parts and frees the corresponding RAM area.

Interpreting the program looks more convenient for software development than compiling. However, it has some drawbacks.

First, all interpreters work slowly. It happens because every time you run the program, the interpreter should translate its source code to machine code. This process takes some time. You should wait for it. Another reason for the low performance of interpreters is disk operations. Loading the program's source code from the disk drive into RAM causes the CPU to idle. It takes up to 10000000 clock cycles, according to Table 1-3.

Second, the interpreter itself is a complex program. It requires some portion of the computer's hardware resources to launch and work. Therefore, the computer executes both interpreter and your program in parallel and shares resources among them. It is an extra overhead that reduces the performance of your program.

Interpreting the program is slow. Does it mean that compilation is better? The compiler generates an executable file with machine instructions. Therefore, you reach almost the program's performance when you compile it or write machine instructions on your own. However, you pay for using the programming language at the compilation stage. A couple of seconds and a few megabytes of RAM are enough to compile a small program. When you compile a large project (for example, the Linux kernel), it takes several hours. If you change the source code, you should recompile the project and wait hours again.

Keep in mind the overhead of interpreters and compilers when choosing a programming language for your project. The interpreter is a good choice in the following cases:

- You want to develop a program quickly.
- You do not care about the program's performance.
- You work on a small and relatively simple project.

The compiler would be better in the following cases:

- You work on a complex and large project.
- Your program should work as fast as possible.
- You want to speed up debugging of your program.

Both compilers and interpreters have an overhead. Does it make sense to discard a programming language and write a program in machine code? You do not waste your time waiting for compilation in this case. Your program works as fast as possible. These benefits sound reasonable. Please do not hurry with your conclusions.

One simple example helps you to realize all advantages of using a programming language. Listing 1-1 shows the source code written in C. This program prints the "Hello world!" text on the screen.

Listing 1-1. Source code of the C program

```
1 #include <stdio.h>
2
3 int main(void)
4 {
5     printf("Hello world!\n");
6 }
```

Listing 1-2 shows the machine instructions of this program in the hexadecimal format.

Listing 1-2. Machine instructions of the program

```
1 BB 11 01 B9 0D 00 B4 0E 8A 07 43 CD 10 E2 F9
2 CD 20 48 65 6C 6C 6F 2C 20 57 6F 72 6C 64 21
```

Even if you don't know the C language, you would prefer to deal with the code in Listing 1-1. You can read and edit it easily. At the same time, you need significant efforts to decode the numbers in Listing 1-2.

Perhaps a professional programmer with huge experience can write a small program in machine code. However, another programmer will spend a lot of time and effort to figure it out. Developing a large project in machine code is a challenging and time-consuming task for any developer.

Using programming language saves your effort and time significantly when developing programs. Also, it reduces the cost of maintaining the existing project. There is no way to develop modern complex software using the machine code only.

Bash Shell

Programming is an applied skill. If you want to learn it, you should choose a programming language and solve tasks. This is the only way to get practical skills.

We use the Bash language in this book. This language is convenient for automating computer administration tasks. Here are few examples of what you can do with Bash:

- Create data backups.
- Manipulate [directories](#)¹⁵² and files.
- Run programs and transfer data between them.

Bash was developed in the Unix environment. Therefore, it bears the imprint of the [Unix philosophy](#)¹⁵³. Despite its roots, you can also use Bash on Windows and macOS.

Development Tools

You need a Bash interpreter and a terminal emulator to run the examples of this chapter. You can install them on all modern operating systems. Let's take a look at how to do this.

Bash Interpreter

Bash is a [script programming language](#)¹⁵⁴. It has the following features:

1. It is interpreted language.
2. It operates existing programs or high-level commands.
3. You can use it as a shell to access the OS functions.

If you use Linux or macOS, you have the preinstalled Bash interpreter. If your OS is Windows, you need both Bash interpreter and POSIX-compatible environment. Bash needs this environment to work correctly. There are two ways to install it.



You can meet the “Unix environment” and “Linux environment” terms. They both mean a software environment that is compatible with POSIX standards.

¹⁵²[https://en.wikipedia.org/wiki/Directory_\(computing\)](https://en.wikipedia.org/wiki/Directory_(computing))

¹⁵³https://en.wikipedia.org/wiki/Unix_philosophy

¹⁵⁴https://en.wikipedia.org/wiki/Scripting_language

The first option is to install the [MinGW¹⁵⁵](#) toolkit. It contains the [GNU compiler collection¹⁵⁶](#) in addition to Bash. If you do not need all MinGW features, you can install Minimal SYStem (MSYS) instead. MSYS is the MinGW component that includes Bash, a terminal emulator and [GNU utilities¹⁵⁷](#). These three things make up a minimal Unix environment.

It is always good to clarify the bitness of your Windows before installing any software. Here are steps to read it:

1. If you have a “Computer” icon on your desktop, right-click on it and select the “Properties” item.
2. If there is no “Computer” icon on your desktop, click the “Start” button. Find the “Computer” item in the menu. Right-click on it and select “Properties”.
3. You have opened the “System” window. Locate the “System Type” item there as Figure 2-1 demonstrates. This item shows you the bitness of Windows.

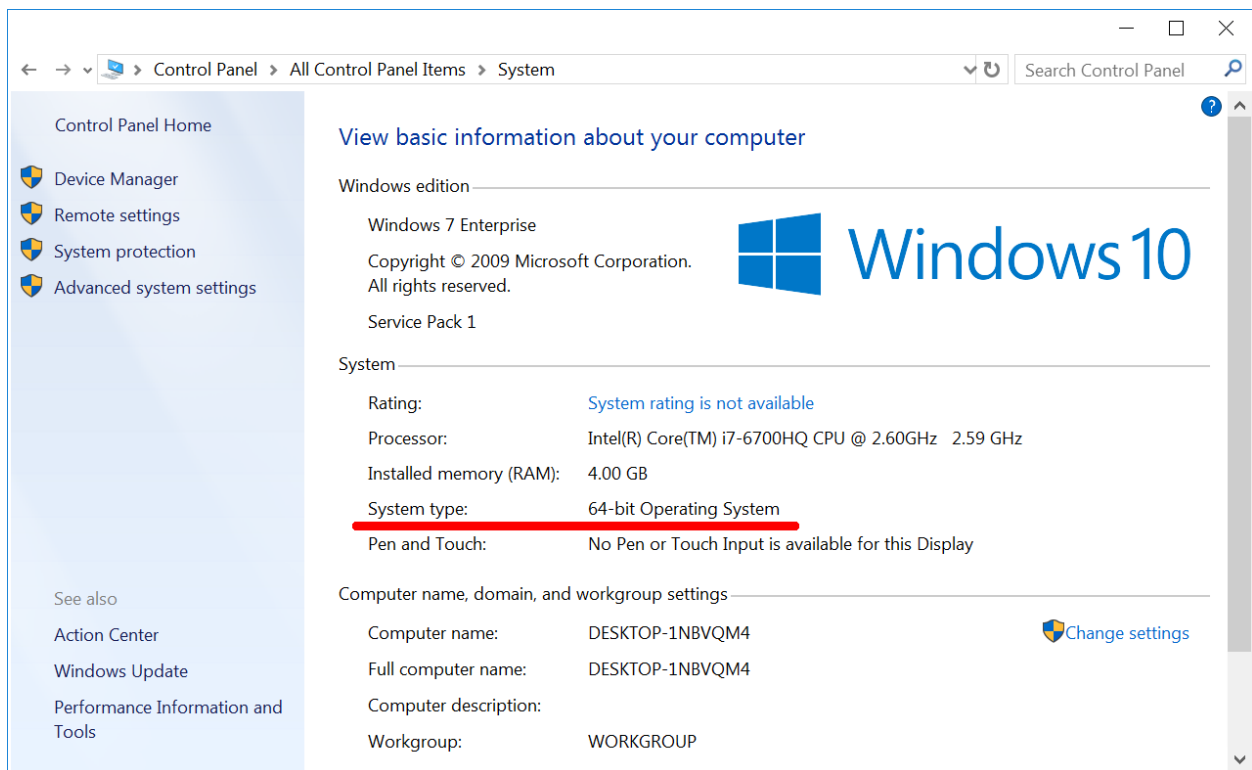


Figure 2-1. System Type

We are going to install the modern MSYS version called MSYS2. Download its installer from the [official website¹⁵⁸](#). You should choose the installer version that fits the bitness of your Windows.

Now we have everything to install MSYS2. Follow these steps for doing it:

¹⁵⁵<https://en.wikipedia.org/wiki/MinGW>

¹⁵⁶https://en.wikipedia.org/wiki/GNU_Compiler_Collection

¹⁵⁷https://en.wikipedia.org/wiki/GNU_Core_Utilities

¹⁵⁸<https://www.msys2.org>

1. Run the installer file. You will see the window as Figure 2-2 shows.

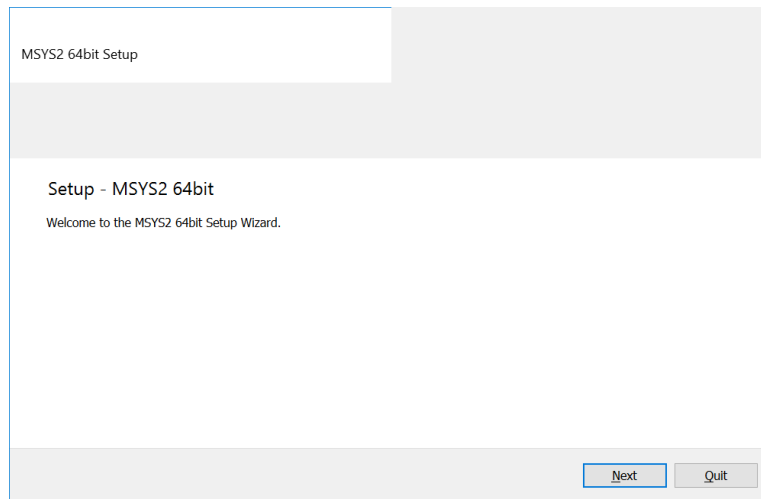


Figure 2-2. MSYS2 installation dialog

2. Click the “Next” button.
3. You see the new window as Figure 2-3 shows. Specify the installation [**path**](https://en.wikipedia.org/wiki/Path_(computing))¹⁵⁹ there and press the “Next” button.

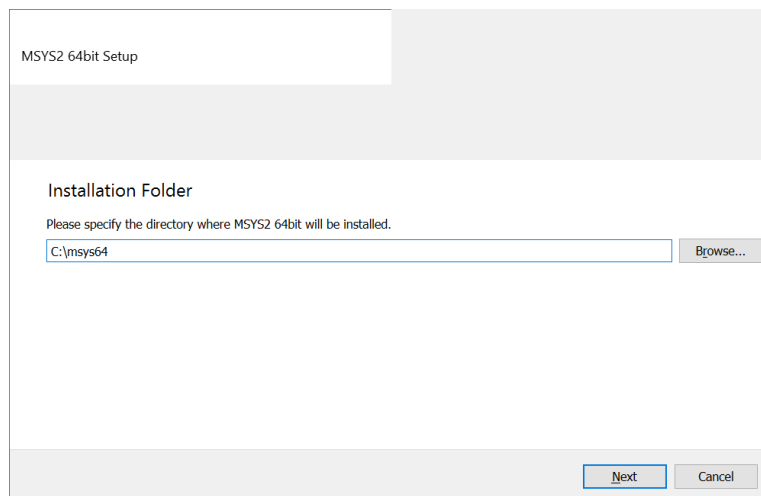


Figure 2-3. Selecting the installation path

4. The next window suggests you to choose the application name for the Windows “Start” menu. Leave it unchanged and click the “Next” button. Then the installation process starts.
5. When the installation finishes, click the “Finish” button. It closes the installer window.

¹⁵⁹[https://en.wikipedia.org/wiki/Path_\(computing\)](https://en.wikipedia.org/wiki/Path_(computing))

You have installed the MSYS2 Unix environment on your hard drive. You can find its files in the `C:\msys64` directory if you did not change the default installation path. Go to this directory and run the `msys2.exe` file. It opens the window where you can work with the Bash shell. Figure 2-4 shows this window.

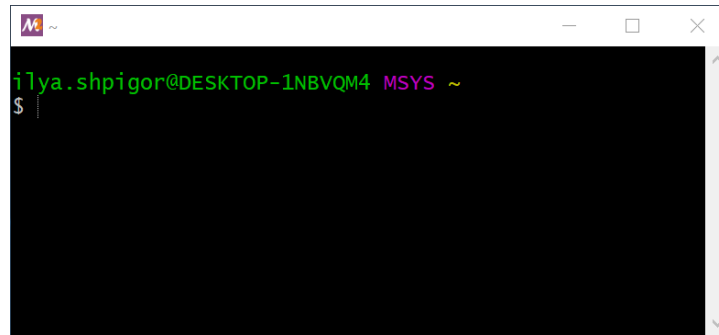


Figure 2-4. The Bash shell

The second option is to install a Unix environment from Microsoft. It is called [Windows subsystem for Linux](#)¹⁶⁰ (WSL). This environment is available for Windows 10 only. It does not work on Windows 8 and 7. You can find the manual to install WSL on the [Microsoft website](#)¹⁶¹.

If you use Linux, you do not need to install Bash. You already have it. Just press the shortcut key `Ctrl+Alt+T` to open a window with the Bash shell.

If you use macOS, you have everything to launch Bash too. Here are the steps for doing that:

1. Click the magnifying glass icon in the upper right corner of the screen. It opens the Spotlight search program.
2. The dialog box appears. Enter the text “Terminal” there.
3. Spotlight shows you a list of applications. Click on the first line in the list with the “Terminal” text.

Terminal emulator

Bash shell is not a regular GUI application. It even does not have its own window. When you run the `msys2.exe` file, it opens a window of the terminal emulator program.

An [emulator](#)¹⁶² is a program that mimics the behavior of another program, OS or device. The emulator solves the compatibility task. For example, you want to run a Windows program on Linux. There are several ways to do that. One option is using the emulator of the Windows environment for Linux. It is called [Wine](#)¹⁶³. Wine provides its own version of the Windows system libraries. When you run your program, it uses these libraries and supposes that it works on Windows.

¹⁶⁰https://en.wikipedia.org/wiki/Windows_Subsystem_for_Linux

¹⁶¹<https://docs.microsoft.com/en-us/windows/wsl/install-win10>

¹⁶²<https://en.wikipedia.org/wiki/Emulator>

¹⁶³[https://en.wikipedia.org/wiki/Wine_\(software\)](https://en.wikipedia.org/wiki/Wine_(software))

The terminal emulator solves the compatibility task too. Command-line programs are designed to work through a terminal device. Nobody uses such devices today. Cheap personal computers and laptops have replaced them. However, there are still many programs that require a terminal for working. You can run them using the terminal emulator. It uses the shell to pass data to the program. When the program returns some results, the shell passes them to the terminal emulator. Then the emulator displays the results on the screen.

Figure 2-5 explains the interaction between input/output devices, the terminal emulator, the shell and the command-line program.

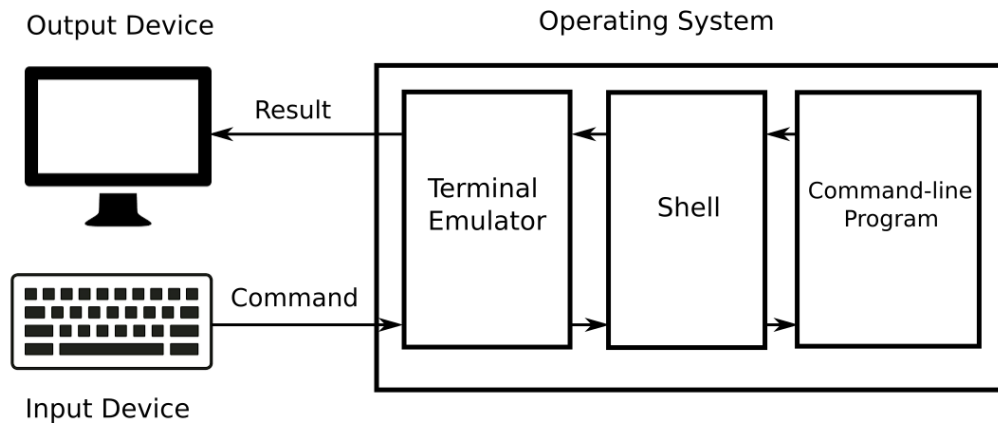


Figure 2-5. The workflow of the terminal emulator

The terminal emulator window in Figure 2-4 shows the following two lines:

```
ilya.shpigor@DESKTOP-1NBVQM4 MSYS ~
$
```

The first line starts with the username. The username is `ilya.shpigor` in my case. Then there is the `@` symbol and the computer name `DESKTOP-1NBVQM4`. You can change the computer name via Windows settings. The word `MSYS` comes next. It means the platform where Bash is running. At the end of the line, there is the symbol `~`. It is the path to the current directory.

Command Interpreter

All interpreters have two working modes: non-interactive and interactive. When you choose the non-interactive mode, the interpreter loads the source code file from the disk and executes it. You do not need to type any commands or control the interpreter. It does everything on its own.

When you choose the interactive mode, you type each command to the interpreter manually. If the interpreter is integrated with the OS and works in the interactive mode, it is called a **command shell**¹⁶⁴ or shell.

¹⁶⁴[https://en.wikipedia.org/wiki/Shell_\(computing\)](https://en.wikipedia.org/wiki/Shell_(computing))

A command shell provides access to the settings and functions of the OS. You can perform the following tasks using it:

- Run programs and system services.
- Manage the file system.
- Control peripherals and internal devices.
- Access the kernel features.

Demand of the CLI

Why would somebody learn the command-line interface (CLI) today? It appeared 40 years ago for computers that are thousands of times slower than today. Then the graphical interface supplanted CLI on PCs and laptops. Everybody prefers to use GUI nowadays.

The CLI seems to be outdated technology. However, this statement is wrong. It should be a reason why developers include Bash in all modern macOS and [Linux distributions](#)¹⁶⁵. Windows also has a command shell called `Cmd.exe`¹⁶⁶. Microsoft has replaced it with `PowerShell`¹⁶⁷ in 2006. Just think about this fact. The developer of the most popular desktop OS has created a new command shell in the 2000s. All these points confirm that CLI is still in demand.

What tasks does the shell perform in modern OSes? First of all, it is a tool for [system administration](#)¹⁶⁸. The OS consist of the kernel and software modules. These modules are libraries, services and system utilities. Most of the modules have settings and special modes of operation. You do not need them in your daily work. Therefore, you cannot access these settings via GUI in most cases.



The Bash shell has good integration with Linux and macOS. You can access most OS functions using Bash there. However, it does not work with Windows smoothly. Microsoft offers you their own PowerShell. We study Bash in this book for two reasons. First, it is compatible with the POSIX standard. Second, Bash is more common than PowerShell. You can use it on all modern OSes.

If the OS fails, you need system utilities to recover it. They have a command-line interface because a GUI often is not available after the failure.

Besides the administration tasks, you would need CLI when [connecting computers over a network](#)¹⁶⁹. There are GUI programs for such connection. The examples are TeamViewer and Remote Desktop. They require a stable and fast network connection for working well. If the connection is not reliable, the GUI programs are slow and often fail. The command interface does not have such a limitation. The remote server receives your command even if the link is poor.

¹⁶⁵https://en.wikipedia.org/wiki/Linux_distribution

¹⁶⁶<https://en.wikipedia.org/wiki/Cmd.exe>

¹⁶⁷<https://en.wikipedia.org/wiki/PowerShell>

¹⁶⁸https://en.wikipedia.org/wiki/System_administrator

¹⁶⁹https://en.wikipedia.org/wiki/Remote_desktop_software

You can say that a regular user does not deal with administration tasks and network connections. Even if you do not have such tasks, using command shell speeds up your daily work with the computer. Here are few things that you can do more effectively with CLI than with GUI:

- Operations on files and directories.
- Creating data backups.
- Downloading files from the Internet.
- Collecting statistics about your computer's resource usage.

An example will explain the effectiveness of CLI. Suppose you rename several files on the disk. You add the same suffix to their names. If you have a dozen of files, you can do this task with [Windows Explorer](#)¹⁷⁰ in a couple of minutes. Now imagine that you should rename thousands of files this way. You will spend the whole day doing that with the Explorer. If you use the shell, you need to launch a single command and wait for several seconds. It will rename all your files automatically.

The example with renaming files shows the strength of the CLI that is scalability. Scalability means that the same solution handles well both small and large amounts of input data. The solution implies a command when we are talking about the shell. The command renames ten and a thousand files with the same speed.

Experience with the command interface is a great benefit for any programmer. When you develop a complex project, you manage plenty of text files with the source code. You use the GUI editor to change the single file. It works well until you do not need to introduce the same change in many files. For example, it can be the new version of the license information in the file headers. You waste your time when solving such a task with the editor. Command-line utilities make this change much faster.

You need to understand the CLI to deal with compilers and interpreters. These programs usually do not have a graphical interface. You should run them via the command line and pass the names of the source code files. The reason for such workflow is the poor scalability of the GUI. If you have plenty of source code files, you cannot handle them effectively via the GUI.

There are special programs to edit and compile source code. Such programs are called **integrated development environments**¹⁷¹ (IDE). You can compile a big project using IDE and its GUI. However, IDE calls the compiler via the command line internally. Therefore, you should deal with the compiler's CLI if you want to change its options or working mode.

If you are an experienced programmer, knowing the CLI encourages you to develop helper utilities. It happens because writing a program with a command interface is much faster than with a GUI. The speed of development is essential when solving one-off tasks.

Here is an example situation when you would need to write a helper utility. Suppose that you have to make a massive change in the source code of your project. You can do it with IDE by repeating the same action many times.

¹⁷⁰https://en.wikipedia.org/wiki/File_Explorer

¹⁷¹https://en.wikipedia.org/wiki/Integrated_development_environment

Another option is to spend time writing a utility that will do this job. You should compare the required time for both ways of solving your task. If you are going to write a GUI helper utility, it takes more time than for a CLI utility. This can lead you to the wrong decision to solve the task manually using the IDE. Automating your job is the best option in most cases. It saves your time and helps to avoid mistakes.

You decide if you need to learn the CLI. I have only given few examples of when it is beneficial. It is hard to switch from using a GUI to a CLI. You have to re-learn many things that you do with Windows Explorer regularly. But once you get the hang of the command shell, your new productivity will surprise you.

Navigating the File System

Let's start introducing the Unix environment and Bash with a **file system**¹⁷². A file system is a software that dictates how to store and read data from disks. It covers the following topics:

- API to access data on the disk that programs can use.
- Universal way for accessing different storage devices.
- Physical operations on the disk storage.

First, we will look at the differences between the directory structure in Unix and Windows. Then we will learn the Bash commands for navigating the file system.

Directory Structure

There is an address bar at the top of the Windows Explorer window. It displays the **absolute path** to the current directory. An absolute path shows the place of the file system object regardless of the current directory.

Another way to specify the file system object place is using the **relative path**. It shows you how to reach the object from the current directory.

A **directory**¹⁷³ is a file system cataloging structure. It can contain files and other directories. Windows terminology calls it **folder**¹⁷⁴. Both names mean the same kind of file system object.

Figure 2-6 shows an Explorer window. The address bar equals `This PC > Local Disk (C:) > msys64` there. It matches the `C:\msys64` absolute path. Thus, we see the contents of the `msys64` directory on the C drive in the Explorer window.

The letter C in the path denotes the local system disk drive. The local drive means the device that is connected to your computer physically. You can also have a network drive. You access such a device via the network. The system disk means one that has the Windows installation on it.

¹⁷²https://en.wikipedia.org/wiki/File_system

¹⁷³[https://en.wikipedia.org/wiki/Directory_\(computing\)](https://en.wikipedia.org/wiki/Directory_(computing))

¹⁷⁴[https://en.wikipedia.org/wiki/Directory_\(computing\)#Folder_metaphor](https://en.wikipedia.org/wiki/Directory_(computing)#Folder_metaphor)

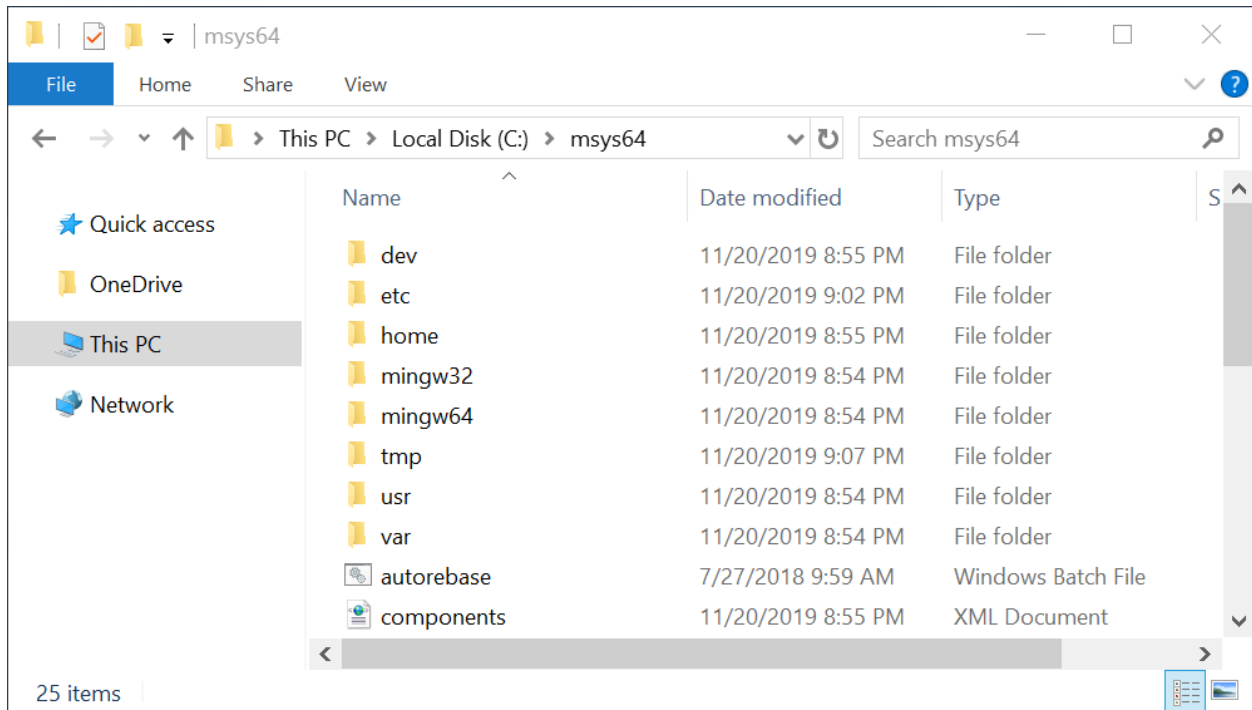


Figure 2-6. Windows Explorer

If you run the MSYS2 terminal emulator, it shows you the current absolute path at the end of the first line. This line behaves like the address bar of Windows Explorer. When you change the current directory, the current path changes too. However, you have to consider that the terminal and Explorer show you different paths for the same current directory. It happens because directory structures of the Unix environment and Windows do not match.

Windows marks each disk drive with a Latin letter. You can open the drive using Explorer as a regular folder. Then you access its content.

For example, let's open the C system drive. It has a [standard set of directories](#)¹⁷⁵. Windows has created them during the installation process. If you open the C drive in Explorer, you see the following directories there:

- Windows
- Program Files
- Program Files (x86)
- Users
- PerfLogs

These directories store OS components, user applications and temporary files.

You can connect extra disk drives to your computer. Another option is to split a single disk into several logical partitions. Windows will assign the Latin letters (D, E, F, etc) to these extra disks and

¹⁷⁵https://en.wikipedia.org/wiki/Directory_structure#Windows_10

partitions. You are allowed to create any directory structure there. Windows does not restrict you in any way.

The [File Allocation Table](#)¹⁷⁶ (FAT) file system dictates how Windows manages disks and provides you access to them. Microsoft developed this file system for the [MS-DOS](#)¹⁷⁷ OS. The principles of FAT became the basis of the [ECMA-107](#)¹⁷⁸ standard. The next-generation file system from Microsoft is called [NTFS](#)¹⁷⁹. It replaced the obsolete FAT in modern versions of Windows. However, the basic principles of disks and directory structure are the same in NAT and FAT. The reason for that is the backward compatibility requirement.

The Unix directory structure follows the [POSIX standard](#)¹⁸⁰. This structure gives you less freedom than the Windows one. It has several predefined directories that you cannot move or rename. You are allowed to put your data in the specific paths only.

The POSIX standard says that the file system should have a top-level directory. It is called the [root directory](#)¹⁸¹. The slash sign / denotes it. All directories and files of all connected disk drives are inside the root directory.

If you want to access the contents of a disk drive, you should mount it. **Mounting** means embedding the contents of a disk into the root directory. When mounting is done, you can access the disk contents through some path. This path is called a [mount point](#)¹⁸². If you go to the mount point, you enter the file system of the disk.

Let's compare the Windows and Unix directory structures by example. Suppose that your Windows computer has two local disks C and D. Listing 2-1 shows their directory structure.

Listing 2-1. The directory structure in Windows

```
C: \
  PerfLogs\
  Windows\
  Program Files\
  Program Files (x86)\
  Users\

D: \
  Documents\
  Install\
```

Suppose that you have installed the Unix environment on your Windows. Then you run the terminal emulator and get the directory structure from Listing 2-2.

¹⁷⁶https://en.wikipedia.org/wiki/File_Allocation_Table

¹⁷⁷<https://en.wikipedia.org/wiki/MS-DOS>

¹⁷⁸<http://www.ecma-international.org/publications/standards/Ecma-107.htm>

¹⁷⁹<https://en.wikipedia.org/wiki/NTFS>

¹⁸⁰https://pubs.opengroup.org/onlinepubs/9699919799/basedefs/V1_chap10.html#tag_10

¹⁸¹https://en.wikipedia.org/wiki/Root_directory

¹⁸²[https://en.wikipedia.org/wiki/Mount_\(computing\)](https://en.wikipedia.org/wiki/Mount_(computing))

Listing 2-2. The directory structure in Unix

```
/
  c/
    PerfLogs/
    Windows/
    Program Files/
    Program Files (x86)/
    Users/

  d/
    Documents/
    Install/
```

Since you launch the MSYS2 terminal, you enter the Unix environment. Windows paths don't work there. You should use Unix paths instead. For example, you can access the `C:\Windows` directory via the `/c/Windows` path only.

There is another crucial difference between Unix and Windows file systems besides the directory structure. The **character case**¹⁸³ makes strings differ in the Unix environment. It means that two words with the same letters are not the same if their character case differs. For example, the `Documents` and `documents` words are not equal. Windows has no case sensitivity. If you type the `c:\windows` path in the Explorer address bar, it opens the `C:\Windows` directory. This approach does not work in the Unix environment. You should type all characters in the proper case.

Here is the last point to mention regarding Unix and Windows file systems. Use the slash sign `/` to separate directories and files in Unix paths. When you work with Windows paths, you use backslash `\` for that.

File System Navigation Commands

We are ready to learn our first Bash commands. Here are the steps to execute a shell command:

1. Open the terminal window.
2. Type the command there.
3. Press Enter.

The shell will execute your command.

When the shell is busy, it cannot process your input. You can distinguish the shell's state by the **command prompt**¹⁸⁴. It is a sequence of one or more characters. The default prompt is the dollar

¹⁸³https://en.wikipedia.org/wiki/Case_sensitivity

¹⁸⁴https://en.wikipedia.org/wiki/Command-line_interface#Command_prompt

sign \$. You can see it in Figure 2-4. If the shell prints the prompt, it is ready for executing your command.

Windows Explorer allows you the following actions to navigate the file system:

- Display the current directory.
- Go to a specified disk drive or directory.
- Find a directory or file on the disk.

You can do the same actions with the shell. It provides you a corresponding command for each action. Table 2-1 shows these commands.

Table 2-1. Commands and utilities for navigating the file system

Command	Description	Examples
ls	Display the contents of the directory. If you call the command without parameters, it shows you the contents of the current directory.	ls ls /c/Windows
pwd	Display the path to the current directory. When you add the -w parameter, the command displays the path in the Windows directory structure.	pwd
cd	Go to the directory at the specified relative or absolute path.	cd tmp cd /c/Windows cd ..
mount	Mount the disk to the root file system. If you call the command without parameters, it shows a list of all mounted disks.	mount
find	Find a file or directory. The first parameter specifies the directory to start searching.	find . -name vim find /c/Windows -name *vim*
grep	Find a file by its contents.	grep "PATH" * grep -Rn "PATH" . grep "PATH" * .*

Bash can perform `pwd` and `cd` commands of Table 2-1 only. They are called **built-ins**¹⁸⁵. Special utilities perform all other commands of the table. Bash calls an appropriate utility if it cannot execute your command on its own.

The MSYS2 environment provides a set of GNU utilities. These are auxiliary highly specialized programs. They give you access to the OS features in Linux and macOS. However, their capabilities are limited in Windows. Bash calls GNU utilities to execute the following commands of Table 2-1:

¹⁸⁵https://www.gnu.org/software/bash/manual/html_node/Bash-Builtins.html

- ls
- mount
- find
- grep

When you read an article about Bash on the Internet, its author can confuse the “command” and “utility” terms. He names both things “commands”. This is not a big issue. However, I recommend you to distinguish them. Calling a utility takes more time than calling Bash built-in. It causes performance overhead in some cases.

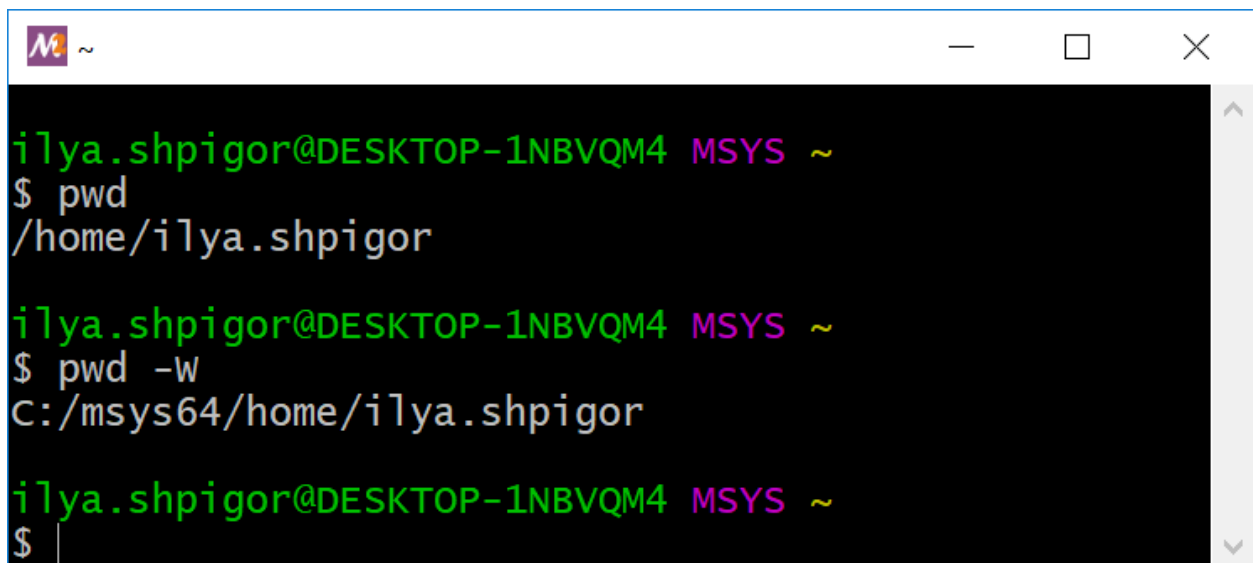
pwd

Let’s consider the commands in Table 2-1. You have just started the terminal. The first thing you do is to find out the current directory. You can get it from the command prompt, but it depends on your Bash configuration. You do not have this feature enabled by default in Linux and macOS.

When you start the terminal, it opens the home directory of the current user. Bash abbreviates this path by the tilde symbol `~`. You see this symbol before the command prompt. Use tilde instead of the home directory absolute path. It makes your commands shorter.

Call the `pwd` command to get the current directory. Figure 2-7 shows this call and its output. The command prints the absolute path to the user’s home directory. It equals `/home/ilya.shpigor` in my case.

If you add the `-w` option to the call, the command prints the path in the Windows directory structure. It is useful when you create a file in the MSYS2 environment and open it in a Windows application afterward. Figure 2-7 shows you the result of applying the `-w` option.

A screenshot of a terminal window with a dark background and light-colored text. The window title bar shows a purple icon and a tilde symbol. The terminal content shows three lines of command and output. The first line shows the prompt `ilya.shpigor@DESKTOP-1NBVQM4 MSYS ~` followed by `$ pwd` and the output `/home/ilya.shpigor`. The second line shows the prompt `ilya.shpigor@DESKTOP-1NBVQM4 MSYS ~` followed by `$ pwd -w` and the output `C:/msys64/home/ilya.shpigor`. The third line shows the prompt `ilya.shpigor@DESKTOP-1NBVQM4 MSYS ~` followed by `$` and a vertical cursor bar.

```
ilya.shpigor@DESKTOP-1NBVQM4 MSYS ~
$ pwd
/home/ilya.shpigor

ilya.shpigor@DESKTOP-1NBVQM4 MSYS ~
$ pwd -w
C:/msys64/home/ilya.shpigor

ilya.shpigor@DESKTOP-1NBVQM4 MSYS ~
$
```

Figure 2-7. The output of the `pwd` command

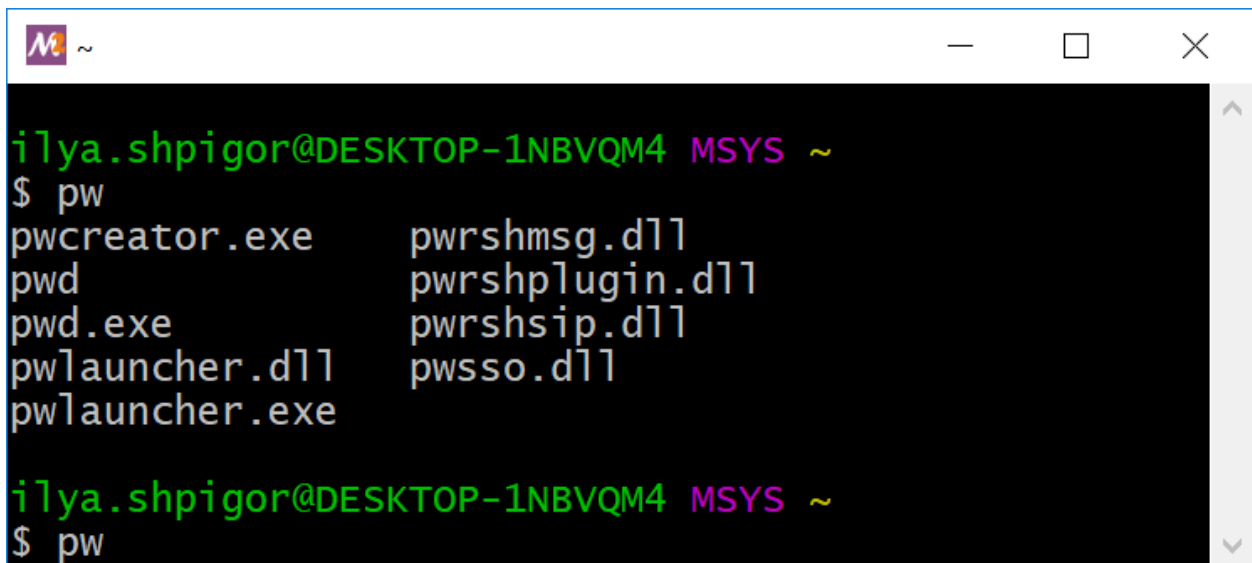
What is a command option? When the program has a CLI only, you have very limited ways to interact with it. The program needs some data on input to do its job. The shell provides you a simple way to pass these data. Just type them after the command name. These data are called **arguments** of the program. Bash terminology distinguishes two kinds of arguments: parameter and option. A **parameter**¹⁸⁶ is the regular word or character you pass to the program. An **option** or **key** is an argument that switches the mode of a program. The standard dictates the option format. It is a word or character that starts with a dash - or a double dash --.

You pass data to the CLI programs and Bash built-ins in the same way. Use parameters and options for doing that.

Typing long commands is inconvenient. Bash provides the autocomplete feature to save your time. Here are the steps for using it:

1. Type the first few letters of the command.
2. Press the Tab key.
3. If Bash finds the command you mean, it completes it.
4. If several commands start with the typed letters, autocomplete does not happen. Press Tab again to see the list of these commands.

Figure 2-8 demonstrates how the autocomplete feature works. Suppose that you type the “pw” word. Then you press the Tab key twice. Bash shows you the commands that start with “pwd” as Figure 2-8 shows.



```
ilya.shpigor@DESKTOP-1NBVQM4 MSYS ~
$ pw
pwcreator.exe      pwrshmsg.dll
pwd                pwrshplugin.dll
pwd.exe            pwrshsip.dll
pwlauncher.dll    pwsso.dll
pwlauncher.exe
```

```
ilya.shpigor@DESKTOP-1NBVQM4 MSYS ~
$ pw
```

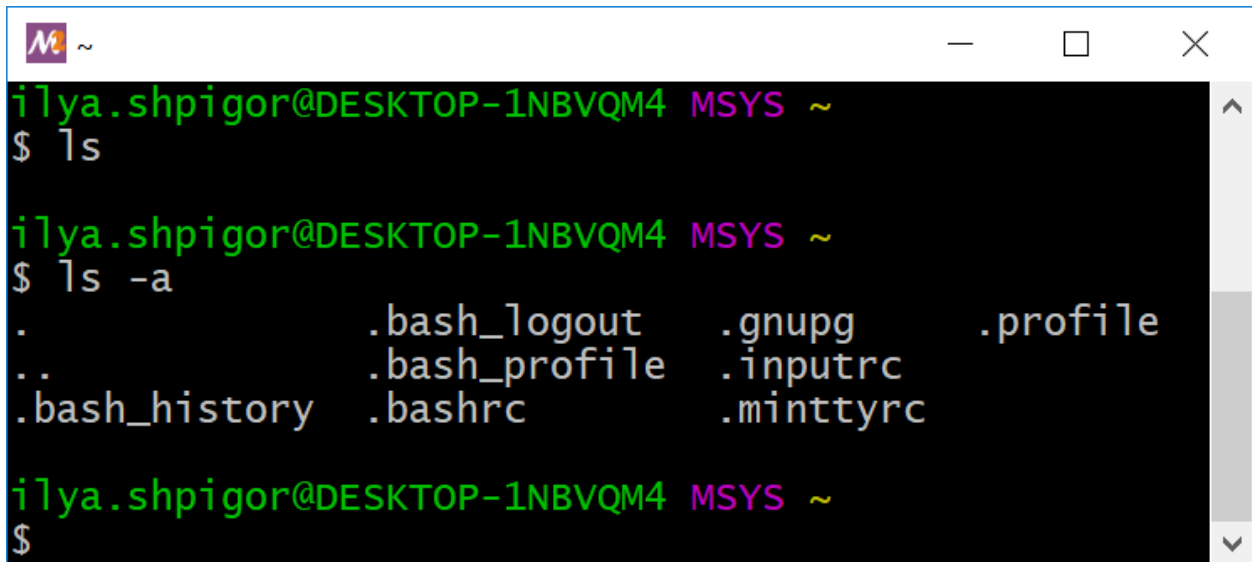
Figure 2-8. Autocomplete for the pw command

¹⁸⁶https://en.wikipedia.org/wiki/Command-line_interface#Arguments

ls

We got the current directory using the `pwd` command. The next step is checking the directory content. The `ls` utility does this task.

Suppose that you have just installed the MSYS2 environment. Then you launched the terminal first time. You are in the user's home directory. Call the “`ls`” command there. Figure 2-9 shows its result. The command output has nothing. It means that the directory is empty or has hidden files and directories only.

A screenshot of a terminal window with a black background and green text. The prompt is 'ilya.shpigor@DESKTOP-1NBVQM4 MSYS ~'. The first command is '\$ ls', which produces no output. The second command is '\$ ls -a', which produces the following output: '. . .bash_logout .gnupg .profile', '. . .bash_profile .inputrc', and '.bash_history .bashrc .minttyrc'. The prompt '\$' is visible at the bottom left of the terminal window.

```
ilya.shpigor@DESKTOP-1NBVQM4 MSYS ~
$ ls

ilya.shpigor@DESKTOP-1NBVQM4 MSYS ~
$ ls -a
. . .bash_logout .gnupg .profile
. . .bash_profile .inputrc
.bash_history .bashrc .minttyrc

ilya.shpigor@DESKTOP-1NBVQM4 MSYS ~
$
```

Figure 2-9. The output of the `ls` utility

Windows has a concept of hidden files and directories. The Unix environment also has it. Applications and OS create hidden files for their own needs. These files store configuration and temporary data.



Windows Explorer does not display hidden files and directories by default. Change the [Explorer settings](#)¹⁸⁷ to see them.

You can make the file hidden in Windows by changing its attribute. If you want to do the same in Unix, you should add a dot at the beginning of the filename.

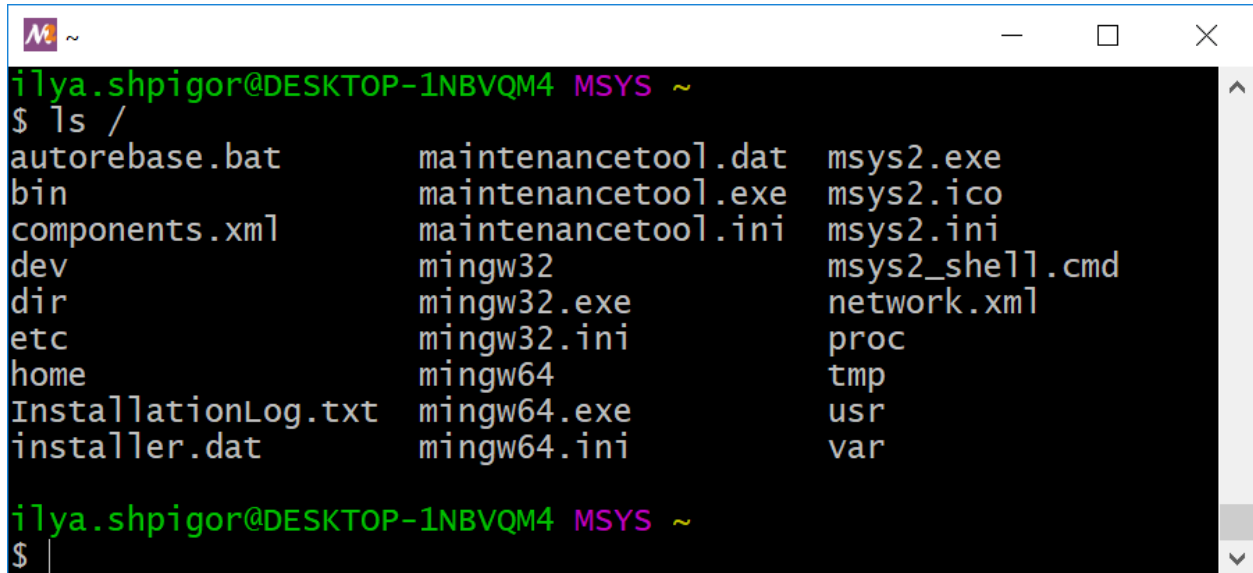
When you launch the `ls` utility without parameters, it does not show you hidden objects. You can add the `-a` option to see them. Figure 2-9 shows a result of such a call.

The `ls` utility can show the contents of the specified directory. Pass a directory's absolute or relative path to the utility. For example, the following command shows the contents of the root directory:

¹⁸⁷<https://support.microsoft.com/en-us/windows/show-hidden-files-0320fe58-0117-fd59-6851-9b7f9840fdb2>

```
ls /
```

Figure 2-10 shows the output of this command.



```

ilya.shpigor@DESKTOP-1NBVQM4 MSYS ~
$ ls /
autorebase.bat      maintenancetool.dat  msys2.exe
bin                 maintenancetool.exe  msys2.ico
components.xml      maintenancetool.ini  msys2.ini
dev                 mingw32              msys2_shell.cmd
dir                 mingw32.exe          network.xml
etc                 mingw32.ini          proc
home                mingw64              tmp
InstallationLog.txt mingw64.exe          usr
installer.dat       mingw64.ini          var

```

Figure 2-10. The output of the `ls` utility

There are no directories `/c` and `/d` in Figure 2-10. These are the mount points of C and D disk drives according to Listing 2-2. The mounting points are in the root directory. Why does not the `ls` utility print them? It happens because the Windows file system does not have a concept of mount points. Therefore, it does not have directories `/c` and `/d`. They are present in the Unix environment only. These are not real directories but paths where you can access the disk file systems. The `ls` utility reads the directory contents in the Windows file system. Thus, it does not show the mount points. The `ls` utility behaves differently in Linux and macOS. It shows mount points properly there.

mount

If your computer has several disk drives, you can read their mount points. Call the `mount`¹⁸⁸ utility without parameters for doing that. Figure 2-11 shows its output.

¹⁸⁸[https://en.wikipedia.org/wiki/Mount_\(Unix\)](https://en.wikipedia.org/wiki/Mount_(Unix))

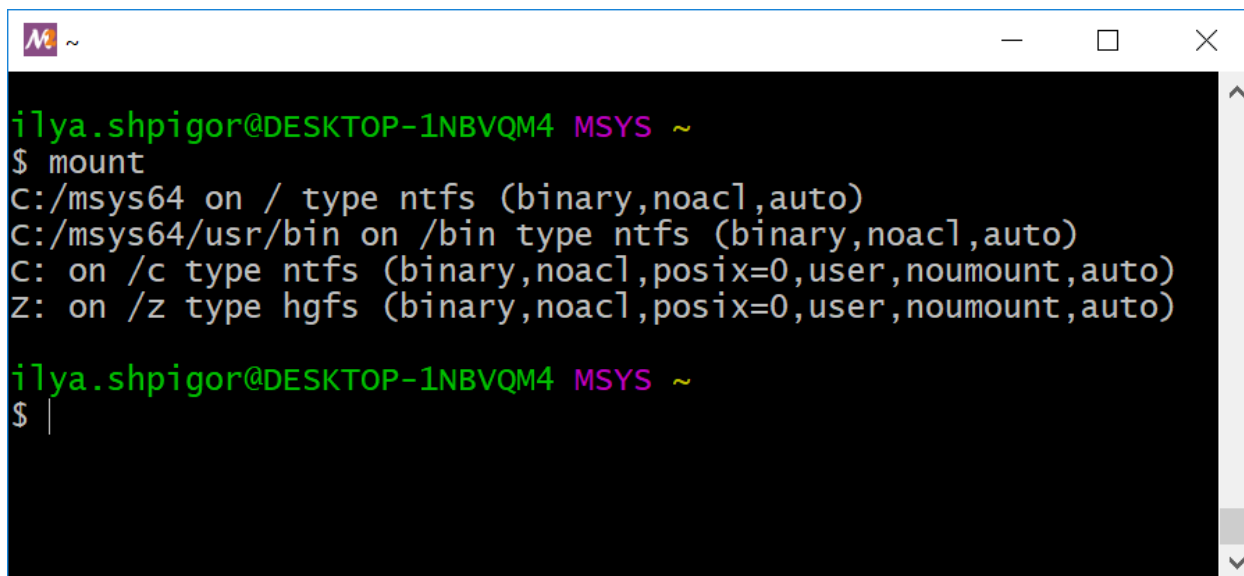


Figure 2-11. The output of the mount utility

Consider this output as a table with four columns. The columns display the following values:

1. The disk drive, its partition or directory. It is the object that the OS has mounted to the root directory.
2. Mount point. It is the path where you can access the mounted disk drive.
3. The file system type of the disk drive.
4. Mounting parameters. An example is access permissions to the disk contents.

If we split the mount utility output into these columns, we get Table 2-2.

Table 2-2. The output of the mount utility

Mounted partition	Mount point	FS type	Mounting parameters
C:/msys64	/	ntfs	binary,noacl,auto
C:/msys64/usr/bin	/bin	ntfs	binary,noacl,auto
C:	/c	ntfs	binary,noacl,posix=0,user,noumount,auto
Z:	/z	hgfs	binary,noacl,posix=0,user,noumount,auto

Table 2-2 confuses most Windows users. MSYS2 mounts C:/msys64 as the root directory. Then it mounts the C and Z disks into the root. Their mount points are /c and /z. It means that you can access the C drive via the C:/msys64/c Windows path in the Unix environment. However, C:/msys64 is the subdirectory of disk C in the Windows file system. We got a contradiction.

Actually, there is no contradiction. The /c path is the mount point that exists only in the Unix environment. It does not exist in the Windows file system. Therefore, Windows knows nothing about the C:/msys64/c path. It is just invalid if you try to open it via Explorer. You can imagine the

mount point `/c` as the [shortcut¹⁸⁹](#) to drive C that exists in the MSYS2 environment only.

The output of the `mount` utility took up a lot of screen space. You can clear the terminal window by the `Ctrl+L` keystroke.

Another useful keystroke is `Ctrl+C`. It interrupts the currently running command. Use it if the command hangs or you want to stop it.

cd

You have got everything about the current directory. Now you can change it. Suppose that you are looking for the Bash documentation. You can find it in the `/usr` system directory. Installed applications stores their non-executable files there. Call the `cd` command to go to the `/usr` path. Do it this way:

```
cd /usr
```

Do not forget about autocompletion. It works for both command and its parameters. Just type “`cd /u`” and press the `Tab` key. Bash adds the directory name `usr` automatically. Figure 2-12 shows the result of the command.

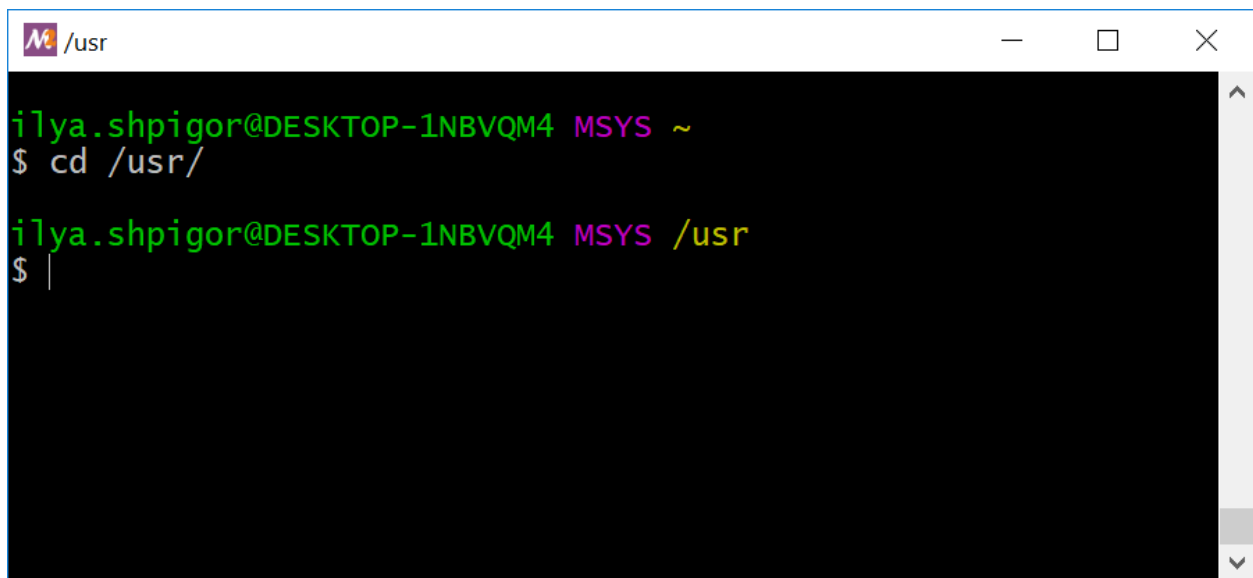


Figure 2-12. The result of the `cd` command

The `cd` command does not output anything if it succeeds. It changes the current directory and that is it. You can read the new path in the line above the command prompt. This line shows the `/usr` path after our `cd` call.

The `cd` command accepts both absolute and relative paths. Relative paths are shorter. Therefore, you type them faster. Prefer them when navigating the file system using a command shell.

¹⁸⁹[https://en.wikipedia.org/wiki/Shortcut_\(computing\)](https://en.wikipedia.org/wiki/Shortcut_(computing))

There is a simple rule to distinguish the type of path. An absolute path starts with a slash /. An example is `/c/Windows/system32`. A relative path starts with a directory name. An example is `Windows/system32`.

Now you are in the `/usr` directory. You can get a list of its subdirectories and go to one of them. Suppose that you want to go one level higher and reach the root directory. There are two ways for doing that: go to the absolute path `/` or the special relative path `..`. The `..` path always points to the parent directory of the current one. Use it in the `cd` call this way:

```
cd ..
```



In addition to `..`, there is another special path `.` (dot). It points to the current directory. If you execute the command “`cd .`”, nothing happens. You stay in the same place. You need the `.` path to run programs from the current directory.

Come back to the `/usr` directory. Then run the `ls` utility there. It will show you the `share` subdirectory. Come to this directory and call `ls` again. You will find the `doc` directory there. It contains Bash documentation. Call the `cd` command this way to reach the documentation:

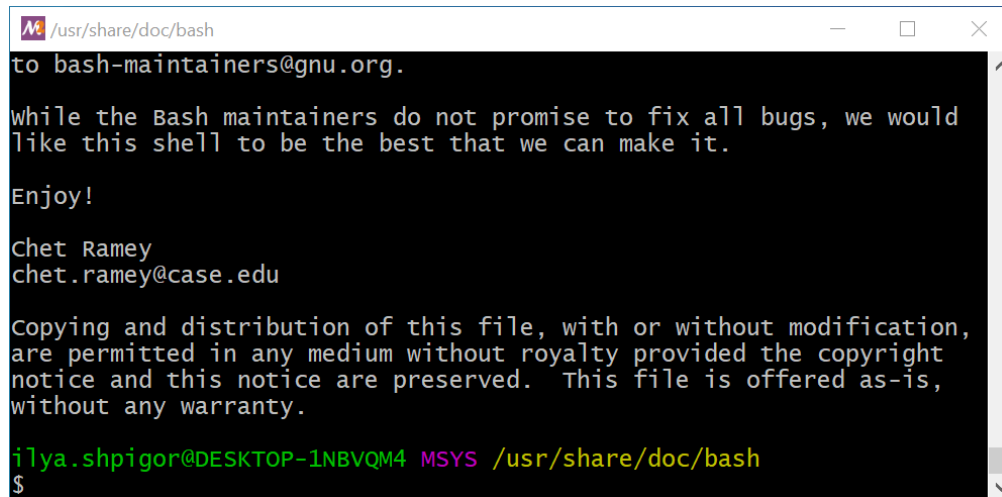
```
cd doc/bash
```

You are in the `/usr/share/doc/bash` directory now. Call the `ls` utility there. It will show you several files. One of them is `README`. It contains a brief description of the Bash interpreter.

You found the documentation file. The next step is to print its contents. The `cat` utility does that. Here is an example of how to run it:

```
cat README
```

Figure 2-13 shows the terminal window after the `cat` call.



```

/usr/share/doc/bash
to bash-maintainers@gnu.org.

while the Bash maintainers do not promise to fix all bugs, we would
like this shell to be the best that we can make it.

Enjoy!

Chet Ramey
chet.ramey@case.edu

Copying and distribution of this file, with or without modification,
are permitted in any medium without royalty provided the copyright
notice and this notice are preserved. This file is offered as-is,
without any warranty.

ilya.shpigor@DESKTOP-1NBVQM4 MSYS /usr/share/doc/bash
$

```

Figure 2-13. The result of the `cat` utility



Some Bash manuals do not recommend printing file contents using the `cat` utility. They said that the utility's purpose is to merge several files and print the result. The manuals suggest combining the `echo` command with a stream redirection. Here is an example of this approach:

```
echo "$(< README.txt)"
```

The `README` file contents do not fit in the terminal window. Therefore, you see the tail of the file in Figure 2-13. Use the scroll bar on the window's right side to check the head of the file. Also, use the `Shift+PageUp` and `Shift+PageDown` hotkeys to scroll [pages](#)¹⁹⁰ up and down. The `Shift+↑` and `Shift+↓` keystrokes scroll the lines.

Command History

Whenever you call a command, Bash saves it in the [command history](#)¹⁹¹. You can navigate the history by up and down arrow keys. Bash automatically types the corresponding command. You just need to press `Enter` for launching it. For example, you have called the “`cat README`” command. Press the up arrow and `Enter` to repeat it.

The `Ctrl+R` shortcut brings up a search over all command history. Press `Ctrl+R` and start typing. Bash will show you the last called command that begins with these characters. Press `Enter` to execute it.

The `history` command shows you the whole history. Run it without parameters this way:

¹⁹⁰https://en.wikipedia.org/wiki/Page_Up_and_Page_Down_keys

¹⁹¹https://en.wikipedia.org/wiki/Command_history

history

The history stores the command that you have executed. It does not keep the command that you typed and then erased.

There is a trick to save the command to the history without executing it. Add the hash symbol # before the command and press Enter. Bash stores the typed line, but it does not execute it. This happens because the hash symbol means comment. When the interpreter meets a comment, it ignores this line. However, Bash adds the commented lines in the history because they are legal constructions of the language.

Here is an example of the trick with comment for our `cat` utility call:

```
#cat README
```

You have saved the commented command in the history. Now you can find it there by pressing the up arrow key. Remove the hash symbol at the beginning of the line and press Enter. Bash will execute your command.

You can do the comment trick by the `Alt+Shift+3` shortcut. It works in most modern terminal emulators. Here are the steps for using the shortcut:

1. Type a command, but do not press Enter.
2. Press `Alt+Shift+3`.
3. Bash saves the command in the history without executing it.

Sometimes you need to copy text from the terminal window. It can be a command or its output. Here is an example. Suppose that some document needs a part of the Bash `README` file. Use the [clipboard](#)¹⁹² to copy it. The clipboard is temporary storage for text data. When you select something in the terminal window with a mouse, the clipboard saves it automatically. Then you can paste this data to any other window.

These are the steps to copy text from the terminal window:

1. Select the text with the mouse. Hold down the left mouse button and drag the cursor over the required text.
2. Press the middle mouse button to paste the text from the clipboard into the same or another terminal window. You insert the text at the current cursor position.
3. Right-click and select the “Paste” item to paste the text to the application other than the terminal.

¹⁹²[https://en.wikipedia.org/wiki/Clipboard_\(computing\)](https://en.wikipedia.org/wiki/Clipboard_(computing))

find

It is inconvenient to search for a file or directory with `cd` and `ls` commands. The special `find` utility does it better.

If you run the `find` utility without parameters, it traverses the contents of the current directory and prints it. The output includes hidden objects. Figure 2-14 shows the result of running `find` in the home directory.

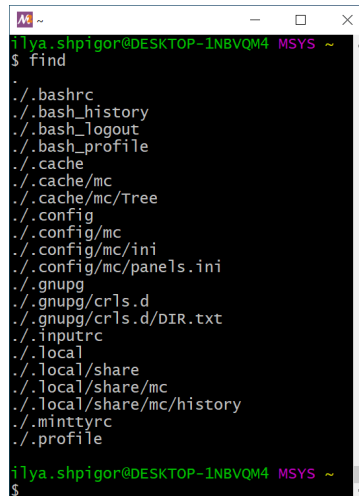
A screenshot of a terminal window titled "iIya.shpigor@DESKTOP-1NBVQM4 MSYS ~". The terminal shows the command `$ find` and its output, which lists various files and directories in the home directory, including hidden files like `./bashrc`, `./bash_history`, `./bash_logout`, `./bash_profile`, `./cache`, `./cache/mc`, `./cache/mc/Tree`, `./config`, `./config/mc`, `./config/mc/ini`, `./config/mc/panels.ini`, `./gnupg`, `./gnupg/cr1s.d`, `./gnupg/cr1s.d/DIR.txt`, `./inputrc`, `./local`, `./local/share`, `./local/share/mc`, `./local/share/mc/history`, `./minttyrc`, and `./profile`. The prompt `$` is visible at the bottom of the terminal.

Figure 2-14. The output of the `find` utility

The first parameter of `find` is the directory to search in. The utility accepts relative and absolute paths. For example, the following command shows the contents of the root directory:

```
find /
```

You can specify search conditions starting from the second parameter. If the found object does not meet these conditions, `find` does not print it. The conditions form a single expression. The utility has an embedded interpreter that processes this expression.

An example of the `find` condition is the specific filename. When you call the utility with such a condition, it prints the found files with this name only.

Table 2-3 shows the format of commonly used conditions for the `find` utility.

Table 2-3. Commonly used conditions for the `find` utility

Condition	Meaning	Example
<code>-type f</code>	Search files only.	<code>find -type f</code>
<code>-type d</code>	Search directories only.	<code>find -type d</code>
<code>-name <pattern></code>	Search for a file or directory with the name that matches a glob pattern ¹⁹³ . The pattern is case-sensitive.	<code>find -name README</code> <code>find -name READ*</code> <code>find -name READ??</code>
<code>-iname <pattern></code>	Search for a file or directory with the name that matches a glob pattern. The pattern is case-insensitive.	<code>find -iname readme</code>
<code>-path <pattern></code>	Search for a file or directory with the path that matches a glob pattern. The pattern is case-sensitive.	<code>find -path */doc/bash/*</code>
<code>-ipath <pattern></code>	Search for a file or directory with the path that matches a glob pattern. The pattern is case-insensitive.	<code>find . -ipath */DOC/BASH/*</code>
<code>-a</code> or <code>-and</code>	Combine several conditions using the logical AND. If the found object fits all conditions, the utility prints it.	<code>find -name README -a -path */doc/bash/*</code>
<code>-o</code> or <code>-or</code>	Combine several conditions using the logical OR. If the found object fits at least one condition, the utility prints it.	<code>find -name README -o -path */doc/bash/*</code>
<code>!</code> or <code>-not</code>	The logical negation (NOT) of the condition. If the found object does not fit the condition, the utility prints it.	<code>find -not -name README</code> <code>find ! -name README</code>

A glob pattern is a search query that contains **wildcard characters**¹⁹⁴. Bash allows three wildcard characters: `*`, `?` and `[`. The asterisk stands for any number of any characters. A question mark means a single character of any kind.

Here is an example of glob patterns. The string `README` matches all following patterns:

- `*ME`
- `README?`
- `*M?`
- `R*M?`

¹⁹³[https://en.wikipedia.org/wiki/Glob_\(programming\)](https://en.wikipedia.org/wiki/Glob_(programming))

¹⁹⁴https://en.wikipedia.org/wiki/Wildcard_character

Square brackets indicate a set of characters at a specific position. For example, the pattern “[cb]at.txt” matches the cat.txt and bat.txt files. You can apply this pattern to the find call this way:

```
find . -name "[cb]at.txt"
```

Exercise 2-1. Glob patterns

What of the following lines corresponds to the pattern “*ME.??” ?

- * 00_README.txt
 - * README
 - * README.md
-

Exercise 2-2. Glob patterns

What of the following lines corresponds to the pattern “*/doc?openssl*” ?

- * /usr/share/doc/openssl/IPAddressChoice_new.html
 - * /usr/share/doc_openssl/IPAddressChoice_new.html
 - * doc/openssl
 - * /doc/openssl
-

Let’s apply glob patterns into practice. Suppose that you do not know the Bash README file location and looking for it. Then you should use the find utility.

Start searching with the utility from the root directory. Now you need a search condition. It is a common practice to store documentation in directories called doc in Unix. Therefore, you can search files in these directories only. This way, you get the following find call:

```
find / -path */doc/*
```

The command shows you all documentation files on all mounted disks. This is a huge list. You can shorten it with an extra search condition. It should be a separate directory for the Bash documentation. The directory is called bash. Add this path as the second search condition. Then you get the following command:

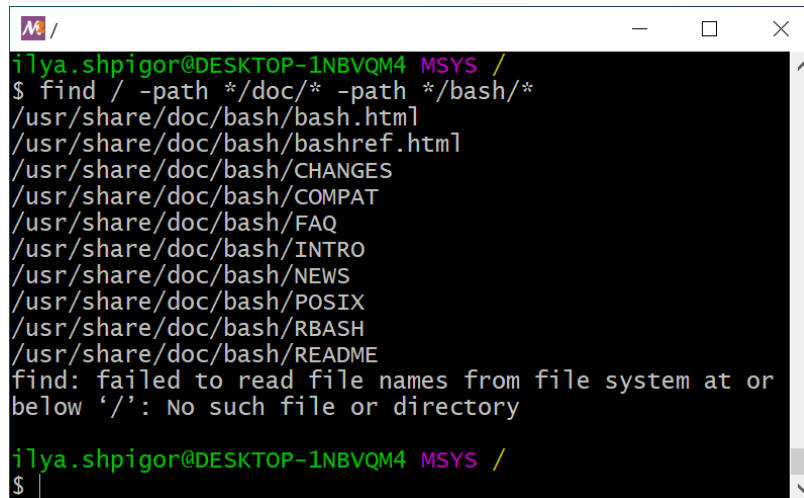
```
find / -path */doc/* -path */bash/*
```

Figure 2-15 shows the result of this command.

The following find call provides the same result:

```
find / -path */doc/* -a -path */bash/*
```

Our `find` calls differ by the `-a` option between conditions. The option means logical AND. If you do not specify any logical operator between conditions, `find` inserts AND by default. This is a reason why both calls provide the same result.



```
ilya.shpigor@DESKTOP-1NBVQM4 MSYS /
$ find / -path */doc/* -path */bash/*
/usr/share/doc/bash/bash.html
/usr/share/doc/bash/bashref.html
/usr/share/doc/bash/CHANGES
/usr/share/doc/bash/COMPAT
/usr/share/doc/bash/FAQ
/usr/share/doc/bash/INTRO
/usr/share/doc/bash/NEWS
/usr/share/doc/bash/POSIX
/usr/share/doc/bash/RBASH
/usr/share/doc/bash/README
find: failed to read file names from file system at or
below '/': No such file or directory
ilya.shpigor@DESKTOP-1NBVQM4 MSYS /
$
```

Figure 2-15. The output of the `find` utility

You can see that the `find` utility reports an error in Figure 2-15. The mount points of Windows disk drives cause it. The utility cannot access them when you start searching from the root directory. You can avoid the problem if you start searching from the `/c` mount point. Do it this way:

```
find /c -path */doc/* -a -path */bash/*
```

There is an alternative solution. You should exclude mount points from the search. The `-mount` option does this. Apply the option this way:

```
find / -mount -path */doc/* -a -path */bash/*
```

When you add the second search condition, the `find` utility shows a short list of documents. You can find the right `README` file easily there.

There are other ways to search for the documentation file. Suppose that you know its name. Then you can specify it together with an assumed path. You will get the `find` call like this:

```
find / -path */doc/* -name README
```

Figure 2-16 shows the result of this command.

```

ilya.shpigor@DESKTOP-1NBVQM4 MSYS /
$ find / -path */doc/* -name README
/usr/share/doc/bash/README
/usr/share/doc/gnupg/examples/README
/usr/share/doc/gnupg/examples/systemd-user/README
/usr/share/doc/gnupg/README
/usr/share/doc/libargp/README
/usr/share/doc/pcre/README
/usr/share/doc/readline/README
/usr/share/doc/rebase/README
/usr/share/doc/xz/README
find: failed to read file names from file system at or
below '/': No such file or directory
ilya.shpigor@DESKTOP-1NBVQM4 MSYS /
$

```

Figure 2-16. The output of the `find` utility

Again you got a short list of files. It is easy to locate the right file there.

You can group the conditions of the `find` utility. Do it using the [escaped¹⁹⁵](#) parentheses. Here is an example of using them. Let's write the `find` call that searches README files with path `*/doc/*` or LICENSE files with an arbitrary path. This call looks like this:

```
find / \( -path */doc/* -name README \) -o -name LICENSE
```

Why should you apply backslashes to escape brackets here? The parentheses are part of the Bash syntax. Therefore, Bash treats them like language constructs. When Bash meets parentheses in a command, it performs an **expansion**. The expansion is the replacement of a part of the command with something else. When you escape parentheses, you force Bash to ignore them. Thus, Bash does not perform the expansion and passes all search conditions to the `find` utility as it is.

The `find` utility can process the found objects. You can specify an action to apply as an extra option. The utility will apply this action to each found object.

Table 2-4 shows the `find` options that specify actions.

Table 2-4. Options for specifying actions on found objects

Option	Meaning	Example
<code>-exec command {} \;</code>	Execute the specified command on each found object.	<code>find -name README -type f -exec cp {} ~ \;</code>
<code>-exec command {} +</code>	Execute the specified command once over all found objects. The command receives all these objects on the input.	<code>find -type d -exec cp -t ~ {} +</code>
<code>-delete</code>	Delete each of the found files. The utility deletes empty directories only.	<code>find -name README -type f -delete</code>

¹⁹⁵https://en.wikipedia.org/wiki/Escape_character

Table 2-4 shows that there are two variants of the `-exec` action. They differ by the last symbol. It can be an escaped semicolon `\;` or a plus sign `+`. Use the plus sign only if the called command handles several input parameters. You will make a mistake if the command accepts one parameter only. It will process the first found object and skip the rest.

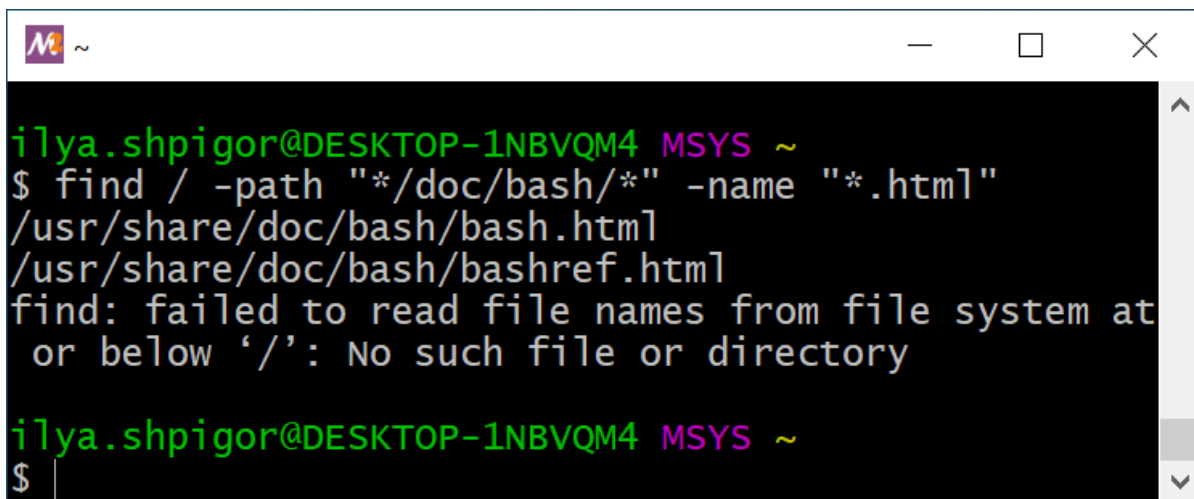
Let's apply the `-exec` action in practice. Suppose that you want to copy files with the Bash documentation into the home directory. You are interested in the HTML files only.

The first step is preparing the correct `find` call for searching the files. You should apply two conditions here. The first one checks the directory of the Bash documentation. The second condition checks the file extensions. If you combine these conditions, you get the following `find` call:

```
find / -path "*/doc/bash/*" -name "*.html"
```

When you pass the glob pattern to the `find` utility, always enclose it in double quotes. The quotes do the same as the backslash before parentheses. They prevent Bash from expanding the patterns. Instead, Bash passes them to the `find` utility.

Figure 2-17 shows the result of our `find` call. You can see that it found HTML files correctly.



```
ilya.shpigor@DESKTOP-1NBVQM4 MSYS ~
$ find / -path "*/doc/bash/*" -name "*.html"
/usr/share/doc/bash/bash.html
/usr/share/doc/bash/bashref.html
find: failed to read file names from file system at
or below '/': No such file or directory
ilya.shpigor@DESKTOP-1NBVQM4 MSYS ~
$
```

Figure 2-17. The output of the `find` utility

The second step for solving your task is adding the `-exec` action. The action should call the `cp` utility. This utility copies files and directories to the specified path. It takes two parameters. The first one is the source object to copy. The second parameter is the target path. When you apply the `-exec` action, you get the following `find` call:

```
find / -path "*/doc/bash/*" -name "*.html" -exec cp {} ~ \;
```

Run this command. It prints an error about the mount point. Despite the error, the command did its job. It copied the HTML files into the home directory.

How does the command work in detail? It calls the `cp` utility for each HTML file it found. When calling the utility, `find` inserts each found object instead of curly braces `{}`. Therefore, two `cp` calls happen here. They look like this:

```
1 cp ./usr/share/doc/bash/bash.html ~
2 cp ./usr/share/doc/bash/bashref.html ~
```

Each `cp` call copies one HTML file to the home directory.

Good job! You just wrote your first program in the language of the `find` utility. The program works according to the following algorithm:

1. Find HTML files starting from the root directory. Their paths match the `*/doc/bash/*` pattern.
2. Copy each found file into the home directory.

The program is quite simple and consists of two steps only. However, it is a scalable solution for finding and copying files. The program processes two or dozens of HTML files with the same speed.

You can combine the `-exec` actions in the same way as the search conditions. For example, let's print the contents of each found HTML file and count the number of its lines. You should call the `cat` utility to print the file contents. The `wc` utility counts the lines. It takes the filename as an input parameter. If you combine `cat` and `wc` calls, you get the following `find` command:

```
find / -path "*/doc/bash/*" -name "*.html" -exec cat {} \; -exec wc -l {} \;
```

There is no logical operation between the `-exec` actions. The `find` utility inserts logical AND by default. This has a consequence in our case. If the `cat` utility fails, `find` does not call the `wc` utility. It means that `find` executes the second action only if the first one succeeds. You can apply the logical OR explicitly. Then `find` always calls `wc`. Here is the command with logical OR:

```
find / -path "*/doc/bash/*" -name "*.html" -exec cat {} \; -o -exec wc -l {} \;
```

You can group the `-exec` actions with escaped parentheses `\(` and `\)`. It works the same way as grouping search conditions.

Exercise 2-3. Searching for files with the `find` utility

Write a `find` call to search for text files in a Unix environment.

Extend the command to print the total number of lines in these files.

Boolean Expressions

The search conditions of the `find` utility are **Boolean expressions**¹⁹⁶. A Boolean expression is a programming language statement. It produces a Boolean value when evaluated. This value equals either “true” or “false”.

The `find` condition is a statement of the utility’s language. It produces the “true” value if the found object meets its requirement. Otherwise, the condition produces “false”. If there are several conditions in the `find` call, they make a single compound Boolean expression.

When we have considered the binary numeral system, we already met Boolean algebra. This section of mathematics studies **logical operators**¹⁹⁷. They differ from the arithmetic operations: addition, subtraction, multiplication, and division.

You can apply a logical operator to Boolean values or expressions. Using an arithmetic operation does not make sense in this case. Addition or subtraction is trivial for Boolean values. It yields nothing. When you apply a logical operator, you get a condition with strict evaluation rules. This way, you wrote search conditions for the `find` utility. When you combine several conditions, you get a program with complex behavior.

An **operand** is an object of a logical operator. Boolean values and expressions can be operands.

Let’s consider Boolean expressions using an example. The example is not related to the `find` utility or Bash for simplicity. Imagine that you are programming a robot for a warehouse. Its job is to move boxes from point A to point B. You can write the following straightforward algorithm for the robot:

1. Move to point A.
2. Pick up the box at point A.
3. Move to point B.
4. Put the box at point B.

This algorithm does not have any conditions. It means that the robot performs each step independently of external events.

Now imagine that an obstacle happens in the robot’s way. For example, another robot stuck there. Executing your algorithm leads to the collision of the robots in this case. You should add a condition in the algorithm to prevent the collision. For example, it can look like this:

1. Move to point A.
2. Pick up the box at point A.
3. If there is no obstacle, move to point B. Otherwise, stop.
4. Put the box at point B.

¹⁹⁶https://en.wikipedia.org/wiki/Boolean_expression

¹⁹⁷https://en.wikipedia.org/wiki/Logical_connective

The third step of the algorithm is called **conditional statement**¹⁹⁸. All modern programming languages have such a statement.

The conditional statement works according to the following algorithm:

1. Evaluate the Boolean expression in the condition.
2. If the expression produces “true”, perform the first action.
3. If the expression produces “false”, perform the second action.

The robot evaluates the value of the Boolean expression “there is no obstacle” in our example. If there is an obstacle, the expression produces “false” and the robot stops. Otherwise, the robot moves to point B.

When writing the conditional statement, you can combine several Boolean expressions using logical operators. Here is an example. Suppose that the robot tries to pick up a box at point A, but there is no box. Then there is no reason for him to move to point B. You can check this situation in the conditional statement. Add the new Boolean expression there using **logical AND**¹⁹⁹ (conjunction). Then the robot’s algorithm becomes like this:

1. Move to point A.
2. Pick up the box at point A.
3. If there is a box AND no obstacle, move to point B. Otherwise, stop.
4. Put the box at point B.

Logical operators produce Boolean values when evaluated. The result of a logical AND equals “true” when both operands are “true”. In our example, it happens when the robot has a box and there is no obstacle on its way. Otherwise, the result of logical AND equals “false”. It forces the robot to stop.

You have used two more logical operators when learning the `find` utility. These operators are **OR**²⁰⁰ (disjunction) and **NOT**²⁰¹ (negation).

Actually, you have already applied logical NOT in the robot’s algorithm. It stays implicitly in the expression “there is no obstacle”. It equals the following negation: “there is NOT an obstacle”. You can specify the logical NOT in the algorithm explicitly this way:

1. Move to point A.
2. Pick up the box at point A.
3. If there is a box AND there is NOT an obstacle, move to point B. Otherwise, stop.
4. Put the box at point B.

¹⁹⁸[https://en.wikipedia.org/wiki/Conditional_\(computer_programming\)](https://en.wikipedia.org/wiki/Conditional_(computer_programming))

¹⁹⁹https://en.wikipedia.org/wiki/Logical_conjunction

²⁰⁰https://en.wikipedia.org/wiki/Logical_disjunction

²⁰¹<https://en.wikipedia.org/wiki/Negation>

You can always replace logical AND by OR with some extra changes. Let's do it for our example but keep the robot's behavior the same. You should add the negation to the first Boolean expression and remove it from the second one. Also, you have to change the order of actions in the conditional statement. If the condition produces "true", the robot stops. If it produces "false", the robot moves to point B. The new algorithm looks this way:

1. Move to point A.
2. Pick up the box at point A.
3. If there is NOT a box OR there is an obstacle, stop. Otherwise, move to point B.
4. Put the box at point B.

Read the new conditional statement carefully. The robot follows the same decisions as before. It stops if it has no box or if there is an obstacle on its way. However, you have exchanged the logical AND to OR. This trick helps you to keep your conditional statements clear. Choose between logical AND and OR depending on your Boolean expressions. Pick one that fits your case better.

You wrote the Boolean expressions as sentences in English in our example. Such a sentence sounds unnatural. You have to read it several times to understand it. This happens because the natural humans' language is not suitable for writing Boolean expressions. This language is not accurate enough. Boolean algebra uses mathematical notation for that reason.

We have considered logical AND, OR and NOT. You will deal with three more operators in programming often:

- Equivalence
- Non-equivalence
- Exclusive OR

Table 2-5 explains them.

Table 2-5. Logical operators

Operator	Evaluation Rule
AND	It produces "true" when both operands are "true".
OR	It produces "true" when any of the operands is "true". It produces "false" when all operands are "false".
NOT	It produces "true" when the operand is "false" and vice versa.
Exclusive OR (XOR)	It produces "true" when the operands have different values (true-false or false-true). It produces "false" when the operands are the same (true-true, false-false).
Equivalence	It produces "true" when the operands have the same values.
Non-equivalence	It produces "true" when the values of the operands differ.

Try to memorize this table. It is simple to reach when you use logical operators often.

grep

The GNU utilities have one more searching tool besides `find`. It is called `grep`. This utility checks file contents when searching.

How to choose the proper utility for searching? Use `find` for searching a file or directory by its name, path or [metadata](https://en.wikipedia.org/wiki/Metadata)²⁰². Metadata is extra information about an object. Examples of the file metadata are size, time of creation and last modification, permissions. Use the `grep` utility to find a file when you know nothing about it except its contents.

Here is an example. It shows you how to choose the right utility for searching. Suppose that you are looking for a documentation file. You know that it contains the phrase “free software”. If you apply the `find` utility, the searching algorithm looks like this:

1. Call `find` to list all the files with the `README` name.
2. Open each file in a text editor and check if it has the phrase “free software”.

Using a text editor for checking dozens of files takes too much effort and time. You should perform several operations with each file manually: open it, activate the editor’s searching mode, type the “free software” phrase. The `grep` utility automates this task. For example, the following command finds all lines with the “free software” phrase in the specified `README` file:

```
grep "free software" /usr/share/doc/bash/README
```

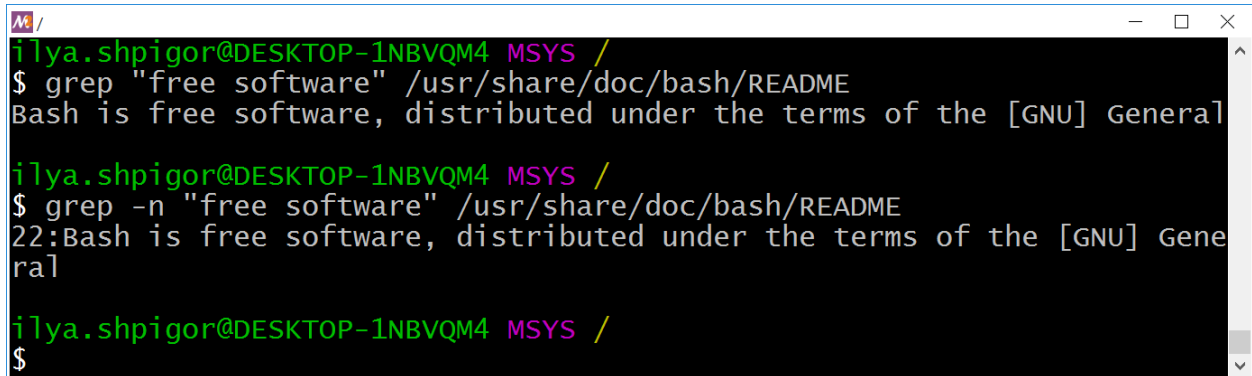
The first parameter of the utility is a string for searching. Always enclose it in double quotes. This way, you prevent Bash expansions and guarantee that the utility receives the string unchanged. Without the quotes, Bash splits the phrase into two separate parameters. This mechanism of splitting strings into words is called [word splitting](http://mywiki.woledge.org/WordSplitting)²⁰³.

The second parameter of `grep` is a relative or absolute path to the file. If you specify a list of files separated by spaces, the utility processes them all. In the example, we passed the `README` file path only.

Figure 2-18 shows the result of the `grep` call.

²⁰²<https://en.wikipedia.org/wiki/Metadata>

²⁰³<http://mywiki.woledge.org/WordSplitting>



```
ilya.shpigor@DESKTOP-1NBVQM4 MSYS /
$ grep "free software" /usr/share/doc/bash/README
Bash is free software, distributed under the terms of the [GNU] General

ilya.shpigor@DESKTOP-1NBVQM4 MSYS /
$ grep -n "free software" /usr/share/doc/bash/README
22:Bash is free software, distributed under the terms of the [GNU] General

ilya.shpigor@DESKTOP-1NBVQM4 MSYS /
$
```

Figure 2-18. The output of the `grep` utility

You see all lines of the file where the utility found the specified phrase. The `-n` option adds the line numbers to the `grep` output. It can help you to check big text files. Add the option before the first parameter when calling the utility. Figure 2-18 shows the output in this case.

We have learned how to use `grep` to find a string in the specified files. Now let's apply the utility to solve our task. You are looking for the documentation files with the phrase "free software". There are two ways to find them with the `grep` utility:

- Use Bash glob patterns.
- Use the file search mechanism of the `grep` utility.

The first method works well when you have all files for checking in the same directory. Suppose that you found two `README` files: one for Bash and one for the `xz` utility. You have copied them to the home directory with the names `bash.txt` and `xz.txt`. The following two commands find the file that contains the phrase "free software":

```
1 cd ~
2 grep "free software" *
```

The first command changes the current directory to the user's home. The second command calls the `grep` utility.

When calling `grep`, we have specified the asterisk for the target file path. This wildcard means any string. Bash expands all wildcards in the command before launching it. In our example, Bash replaces the asterisk with all files of the home directory. The resulting `grep` call looks like this:

```
grep "free software" bash.txt xz.txt
```

Launch both versions of the `grep` call: with the `*` pattern and with a list of two files. The utility prints the same result for both cases.

You can search for the phrase in a single command. Just exclude the `cd` call. Then add the home directory to the search pattern. You will get the following `grep` call:

```
grep "free software" ~/*
```

This command does not handle subdirectories. It means that the `grep` call does not check the files in the `~/tmp` directory, for example.

There is an option to check how the Bash expands a glob pattern. Use the `echo` command for that. Here are `echo` calls for checking our patterns:

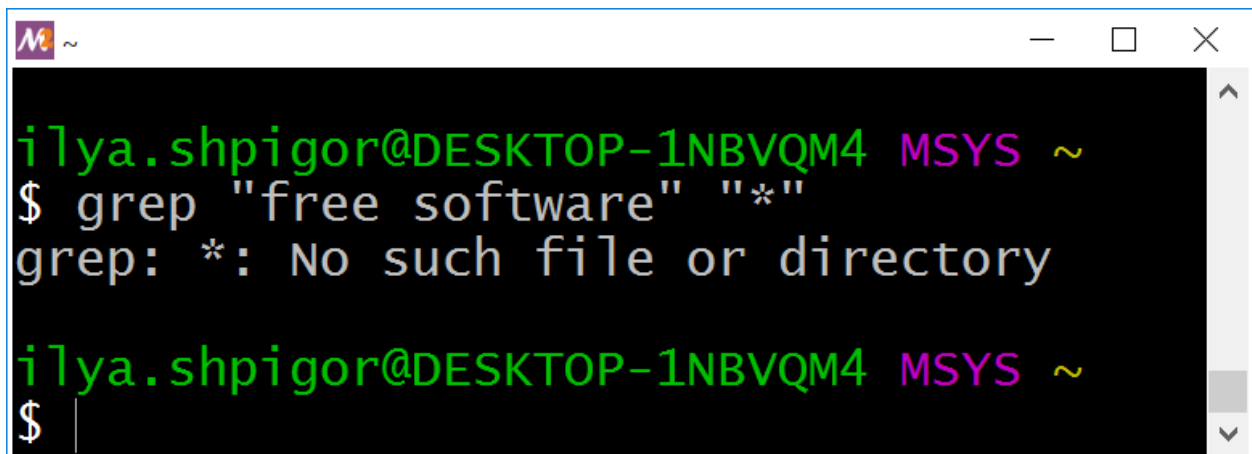
```
1 echo *
2 echo ~/*
```

Run these commands. The first one lists files and subdirectories of the current directory. The second command does the same for the home directory.

Do not enclose search patterns in double quotes. Here is an example of the wrong command:

```
grep "free software" "*"
```

Quotes prevent the Bash expansion. Therefore, Bash does not insert the filenames to the command but passes the asterisk to the `grep` utility. The utility cannot handle the glob pattern properly as `find` does. Thus, you will get an error like Figure 2-19 shows.

A terminal window screenshot showing a command prompt. The prompt is `ilya.shpigor@DESKTOP-1NBVQM4 MSYS ~`. The user enters `$ grep "free software" "*"`. The output is `grep: *: No such file or directory`. The prompt repeats, and the user enters `$` at the end of the line.

```
ilya.shpigor@DESKTOP-1NBVQM4 MSYS ~
$ grep "free software" "*"
grep: *: No such file or directory
ilya.shpigor@DESKTOP-1NBVQM4 MSYS ~
$
```

Figure 2-19. The result of processing a search pattern by `grep`

When expanding the `*` pattern, Bash ignores hidden files and directories. Therefore, the `grep` utility ignores them too in our example. Add the dot before the asterisk to get the glob pattern for hidden objects. It looks like `.*`. If you want to check all files at once, specify two patterns separated by the space. Here is an example `grep` call:

```
grep "free software" * .*
```

The second approach to search files with `grep` is using its built-in mechanism. It traverses the directories recursively and checks all files there. The `-r` option enables this mechanism. When using this option, specify the search directory in the second utility's parameter.

Here is an example of using the `-r` option:

```
grep -r "free software" .
```

This command finds the “free software” phrase in the files of the current directory. It processes the hidden objects too.

If you work on Linux or macOS, prefer the `-R` option instead of `-r`. It forces `grep` to follow [symbolic links](#)²⁰⁴ when searching. Here is an example:

```
grep -R "free software" .
```



A symbolic link is a file of a special type. Instead of data, it contains a pointer to another file or directory.

You can specify the starting directory for searching by a relative or absolute path. Here are the examples for both cases:

- 1 `grep -R "free software" ilya.shpigor/tmp`
- 2 `grep -R "free software" /home/ilya.shpigor/tmp`

Suppose that you are interested in a list of files that contain a phrase. You do not need all occurrences of the phrase in each file. The `-l` option switches the `grep` utility in the mode you need. Here is an example of using it:

```
grep -Rl "free software" .
```

Figure 2-20 shows the result of this command.

²⁰⁴https://en.wikipedia.org/wiki/Symbolic_link

```

ilya.shpigor@DESKTOP-1NBVQM4 MSYS ~
$ grep -RL "free software" .
./bash.txt
ilya.shpigor@DESKTOP-1NBVQM4 MSYS ~
$

```

Figure 2-20. The `grep` outputs filenames only

You see a list of files where the phrase “free software” occurs at least once. Suppose that you need the opposite result: a list of files without the phrase. Use the `-L` option for finding them. Here is an example:

```
grep -RL "free software" .
```

The `grep` utility processes the text files only. Therefore, it deals well with the source code files. You can use the utility as an add-on to your code editor or IDE.

You may have liked the `grep` utility. You want to process PDF²⁰⁵ and MS Office documents with it. Unfortunately, this approach does not work. The contents of these files are not text. It is encoded. You need another utility to process such files. Table 2-6 shows `grep` alternatives for non-text files.

Table 2-6. Utilities for text searching in PDF and MS Office files

Utility	Features
<code>pdftotext</code> ²⁰⁶	It converts a PDF file into text format.
<code>pdfgrep</code> ²⁰⁷	It searches PDF files by their contents.
<code>antiword</code> ²⁰⁸	It converts an MS Office document into text format.
<code>catdoc</code> ²⁰⁹	It converts an MS Office document into text format.
<code>xdoc2txt</code> ²¹⁰	It converts PDF and MS Office files into text format.

Some of these utilities are available in the MSYS2 environment. Use the `pacman` package manager

²⁰⁵<https://en.wikipedia.org/wiki/PDF>

²⁰⁶<http://www.xpdfreader.com>

²⁰⁷<https://pdfgrep.org>

²⁰⁸<http://www.winfield.demon.nl>

²⁰⁹<https://www.wagner.pp.ru/~vitus/software/catdoc>

²¹⁰https://documentation.help/xdoc2txt/xdoc2txt_en.html

for installing them. The last chapter of the book describes how to use it.

Exercise 2-4. Searching for files with the `grep` utility

Write a `grep` call to find system utilities with a free license.

Here are widespread licenses for open-source software:

1. GNU General Public License
 2. MIT license
 3. Apache license
 4. BSD license
-

Command Information

We got acquainted with commands for navigating the file system. Each command has several options and parameters. We have covered the most common ones only. What if you need a rare feature that is missing in this book? You would need official documentation in this case.

All modern OSes and applications have documentation. However, you rarely need it when using the graphical interface. It happens because graphic elements are self-explanatory in most cases. Therefore, most PC users do not care about documentation.

When working with the CLI, the only way to know about available features of the software is by reading documentation. Besides that, you do not have anything that gives you a quick hint. When using CLI utility, it is crucial to know its basics. The negligence can lead to loss or corruption of your data.

The first versions of Unix had paper documentation. Using it was inconvenient and time-consuming. Soon it became even worse because the documentation volume grew rapidly. It exceeded the size of a single book. The Unix developers introduced the system called `man page`²¹¹ to solve the issue with documentation. Using this software, you can quickly find the required topic. It contains information about OS features and all installed applications.

The `man page` system is a centralized place to access documentation. Besides it, every program in Unix provides brief information about itself. For example, the Bash interpreter has its own documentation system. It is called `help`.

Suppose that you want to get a list of all Bash built-ins. Launch the `help` command without parameters. Figure 2-21 shows its output.

²¹¹https://en.wikipedia.org/wiki/Man_page

```

ilya.shpigor@DESKTOP-1NBVQM4 MSYS ~
$ help
GNU bash, version 4.4.23(1)-release (x86_64-pc-msys)
These shell commands are defined internally.  Type `help' to see this list.
Type `help name' to find out more about the function `name'.
Use `info bash' to find out more about the shell in general.
Use `man -k' or `info' to find out more about commands not in this list.

A star (*) next to a name means that the command is disabled.

job_spec [&]                history [-c] [-d offset] [n] or hist>
(( expression ))           if COMMANDS; then COMMANDS; [ elif C>
. filename [arguments]    jobs [-lnprs] [jobspec ...] or jobs >
:                          kill [-s sigspec | -n signum | -sigs>
[ arg... ]                let arg [arg ...]
[[ expression ]]          local [option] name[=value] ...
alias [-p] [name[=value] ... ]  logout [n]
bg [job_spec ...]         mapfile [-d delim] [-n count] [-O or>
bind [-lpsvPSVX] [-m keymap] [-f file>
break [n]                 popd [-n] [+N | -N]
builtin [shell-builtin] [arg ...]  printf [-v var] format [arguments]
caller [expr]             pushd [-n] [+N | -N | dir]
case WORD in [PATTERN [ | PATTERN]...>
cd [-L|[-P [-e]] [-@]] [dir]      pwd [-LPW]
command [-pvv] command [arg ...]  read [-ers] [-a array] [-d delim] [->
compgen [-abcdefgjkusv] [-o option] [>
complete [-abcdefgjkusv] [-pr] [-DE] >
compopt [-o|+o option] [-DE] [name ..>
continue [n]              select NAME [in WORDS ... ;] do COMM>
coproc [NAME] command [redirections]
declare [-aAfFgIlNrtux] [-p] [name[=v>
set [-c|pv] [+N] [-N]
dirs [-c|pv] [+N] [-N]
disown [-h] [-ar] [jobspec ... | pid >
echo [-neE] [arg ...]
enable [-a] [-dnps] [-f filename] [na>
eval [arg ...]            suspend [-f]
                                test [expr]
                                time [-p] pipeline
                                times
                                trap [-lp] [[arg] signal_spec ...]

```

Figure 2-21. The output of the help command

You see a list of all commands that Bash executes on its own. If some command is missing in this list, Bash calls a GNU utility or another program to execute it.

Here is an example. The `cd` command presents in the `help` list. It means that Bash executes it without calling another program. Now suppose you type the `find` command. It is missing in the `help` list. Therefore, Bash looks for an executable file with the `find` name on the disk drive. If it succeeds, Bash launches this file.

Where does Bash look for files that execute your commands? Bash has a list of paths where it searches utilities and programs. The **environment variable** called `PATH` stores this list. The **variable**²¹² is a named area of memory. If you write a program in machine code and want to access the memory area, you should specify its address. A variable is a mechanism of a programming language. It allows you to use the variable name instead of the memory address. Therefore, you do not need to remember addresses, which are long numbers.

Bash stores about a hundred environment variables. They hold data that affect the interpreter's behavior. Most of these data are system settings. We will consider Bash variables in the next chapter.



Call the `env` utility without parameters. It shows you all the environment variables that Bash uses at the moment.

²¹²[https://en.wikipedia.org/wiki/Variable_\(computer_science\)](https://en.wikipedia.org/wiki/Variable_(computer_science))

You can imagine the variable as a value that has a name. For example, you can say: “The time now is 12 hours”. “Time now” is the variable name. Its value equals “12 hours”. The computer stores it in memory at some address. You do not know the address. However, you can ask a computer the value of the “time now” variable. It returns you “12 hours”. This is how the variables work.

The `echo` command prints strings. It can also show you the value of a variable. For example, the following `echo` call prints the `PATH` variable:

```
echo "$PATH"
```

Why do we need the dollar sign `$` before the variable name? The `echo` command receives the string on input and outputs it. For example, this `echo` call prints the text “123”:

```
echo 123
```

The dollar sign before a word tells Bash that it is a variable name. The interpreter handles it differently than a regular word. When Bash encounters a variable name in a command, it checks its variable list. If the name presents there, Bash inserts the variable value into the command. Otherwise, the interpreter places an empty string there.



Enclosing variable names in double quotes is a [good practice](#)²¹³. This way, you avoid potential errors. There is an example. Bash replaces the variable name by its value. The value contains [control characters](#)²¹⁴. The interpreter handles these characters. As a result, the inserted variable’s value differs from one that is stored in memory. This effect leads to incorrect behavior of the program. Double quotes prevent Bash from processing strings.

Let’s come back to the `echo` command that prints the `PATH` variable. Figure 2-22 shows this output.

```
iIya.shpigor@DESKTOP-1NBVQM4 MSYS ~
$ echo "$PATH"
/usr/local/bin:/usr/bin:/bin:/opt/bin:/c/windows/system32:/c/windows:/c/windows/system32/wbem:/c/windows/system32/windowspowershell/v1.0/
iIya.shpigor@DESKTOP-1NBVQM4 MSYS ~
$
```

Figure 2-22. The value of the `PATH` variable

What does this line mean? It is a list of paths separated by colons. If you write each path on a new line, you get the following list:

²¹³<https://www.tldp.org/LDP/abs/html/quotingvar.html>

²¹⁴https://en.wikipedia.org/wiki/Control_character

```
/usr/local/bin
/usr/bin
/bin
/opt/bin
/c/Windows/System32
/c/Windows
/c/Windows/System32/Wbem
/c/Windows/System32/WindowsPowerShell/v1.0/
```

The format of the PATH variable raises questions. Why does Bash use colons as delimiters instead of [line breaks](#)²¹⁵? Line breaks make it easy to read the list. The reason is the specific behavior of Bash and some utilities when handling line breaks. Colons allow developers to avoid potential problems.

Suppose that you want to locate an executable file of some program on the disk. The PATH variable gives you a hint of where to look. Then you can apply the `find` utility and locate the file. For example, the following command searches the executable of the `find` utility:

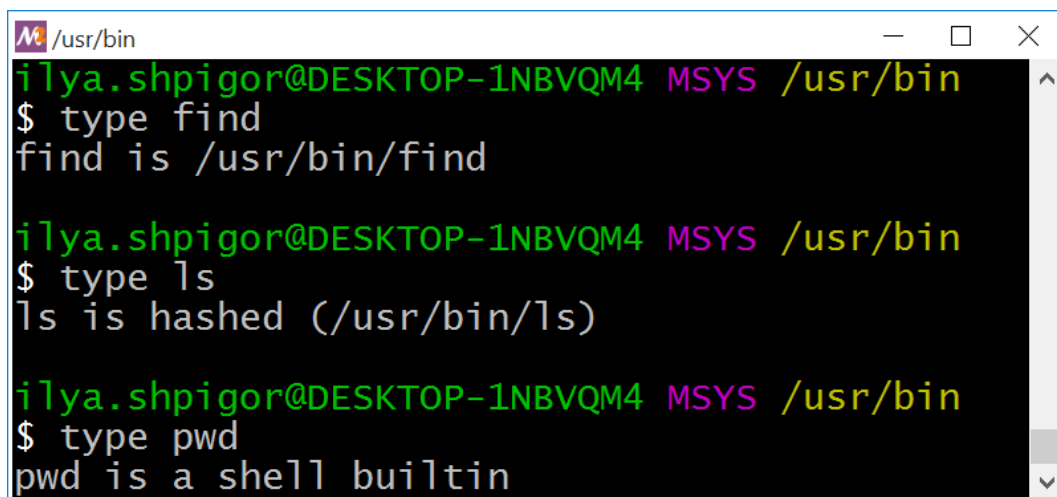
```
find / -name find
```

The command shows you two locations of the `find` file:

- /bin
- /usr/bin

Both locations present in the PATH variable.

There is a much faster way to locate an executable on the disk. The `type` Bash built-in does it. Call the command and give it a program name. You will get the absolute path to the program's executable. Figure 2-23 shows how it works.



```
ilya.shpigor@DESKTOP-1NBVQM4 MSYS /usr/bin
$ type find
find is /usr/bin/find

ilya.shpigor@DESKTOP-1NBVQM4 MSYS /usr/bin
$ type ls
ls is hashed (/usr/bin/ls)

ilya.shpigor@DESKTOP-1NBVQM4 MSYS /usr/bin
$ type pwd
pwd is a shell builtin
```

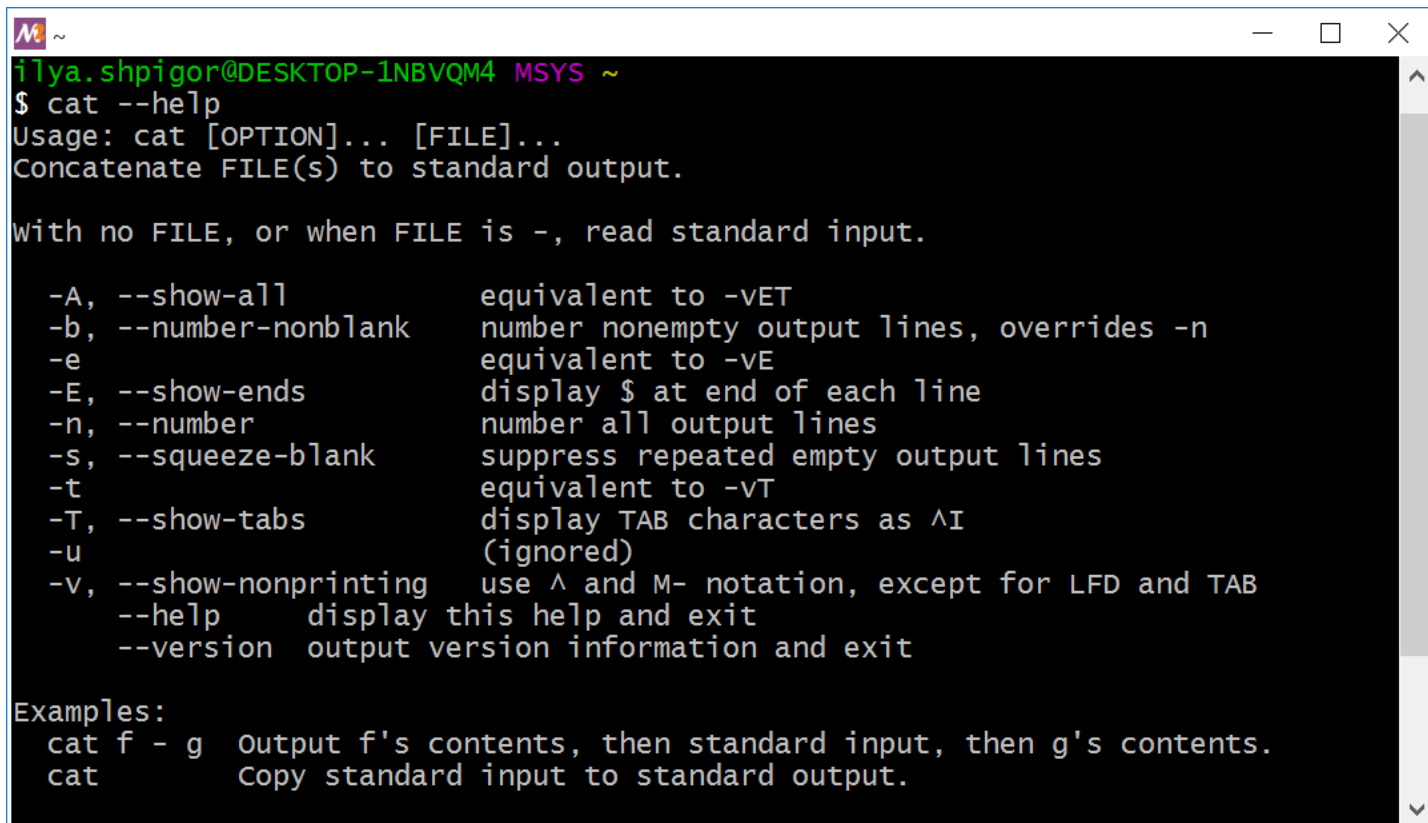
Figure 2-23. The output of the `type` command

²¹⁵<https://en.wikipedia.org/wiki/Newline>

You see that the `/usr/bin` directory stores the executables of `find` and `ls` utilities. The `ls` utility is marked as **hashed**. It means that Bash has remembered its path. When you call `ls`, the interpreter does not search the executable on the disk. Bash uses the stored path and calls the utility directly. If you move the hashed executable, Bash cannot find it anymore.

You can call the `type` command and pass a Bash built-in there. Then `type` tells you that Bash executes this command. Figure 2-23 shows an example of such output for the `pwd` command.

Suppose that you found the executable of the required utility. How do you know the parameters it accepts? Call the utility with the `--help` option. The option prints a brief help. Figure 2-24 shows this help for the `cat` utility.



```

tlya.shpigor@DESKTOP-INBVQM4 MSYS ~
$ cat --help
Usage: cat [OPTION]... [FILE]...
Concatenate FILE(s) to standard output.

With no FILE, or when FILE is -, read standard input.

  -A, --show-all           equivalent to -vET
  -b, --number-nonblank    number nonempty output lines, overrides -n
  -e                       equivalent to -vE
  -E, --show-ends         display $ at end of each line
  -n, --number            number all output lines
  -s, --squeeze-blank     suppress repeated empty output lines
  -t                       equivalent to -vT
  -T, --show-tabs        display TAB characters as ^I
  -u                       (ignored)
  -v, --show-nonprinting  use ^ and M- notation, except for LFD and TAB
  --help                 display this help and exit
  --version              output version information and exit

Examples:
cat f - g  output f's contents, then standard input, then g's contents.
cat       Copy standard input to standard output.

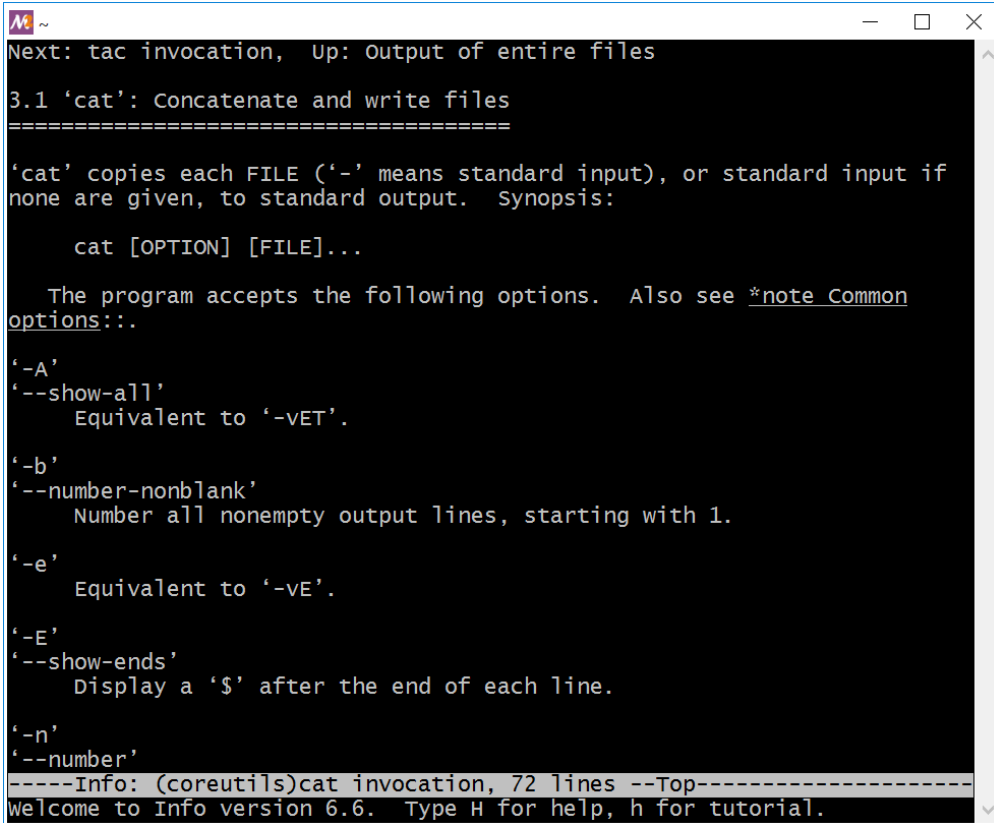
```

Figure 2-24. The brief help for the `cat` utility

If the brief help is not enough, refer to the documentation system called `info`. Suppose you need examples of how to use the `cat` utility. The following command shows them:

```
info cat
```

Figure 2-25 shows the result of the command.

A screenshot of a terminal window displaying the 'info' page for the 'cat' utility. The window title is 'M ~'. The text in the terminal is as follows:

```
Next: tac invocation, Up: Output of entire files
3.1 'cat': Concatenate and write files
=====
'cat' copies each FILE ('-' means standard input), or standard input if
none are given, to standard output. Synopsis:
    cat [OPTION] [FILE]...
The program accepts the following options. Also see *note Common
options::.
'-A'
'--show-all'
    Equivalent to '-vET'.
'-b'
'--number-nonblank'
    Number all nonempty output lines, starting with 1.
'-e'
    Equivalent to '-vE'.
'-E'
'--show-ends'
    Display a '$' after the end of each line.
'-n'
'--number'
-----Info: (coreutils)cat invocation, 72 lines --Top-----
welcome to Info version 6.6. Type H for help, h for tutorial.
```

Figure 2-25. The info page for the cat utility

You see a program for reading text documents. Use the arrow keys, PageUp and PageDown to scroll the text. Press the Q key to end the program.

Developers of GNU utilities have created the `info` system. Before that, all Unix distributions used the `man` page system. It is also known as `man`. The capabilities of `info` and `man` are similar. The MSYS2 environment uses the `info` system, which is more modern.

Your Linux distribution may use `man` instead of `info`. Use it in the same way as `info`. For example, the following `man` call shows you help for the `cat` utility:

```
man cat
```

When you know which utility solves your task, it is easy to get help. What would you do if you don't know how to solve the task? The best approach is to look for the answer on the Internet. You will find tips there. They are more concise than the manuals for GUI programs. You don't need screenshots and videos that explain each action. Instead, you will find a couple of lines with command calls that do everything you need.

Exercise 2-5. The documentation system

Find documentation for each of the built-in commands and utilities of Table 2-1. Check the parameters of the `ls` and `find` utilities that we did not consider.

Actions on Files and Directories

You have learned how to find a file or directory on the disk. Now let's discuss what you can do with it. If you have an experience with Windows GUI, you know the following actions with file system objects:

- Create
- Delete
- Copy
- Move or rename

Each of these actions has a corresponding GNU utility. Call them to manage the file system objects. Table 2-7 describes these utilities.

Table 2-7. Utilities for operating files and directories

Utility	Feature	Examples
<code>mkdir</code>	It creates the directory with the specified name and path.	<code>mkdir ~/tmp/docs</code> <code>mkdir -p tmp/docs/report</code>
<code>rm</code>	It deletes the specified file or directory	<code>rm readme.txt</code> <code>rm -rf ~/tmp</code>
<code>cp</code>	It copies a file or directory. The first parameter is the current path. The second parameter is the target path.	<code>cp readme.txt tmp/readme.txt</code> <code>cp -r /tmp ~/tmp</code>
<code>mv</code>	It moves or renames the file or directory specified by the first parameter.	<code>mv readme.txt documentation.txt.</code> <code>mv ~/tmp ~/backup</code>

Each of these utilities has the `--help` option. It displays a brief help. Please read it before using the utility the first time. You will find there some modes that this book misses. Refer to the `info` or `man` system if you need more details.

It is time to consider the utilities of Table 2-7.

mkdir

The `mkdir` utility creates a new directory. Specify its target path in the first parameter of the command. Here is an example `mkdir` call for creating the `docs` directory:

```
mkdir ~/docs
```

We specified the absolute path to the `docs` directory. You can pass the relative path instead. There are two steps to take it:

1. Navigate the home directory.
2. Call the `mkdir` utility there.

Here are the corresponding commands:

```
1 cd ~
2 mkdir docs
```

The utility has an option `-p`. It creates the nested directories. Here is an example of when to use it. Suppose you want to move the documents into the `~/docs/reports/2019` path. However, the `docs` and `reports` directories do not exist yet. If you use `mkdir` in the default mode, you should call it three times to create each of the nested directories. Another option is to call `mkdir` once with the `-p` option like this:

```
mkdir -p ~/docs/reports/2019
```

This command succeeds even if the `docs` and `reports` directories already exist. It creates only the missing `2019` directory in this case.

rm

The `rm` utility deletes files and directories. Specify the object to delete by its absolute or relative path. Here are examples of `rm` calls:

```
1 rm report.txt
2 rm ~/docs/reports/2019/report.txt
```

The first call deletes the `report.txt` file in the current directory. The second one deletes it in the `~/docs/reports/2019` path.

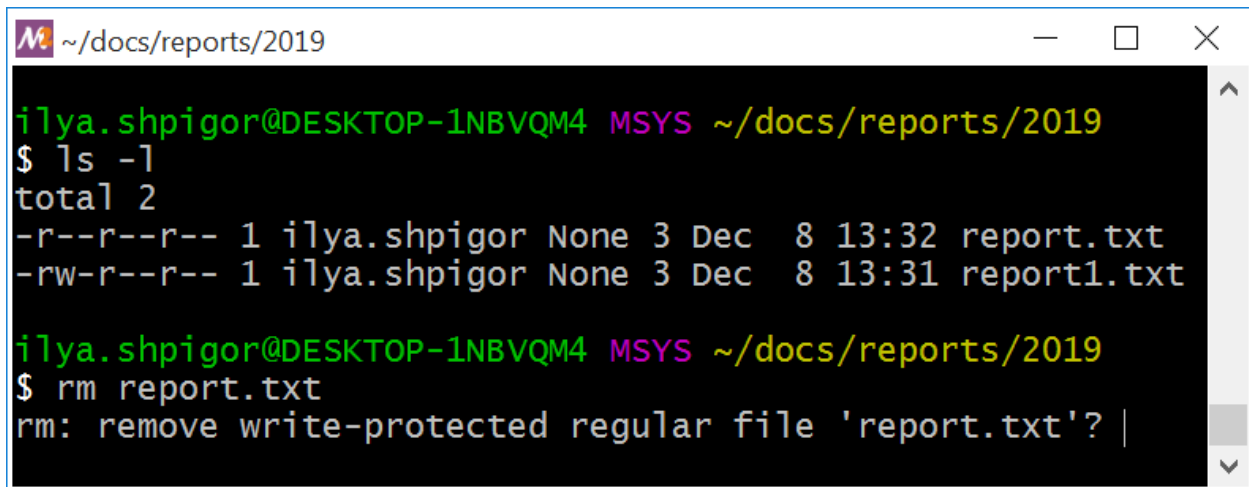
The `rm` utility can remove several files at once. Specify a list of filenames separated by spaces in this case. Here is an example:


```
rm report.txt ~/docs/reports/2019/report.txt
```

If you want to delete dozens of files, listing them all is inconvenient. Use a Bash glob pattern in this case. For example, you need to delete all text files whose names begin with the word “report”. The following `rm` call does it:

```
rm ~/docs/reports/2019/report*.txt
```

When removing a write-protected file, the `rm` utility shows you a warning. You can see how it looks like in Figure 2-26.



```

~/docs/reports/2019
ilya.shpigor@DESKTOP-1NBVQM4 MSYS ~/docs/reports/2019
$ ls -l
total 2
-r--r--r-- 1 ilya.shpigor None 3 Dec  8 13:32 report.txt
-rw-r--r-- 1 ilya.shpigor None 3 Dec  8 13:31 report1.txt

ilya.shpigor@DESKTOP-1NBVQM4 MSYS ~/docs/reports/2019
$ rm report.txt
rm: remove write-protected regular file 'report.txt'? |

```

Figure 2-26. The warning when deleting a write-protected file

When you see such a warning, there are two options. You can press the Y (short for yes) and Enter. Then the `rm` utility removes the file. Another option is to press N (no) and Enter. It cancels the operation.

If you want to suppress any `rm` warnings, use the `-f` or `--force` option. The utility removes files without confirmation in this case. Here is an example call:

```
rm -f ~/docs/reports/2019/report*.txt
```



The GNU utility options have two forms. The short form consists of one letter and begins with a dash `-`. The full form is a word preceded by a double dash `--`. This option format is recommended by the [POSIX standard](https://www.gnu.org/software/libc/manual/html_node/Argument-Syntax.html)²¹⁶ and its [GNU extension](https://www.gnu.org/prep/standards/html_node/Command_Line-Interfaces.html)²¹⁷.

The `rm` utility cannot remove a directory unless you pass one of two possible options there. The first option is `-d` or `--dir`. Use it for removing an empty directory. Here is an example:

²¹⁶https://www.gnu.org/software/libc/manual/html_node/Argument-Syntax.html

²¹⁷https://www.gnu.org/prep/standards/html_node/Command_Line-Interfaces.html

```
rm -d ~/docs
```

If the directory contains files or subdirectories, use the `-r` or `--recursive` option to remove it. Such a call looks like this:

```
rm -r ~/docs
```

The `-r` option removes empty directories too. Therefore, you can always use the `-r` option when calling `rm` for a directory.

cp and mv

The `cp` and `mv` utilities copy and move file system objects. Their interfaces are almost the same. Specify the target file or directory in the first parameter. Pass the new path for the object in the second parameter.

Here is an example. You want to copy the `report.txt` file. First, you should come to its directory. Second, call the `cp` utility this way:

```
cp report.txt report-2019.txt
```

This command creates the new file `report-2019.txt` in the current directory. Both `report-2019.txt` and `report.txt` files have the same contents.

Suppose that you do not need the old file `report.txt`. You can remove it with the `rm` utility after copying. The second option is to combine copying and removing in a single command. The `mv` utility does that:

```
mv report.txt report-2019.txt
```

This command does two things. First, it copies the `report.txt` file with the new name `report-2019.txt`. Second, it removes the old file `report.txt`.

Both `cp` and `mv` utilities accept relative and absolute paths. For example, let's copy a file from the home directory to the `~/docs/reports/2019` path. Here is the command for that:

```
cp ~/report.txt ~/docs/reports/2019
```

This command copies the `report.txt` file into the `~/docs/reports/2019` directory. The copy has the same name as the original file.

You can repeat the copying command with relative paths. Come to the home directory and call the `cp` utility there. The following commands do it:

```
1 cd ~
2 cp report.txt docs/reports/2019
```

When copying a file between directories, you can specify the copy name. Here is an example:

```
cp ~/report.txt ~/docs/reports/2019/report-2019.txt
```

This command creates a file copy with the `report-2019.txt` name.

Moving files works the same way as copying. For example, the following command moves the `report.txt` file:

```
mv ~/report.txt ~/docs/reports/2019
```

The following command moves and renames the file at once:

```
mv ~/report.txt ~/docs/reports/2019/report-2019.txt
```

You can rename a directory using the `mv` utility too. Here is an example:

```
mv ~/tmp ~/backup
```

This command changes the name of the `tmp` directory to `backup`.

The `cp` utility cannot copy a directory when you call it in the default mode. Here is an example. Suppose you want to copy the directory `/tmp` with the temporary files to the home directory. You call `cp` this way:

```
cp /tmp ~
```

This command fails.

You must add the `-r` or `--recursive` option when copying directories. Then the `cp` utility can handle them. This is the correct command for our example:

```
cp -r /tmp ~
```

Suppose you copy or move a file. If the target directory already has the file with the same name, the `cp` and `mv` utilities ask you to confirm the operation. If you press the `Y` and `Enter` keys, utilities overwrite the existing file.

There is an option to suppress the confirmation when copying and moving files. Use the `-f` or `--force` option. It forces `cp` and `mv` utilities to overwrite the existing files. Here are examples:

```
1 cp -f ~/report.txt ~/tmp
2 mv -f ~/report.txt ~/tmp
```

Both commands overwrite the existing `report.txt` file in the `tmp` directory. You do not need to confirm these operations.

Exercise 2-6. Operations with files and directories

Handle your photos from the past three months using the GNU utilities.

Make a backup before you start.

Separate all photos by year and month.

You should get a directory structure like this:

```
~/
  photo/
    2019/
      11/
      12/
    2020/
      01/
```

File System Permissions

Each utility of Table 2-7 checks the **file system permissions**²¹⁸ before acting. These permissions define if you are allowed to operate the target object. Let's consider this file system mechanism in detail.

The permissions restrict the user actions on the file system. The OS tracks these actions and checks their allowance. Each user can access only his files and directories, thanks to this feature. It also restricts access to the OS components.



Users can share files with each other, but they should allow it explicitly. This sharing does not work by default.

The permissions allow several people to share one computer. This workflow was widespread in the 1960s until the advent of PCs. Hardware resources were expensive at that time. Therefore, several users have to operate with one computer.

Today most users have their own PC or laptop. However, the file system permissions are still relevant. They protect your Linux or macOS system from unauthorized access and malware.

Have a look at Figure 2-26 again. There you see the output of the `ls` utility with the `-l` option. It is the table. Each row corresponds to a file or directory. The columns have the following meaning:

²¹⁸https://en.wikipedia.org/wiki/File-system_permissions

1. Permissions to the object.
2. The number of hard links to the file or directory.
3. Owner.
4. Owner's group.
5. The object's size in bytes.
6. Date and time of the last change.
7. File or directory name.

The permissions to the file `report.txt` equal the “`-r-r-r-`” string. What does it mean?

Unix stores permissions to a file object as a **bitmask**²¹⁹. The bitmask is a positive integer. When you store it in computer memory, the integer becomes a sequence of zeros and ones. Each bit of the mask keeps a value that is independent of the other bits. Therefore, you can pack several values into a single bitmask.

What values can you store in a bitmask? This is a set of object's properties, for example. Each bit of the mask corresponds to one property. If it is present, the corresponding bit equals one. Otherwise, the bit equals zero.

Let's come back to the file access rights. We can represent these rights as the following three attributes:

1. Read permission.
2. Write permission.
3. Permission to execute.

If you apply a mask of three bits, you can encode these attributes there. Suppose a user has full access to the file. He can read, change, copy, remove or execute it. It means that the user has read, write, and execute permissions to the file. The writing permission allows changing the file and removing it. Therefore, the file permissions mask looks like this:

```
111
```

Suppose the user cannot read or execute the file. The first bit of the mask corresponds to the read access. The third bit is execution permission. When you set both these bits to zero, you restrict the file access. Then you get the following mask:

```
010
```

You should know the meaning of each bit in the mask if you want to operate it properly. The mask itself does not provide this information.

Our mask with three bits is a simplified example of file permissions. The permissions in Unix follow the same idea. However, bitmasks there have more bits. The `ls` utility prints these access rights to the `report.txt` file:

²¹⁹[https://en.wikipedia.org/wiki/Mask_\(computing\)](https://en.wikipedia.org/wiki/Mask_(computing))

-r--r--r--

This string is the bitmask. Here dashes correspond to zeroed bits. Latin letters match the set bits. If you follow this notation, you can convert the “-r-r-r-” string to the 0100100100 mask. If all bits of the mask equal ones, the `ls` prints it like the “drwxrwxrwx” string.

The Unix permissions string has four parts. Table 2-8 explains their meaning.

Table 2-8. Parts of the permissions string in Unix

d	rwX	rwX	rwX
The directory attribute.	The permissions of the object’s owner. The owner is a user who has created the object.	The permissions of the user group that is attached to the object. By default, it is the group to which the owner belongs.	The permissions of all other users except the owner and the group attached to the object.



The `groups` utility prints the list of groups that the current user belongs.

You can imagine the Unix permissions as four separate bitmasks. Each of them corresponds to one part of Table 2-8. All bitmasks have a size of four bits. Using this approach, you can represent the “-r-r-r-” string this way:

0000 0100 0100 0100

The Latin letters in the Unix permissions have special meaning. First of all, they match bits that are set to one. The position of each bit defines the allowed action to the object. You do not need to remember the meaning of each position. The Latin letters give you a hint. For example, “r” means read access. Table 2-9 explains the rest letters.

Table 2-9. Letters in the Unix permissions string

Letter	Meaning for a file	Meaning for a directory
d	If the first character is a dash instead of <code>d</code> , the permissions correspond to a file.	The permissions correspond to a directory.
r	Access for reading.	Access for listing the directory contents.
w	Access for writing.	Access for creating, renaming or deleting objects in the directory.
x	Access for executing.	Access for navigating to the directory and accessing its nested objects.

Table 2-9. Letters in the Unix permissions string

Letter	Meaning for a file	Meaning for a directory
—	The corresponding action is prohibited.	The corresponding action is prohibited.

Suppose that all users of the system have full access to the file. Then its permissions look like this:

```
-rwxrwxrwx
```

If all users have full access to a directory, the permissions look this way:

```
drwxrwxrwx
```

The only difference is the first character. It is `d` instead of the dash.

Now you know everything to read the permissions of Figure 2-26. It shows two files: `report.txt` and `report1.txt`. All users can read the first one. Nobody can modify or execute it. All users can read the `report1.txt` file. Only the owner can change it. Nobody can execute it.

We have considered commands and utilities for operating the file system. When you call each of them, you specify the target object. You should have appropriate permissions to the object. Otherwise, your command fails. Table 2-10 shows the required permissions.

Table 2-10. Commands and required file system permissions for them

Command	Required Bitmask	Required Permissions	Comment
<code>ls</code>	<code>r--</code>	Reading	Applied for directories only.
<code>cd</code>	<code>--x</code>	Executing	Applied for directories only.
<code>mkdir</code>	<code>-wx</code>	Writing and executing	Applied for directories only.
<code>rm</code>	<code>-w-</code>	Writing	Specify the <code>-r</code> option for the directories.
<code>cp</code>	<code>r--</code>	Reading	The target directory should have writing and executing permissions.
<code>mv</code>	<code>r--</code>	Reading	The target directory should have writing and executing permissions.
Execution	<code>r-x</code>	Reading and executing.	Applied for files only.

Files Execution

Windows has strict rules for executable files. The file extension defines its type. The Windows loader runs only files with the EXE and COM extensions. These are compiled executable of programs. Besides them, you can run scripts. The script's extension defines the interpreter that launches it. Windows cannot run the script if there is no installed interpreter for it. The possible extensions of the scripts are BAT, JS, PY, RB, etc.

Unix rules for executing files differ from Windows ones. Here you can run any file if it has permissions for reading and executing. Its extension does not matter, unlike Windows. For example, the file called `report.txt` can be executable.

There is no convention for extensions of the executable files in Unix. Therefore, you cannot deduce the file type from its name. Use the `file` utility to get it. The command receives the file path on input and prints its type. Here is an example of calling `file`:

```
file /usr/bin/ls
```

If you launch the command in the MSYS2 environment, it prints the following information:

```
/usr/bin/ls: PE32+ executable (console) x86-64 (stripped to external PDB), for MS Wi\ndows
```

The output says that the `/usr/bin/ls` file has the [PE32²²⁰](#) type. Files of this type are executable and contain machine code. The Windows loader can run them. The output also shows the bitness of the file “x86-64”. It means that this version of `ls` utility works on 64-bit Windows only.

If you run the same `file` command on Linux, it gives another output. For example, it might look like this:

```
/bin/ls: ELF 64-bit LSB executable, x86-64, version 1 (SYSV), dynamically linked, in\terpreter /lib64/ld-linux-x86-64.so.2, for GNU/Linux 2.6.32, BuildID[sha1]=d0bc0fb9b\3f60f72bbad3c5a1d24c9e2a1fde775, stripped
```

This is the executable file with machine code. It has the [ELF²²¹](#) type. Linux loader can run it. The file bitness “x86-64” is the same as in MSYS2.

We have learned to distinguish executable and non-executable files in the Unix environment. Now let's find out where you can find them.

GNU utilities are part of OS. Therefore, they are available right after installing the system. You do not need to install them separately. Their executable files are located in the `/bin` and `/usr/bin`

²²⁰https://en.wikipedia.org/wiki/Portable_Executable

²²¹https://en.wikipedia.org/wiki/Executable_and_Linkable_Format

directories. The Bash variable `PATH` stores these paths. Now the question is, where can you find newly installed applications?

Windows installs new applications in the `Program Files` and `Program Files (x86)` directories on the system drive. Each application has its own subdirectory. For example, it can be `C:\Program Files (x86)\Notepad++`. The installer program copies executables, libraries, configuration and resource files into that subdirectory. The application requires all these files to work properly. You can specify another installation directory than `Program Files` and `Program Files (x86)`. Then the installer program creates the application subdirectory there.

There are two approaches to install applications to the Unix environment. The first one resembles the Windows way. There is the system directory `/opt`. The installer program creates an application subdirectory with all its files there.

Here is an example. Suppose that you are installing the TeamViewer application. Its installer creates the `/opt/teamviewer` subdirectory. You can find the `TeamViewer` executable there. Developers of proprietary applications prefer this way of installing programs.

Developers of open-source programs follow another approach. An application requires files of various types. Each file type has a separate system directory in Unix. It means that the executable files of all applications occupy the same directory. The documentation for them is in another directory and so on. The POSIX standard dictates the purposes of all system directories.

Table 2-11 explains the purposes of Unix system directories.

Table 2-11. Unix system directories

Directory	Purpose
<code>/bin</code>	It stores executable files of system utilities.
<code>/etc</code>	It stores configuration files of applications and system utilities.
<code>/lib</code>	It stores libraries of system utilities.
<code>/usr/bin</code>	It stores executable files of user applications.
<code>/usr/lib</code>	It stores libraries of user applications.
<code>/usr/local</code>	It stores applications that the user compiled on his own.
<code>/usr/share</code>	It stores architecture-independent resource files of user applications (e.g. icons).
<code>/var</code>	It stores files created by applications and utilities while running (e.g. log files).

Copying all files of the same type into one directory sounds like a controversial solution. Its

disadvantage is the complexity of maintenance. Suppose that the application updates to the next version. It should update all its files in all system directories. If the application misses one of the files, it cannot run anymore.

However, the Unix system directories have an advantage. When you install an application on Windows, it brings all files it needs. There are libraries with subroutines among these files. Some applications require the same libraries to run. When each application has its own copy of the same library, it causes the file system overhead.

The Unix way gets rid of library copies. Suppose that all applications respect the agreement and install their files to the proper system directories. Then applications can locate files of each other. Therefore, they can use the same library if they require it. A single instance of each library is enough for supporting all dependent applications.

Suppose that you have installed a new application (e.g., a browser). Its executable file (for example, `firefox`) comes to the `/usr/bin` path according to Table 2-11. How to run this application in Bash? There are several ways for that:

1. By the name of the executable file.
2. By the absolute path.
3. By the relative path.

Let's consider each way in detail.

You have used the first approach when calling GNU utilities. For example, the following command runs the `find` utility:

```
find --help
```

It launches the `/usr/bin/find` executable file.

Use a similar command to run a newly installed application. Here is an example for the Firefox browser:

```
firefox
```

Why does this command work? The executable file `firefox` is located in the `/usr/bin` system directory. The Bash variable `PATH` stores this path. When Bash receives the “`firefox`” command, it searches the executable with that name. The shell takes searching paths from the `PATH` variable. This way, Bash finds the `/usr/bin/firefox` file and launches it.

The paths have a specific order in the `PATH` variable. Bash follows this order when searching for an executable. There is an example. Suppose that both `/usr/local/bin` and `/usr/bin` directories contain the `firefox` executable. If the path `/usr/local/bin` comes first in the `PATH` list, Bash always runs the file from there. Otherwise, Bash calls the `/usr/bin/firefox` executable.

The second way to run an application reminds the first one. Instead of the executable filename, you type its absolute path. For example, the following command runs the Firefox browser:

```
/usr/bin/firefox
```

You would need this approach when launching proprietary applications. They are installed to the `/opt` system directory. The `PATH` variable does not contain this path by default. Therefore, Bash cannot find executables there. You can help Bash by specifying an absolute path to the program.

The third approach to run an application is something in between the first and second ways. You use a relative executable path instead of the absolute one. Here is an example for the Firefox browser:

```
1 cd /usr
2 bin/firefox
```

The first command navigates to the `/usr` directory. Then the second command launches the browser by its relative path.

Now let's change the first command. Suppose that you navigate the `/opt/firefox/bin` directory. The following try to launch the browser fails:

```
1 cd /opt/firefox/bin
2 firefox
```

Bash reports that it cannot find the `firefox` file. It happens because you are launching the application by the executable filename here. It is the first way to run applications. Bash looks for the `firefox` executable in the paths of the `PATH` variable. However, the application is located in the `/opt` directory, which is not there.

You should specify the relative path to the executable instead of its filename. If the current directory contains the file, mention it in the relative path. The dot symbol indicates the current directory. Thus, the following commands run the browser properly:

```
1 cd /opt/firefox/bin
2 ./firefox
```

Now Bash follows your hint and searches the executable in the current directory.

Suppose that you have installed a new application. You are going to use it in your daily work frequently. Add its installation path to the `PATH` variable in this case. The following steps explain how to do it for the `/opt/firefox/bin` directory:

1. Navigate the home directory:

```
cd ~
```

2. Print its Windows path:

```
pwd -W
```

3. Open the file `~/.bash_profile` in the text editor (for example, Notepad).
4. Add the following line at the end of the file:

```
PATH="/opt/firefox/bin:${PATH}"
```

You have redefined the `PATH` variable this way. The next chapter considers Bash variables in detail. There you will know how to operate them.

Restart the MSYS2 terminal for applying changes. Now you can run the browser by the name. Bash finds the corresponding executable in the `/opt/firefox/bin` path correctly.

Extra Bash Features

We have learned the basic Bash built-ins and GNU utilities for operating the file system. Now you know how to run a program or copy a file using the shell. You can do the same things in GUI. When solving such simple tasks, both types of interfaces are effective.

Bash provides several features that a GUI does not have. They give you a significant performance gain when solving some specific tasks. Use these features to automate your work and save time.

These are the Bash features that we are talking about:

1. I/O redirection.
2. Pipeline.
3. Logical operators.

Unix Philosophy

[Douglas McIlroy](https://en.wikipedia.org/wiki/Douglas_McIlroy)²²² is one of the Unix developers. He wrote several famous command-line utilities: `spell`, `diff`, `sort`, `join`, `graph`, `speak`, and `tr`. McIlroy summarized the best practices of software development for Unix in the following points:

1. Write programs that do one thing and do it well.
2. Write programs to work together.
3. Write programs to handle text streams, because that is a universal interface.

²²²https://en.wikipedia.org/wiki/Douglas_McIlroy

These principles became a part of the [Unix philosophy](#)²²³.

The cornerstone of the Unix philosophy is the plain text format. McIlroy emphasized its significant role and called it a universal interface. Using text format allows you both to develop highly specialized programs and combine them together.

The primary feature of the text format is the simplicity of data exchange between programs. Here is an example. Suppose that two developers wrote two utilities independently of each other. The utilities use the text format for input and output data. This decision allows you to combine these utilities. This way, you apply their feature to solve your task. All you need is to pass the output of the first utility to the input of the second one.

When programs interact easily, there is no need to overload them with extra features. For example, you write a program that copies files. It does the job well. At some moment, you realize that the program needs a search function. This feature will speed up your work because you can find and copy the files at once. You add the searching mechanism and it works well. Then you decide to add the feature of creating new directories. It makes the program more convenient for use and so on. This example shows that the requirements for a self-contained application grow rapidly.

Having a universal interface, you can get a special program for each task. You do not need to add a search function to your program. Just call the `find` utility and use its results. This utility works better than your code. The reason is many people use it for a long time. Therefore, they find most of its bugs and fixed them.

Always prefer existed utilities when developing software for Unix and Linux.

I/O Redirection

GNU utilities were done by 1987. The idea behind them is to provide open-source software for Unix developers. Most of the original Unix programs were proprietary. It means that you should buy a Unix system and launch them there.

GNU utilities copy all features of their originals. Therefore, they follow the Unix philosophy too. Using them, you get all benefits of the universal text format.

You have several options to transfer text data between the utilities. The simplest way is using the clipboard. Suppose that the data fits one line. Follow these steps to move them from one utility to another one:

1. Select the utility's output with the mouse. Now you got it in the clipboard.
2. Type the command to call another utility.
3. Paste the clipboard data with the middle mouse button.
4. Launch the command.

²²³https://en.wikipedia.org/wiki/Unix_philosophy

This approach does not work for copying multiple lines. When you paste them, Bash handles each line break as a command delimiter. It treats the text before the delimiter as a separate command. Thus, the shell loses some copied lines.

Another solution for data transfer is using the file system. Create a temporary file to save the utility's output. Then pass the filename to another utility. It will read the data there. This approach is more convenient than the clipboard for two reasons:

1. There is no limit on the number of text lines to transfer.
2. There are no manual operations with the clipboard.

Bash provides you a mechanism that redirects the command's output to the file. It can also redirect the file contents to the command input. It means that your application does not need a feature for interacting with the file system. Instead, it should support the text data format. Then Bash takes care of redirecting data.

Here is an example of redirecting text data. Suppose that you are looking for the files on the disk. You want to save the searching result into a file. Use the `find` utility and the redirection operator `1>`. Then the utility call looks like this:

```
find / -path */doc/* -name README 1> readme_list.txt
```

The command creates the `readme_list.txt` file in the current directory. It writes the `find` utility's output there. The file contents look the same as it is printed on the screen without the redirection operator. If the current directory has the `readme_list.txt` file already, the command overwrites it.

What does the `1>` operator mean? It is a [redirection](https://en.wikipedia.org/wiki/Redirection_(computing))²²⁴ of the standard output stream. There are three [standard streams](https://en.wikipedia.org/wiki/Standard_streams)²²⁵ in Unix. Table 2-12 explains them.

Table 2-12. POSIX standard streams

Number	Name	Purpose
0	Standard input stream (stdin).	A program receives input data from this stream. By default, it comes from an input device like a keyboard.
1	Standard output stream (stdout).	A program outputs data there. The terminal window prints this stream by default.
2	Standard error stream (stderr).	A program outputs the error messages there. The terminal window prints this stream by default.

Any program operates in the software environment that the OS provides. You can imagine each standard stream as a communication channel between the program and the OS environment.

²²⁴[https://en.wikipedia.org/wiki/Redirection_\(computing\)](https://en.wikipedia.org/wiki/Redirection_(computing))

²²⁵https://en.wikipedia.org/wiki/Standard_streams

Early Unix systems have used only physical channels for data input and output. The input channel comes from the keyboard. The same way the output channel goes to the monitor. Then developers introduced the streams as an [abstraction](#)²²⁶ for these channels.

The abstraction makes it possible to work with different objects using the same algorithm. It allows replacing a real device input with the file data. Similarly, it replaces printing data to the screen with writing them to the file. The same OS code handles these I/O operations.

The purpose of the input and output streams is clear. However, the error stream causes questions. Why does a program need it? Imagine that you run the `find` utility to search for files. You do not have access to some directories. When the `find` utility reads their contents, it is unavailable. The utility reports about these issues in the error messages.

Suppose that the `find` utility found many files. You can miss error messages in a huge file list. Separating the output and error streams helps you in this case. For example, you can redirect the output stream to the file. Then the utility prints only error messages on the screen.

The `2>` operator redirects the standard error stream. Use it in the same way as the `1>` operator. Here is an example with the `find` utility:

```
find / -path */doc/* -name README 2> errors.txt
```

Each redirection operator has a number before the angle bracket. The number specifies the stream's number from Table 2-12. For example, the `2>` operator redirects the second stream.

If you need to redirect the standard input stream, the operator looks like `0<`. You cannot handle this stream with the `0>` operator. Here is an example. The following call searches the “Bash” pattern in the `README.txt` file:

```
grep "Bash" 0< README.txt
```

This command is for demonstration only. It uses the `grep` utility's interface that handles the standard input stream. However, the utility can read the contents of a specified file on its own. Use this mechanism and always pass the filename to the utility. Here is an example:

```
grep "Bash" README.txt
```

Let's take a more complicated example. Some Bash manuals recommend the `echo` command to print a file's contents. Using this approach, you can print the `README.txt` file this way:

```
echo $( 0< README.txt )
```

Here `echo` receives the output of the following command:

²²⁶https://en.wikipedia.org/wiki/Abstraction_layer

```
0< README.txt
```

We have used the **command substitution** Bash mechanism to embed one command into another one. When the shell encounters the `$(` and `)` characters, it executes everything enclosed between them. Then Bash inserts the output of the executed command instead of the `$(. . .)` block.

Bash executes our `echo` call in two steps because of the command substitution. These are the steps:

1. Pass the `README.txt` file's contents to the standard input stream.
2. Print data from the input stream on the screen.

Please take into account the execution order when invoking the command substitution. Bash executes sub-commands and inserts their results in the order they follow. Only when all substitutions are done, Bash executes the resulting command.

The following `find` call demonstrates a typical mistake when using the command substitution:

```
$ find / -path */doc/* -name README -exec echo ${0< {}} \;
```

This command should print the contents of all found files. However, it leads to the following Bash error:

```
bash: {}: No such file or directory
```

The problem happens because Bash executes the `0< {}` command before calling the `find` utility. When executing this command, the shell redirects the file called `{}` to the input stream. However, there is no file with such a name. We expect the `find` utility substitutes the brackets `{}` by its results. It does not happen because these results are not ready yet.

Replace the `echo` command with the `cat` utility. It solves the problem. Then you get the following `find` call:

```
find / -path */doc/* -name README -exec cat {} \;
```

This command prints the contents of all found files.

Bash users apply the redirection operators frequently. Therefore, the shell developers have added short forms for some of them. Here are these forms:

- The `<` operator redirects the input stream.
- The `>` operator redirects the output stream.

Here is an example of using the `>` operator:


```
find / -path */doc/* -name README > readme_list.txt
```

This command writes a list of all found README files to the `readme_list.txt` file.

Here is an example of using the `<` operator:

```
echo $( < README.txt )
```

Suppose that you redirect the output stream to a file. Then you found that this file already exists. You decide to keep its content and add new data at the end. Use the `>>` operator in this case.

Here is an example. You are searching the README files in the `/usr` and `/opt` system directories. You want to store the searching results in the `readme_list.txt` file. Then you should call the `find` utility twice. The first call uses the `>` operator. The second call should use the `>>` operator. These calls look like this:

```
1 find /usr -path */doc/* -name README > readme_list.txt
2 find /opt -name README >> readme_list.txt
```

The first `find` call creates the `readme_list.txt` file and writes its result there. If the file already exists, the utility overwrites its contents. The second `find` call appends its output to the end of `readme_list.txt`. If the file does not exist, the `>>` operator creates it.

The full form of the `>>` operator looks like `1>>`. You can use this form for both output and error streams. The operator for redirecting the error stream looks like `2>>`.

Suppose that you need to redirect both the output and the error streams to the same file. Use the `&>` and `&>>` operators in this case. The first operator overwrites an existing file. The second one appends data at the end of the file. Here is an example:

```
find / -path */doc/* -name README &> result_and_errors.txt
```

This command works in Bash properly. However, the `&>` operator may be absent in other shells. If you should use the features of the POSIX standard only, apply the `2>&1` operator. Here is an example:

```
find / -path */doc/* -name README > result_and_errors.txt 2>&1
```

The `2>&1` operator is called **stream duplication**. It redirects both output and error streams to the same target.

Be careful when using stream duplication. It is easy to make a mistake and mix up the operators' order in a command. If you work in Bash, always prefer the `&>` and `&>>` operators.

The following command demonstrates a mistake with using stream duplication:

```
find / -path */doc/* -name README 2>&1 > result_and_errors.txt
```

This command outputs the error stream data on the screen. However, we expect that these data come to the `result_and_errors.txt` file. The problem happens because of the wrong order of `2>&1` and `>` operators.

Here are the details of the problem. The POSIX standard has the concept of the [file descriptor](#)²²⁷. The descriptor is a pointer to a file or communication channel. It serves as an abstraction that makes it easier to handle streams.

When you start a program, both descriptors of output and error streams point to the terminal window. You can associate them with files instead. If you do so, the streams' descriptors point to that file. The [BashGuide article](#)²²⁸ describes this mechanism in detail.

Let's go back to our `find` call. Bash processes redirection operators one by one from left to right. Table 2-13 shows how it happens in our example.

Table 2-13. The order for applying redirection operators

Number	Operation	Result
1	<code>2>&1</code>	Now the error stream points to the same target as the output stream. The target is the terminal window.
2	<code>> result_and_errors.txt</code>	Now the output stream points to the file <code>result_and_errors.txt</code> . The error stream is still associated with the terminal window.

We should change the order of the redirection operators to fix the mistake. The `>` operator comes first. Then stream duplication takes place. Here is the corrected command:

```
find / -path */doc/* -name README > result_and_errors.txt 2>&1
```

Both output and error streams point to the `result_and_errors.txt` file here.

Suppose that you want to redirect output and error streams into two different files. Specify redirection operators for each stream one after another in this case. Here is an example:

```
find / -path */doc/* -name README > result.txt 2> errors.txt
```

The order of the operators is not important here.

Pipeline

The redirection operators are useful when you save data for manual analysis or processing. When you want to process data with another program, storing them in temporary files is inconvenient.

²²⁷https://en.wikipedia.org/wiki/File_descriptor

²²⁸<http://mywiki.woledge.org/BashFAQ/055>

Managing these files takes extra effort. You should keep in mind their paths and remove them after usage.

Unix provides an alternative solution for transferring text data. It is called a [pipeline](#)²²⁹. This mechanism shares data between programs by passing messages. It does not use the file system.

An example will demonstrate how pipelines work. Suppose that you are looking for information about the Bash [license](#)²³⁰. Bash documentation has it. Therefore, you call the `grep` utility to parse documentation files this way:

```
grep -R "GNU" /usr/share/doc/bash
```

Another source of the Bash license information is the `info` help page. You can take this information and transfer it to the `grep` utility. The pipeline does this job. It takes the output of one program and sends it to the input of another one. The following command does it for `info` and `grep` programs:

```
info bash | grep -n "GNU"
```

The `info` utility sends its result to the output stream. This stream is associated with the monitor by default. Therefore, you see the information in the terminal window. You do not get it in our case because of the pipeline.

The vertical bar `|` means the pipeline operator. When you add it after the `info` call, the utility's output comes to the pipeline. You should add another command after the `|` symbol. This command receives the data from the pipeline. This is the `grep` call in our example.

The general algorithm of our command looks like this:

1. Call the `info` program to receive the Bash help.
2. Send the `info` output to the pipeline.
3. Call the `grep` utility.
4. Pass the data from the pipeline to `grep`.

The `grep` utility searches the “GNU” word in the input data. If the utility finds the word, it prints the corresponding string of the input data. Check the `grep` output. If it is not empty, the Bash license is [GNU GPL](#)²³¹.

We use the `-n` option when calling `grep`. It adds the line numbers to the utility's output. The option helps you find the exact place of the “GNU” word on the help page.

²²⁹[https://en.wikipedia.org/wiki/Pipeline_\(Unix\)](https://en.wikipedia.org/wiki/Pipeline_(Unix))

²³⁰https://en.wikipedia.org/wiki/Software_license

²³¹https://en.wikipedia.org/wiki/GNU_General_Public_License

du

Here is a more complex example with pipelines and the `du` utility. The utility evaluates disk space usage. Run it without parameters in the current directory. The utility passes through all subdirectories **recursively**. It prints the space occupied by each of them.

Traversing a directory recursively means visiting all its subdirectories. If any of them has subdirectories, we should visit them too, etc. The traversal algorithm looks like this:

1. Check the contents of the current directory.
2. If there is an unvisited subdirectory, go to it and start from the 1st step of the algorithm.
3. If all subdirectories are visited, go to the parent directory and start from the 1st step of the algorithm.
4. If it is impossible to go to the parent directory, finish the algorithm.

You should select a starting point for this algorithm. It is a specific file system path. Then the algorithm bypasses all subdirectories starting from this path. The traversing finishes when the algorithm should come to the parent directory of the starting point.

We have considered the universal traversal algorithm. You can add any action to it for processing each subdirectory. The action of the `du` utility is calculating disk space usage.

The algorithm of the `du` utility looks like this:

1. Check the contents of the current directory.
2. If there is an unvisited subdirectory, go to it and start from the 1st step of the algorithm.
3. If there are no subdirectories:
 - 3.1 Calculate and print the disk space occupied by the current directory.
 - 3.2 Go to the parent directory.
 - 3.3 Start from the 1st step of the algorithm.
4. If it is impossible to go to the parent directory, finish the algorithm.

When calling the `du` utility, you can specify a path to a file or directory. If you pass the file's path, the utility prints its size and finishes. In the case of the directory, it executes the traversing algorithm.

Here is an example of the `du` call for the `/usr/share` path:

```
du /usr/share
```

It gives the following output:

```

1 261    /usr/share/aclocal
2  58    /usr/share/awk
3 3623   /usr/share/bash-completion/completions
4  5     /usr/share/bash-completion/helpers
5 3700   /usr/share/bash-completion
6  2     /usr/share/cmake/bash-completion
7  2     /usr/share/cmake
8  8     /usr/share/cygwin
9 1692   /usr/share/doc/bash
10 85    /usr/share/doc/flex
11 ...

```

You see a table of two columns. The right column shows the subdirectories. The left column shows the number of bytes they occupy.

You can add the statistics for the files to the `du` output. Use the `-a` option for that. Here is an example `du` call:

```
du /usr/share -a
```

The `-h` option improves the `du` output. The option converts bytes to kilobytes, megabytes, and gigabytes.

Suppose that you want to evaluate the size of all HTML files in the `/usr/share` path. The following command does it:

```
du /usr/share -a -h | grep "\.html"
```

Here the pipeline redirects the `du` output to the `grep` input. The `grep` utility filters it and prints the lines that match the `“.html”` pattern.

The backslash `\` escapes the dot in the `“.html”` pattern. The dot means a single occurrence of any character. If you specify the `“.html”` pattern, the `grep` output includes non-HTML files (like `pod2html.pl` or `perl.gz`) and subdirectories (like `/usr/share/doc/pcre/html`). When you escape the dot, the `grep` utility treats it as a dot character.

The pipeline combines calls of `du` and `grep` utilities in our example. However, you can combine more than two commands. Suppose that you need to sort the found HTML files. Call the `sort` utility for doing this job. Then you get the following pipeline :

```
du /usr/share -a -h | grep "\.html" | sort -h -r
```

The `sort` utility sorts the strings in ascending **lexicographic order**²³² when called without options. The following example explains the lexicographic order. Suppose that you have a file. It has this contents:

²³²https://en.wikipedia.org/wiki/Lexicographic_order

```
1 abc
2 aaaa
3 aaa
4 dca
5 bcd
6 dec
```

You call the `sort` utility for this file. It gives you the following output:

```
1 aaa
2 aaaa
3 abc
4 bcd
5 dca
6 dec
```

The `-r` option of the utility reverts the sorting order. You get this output when applying the option:

```
1 dec
2 dca
3 bcd
4 abc
5 aaaa
6 aaa
```

The `du` utility prints its results in the table. The first column contains sizes of files and directories. The `sort` utility processes its input data line by line from left to right. Suppose that you transfer the `du` output to the `sort` input. Then the `sort` utility deals with numbers which are sizes of objects. The lexicographic sorting does not work well in this case. An example will explain the issue to you.

There is a file with three integers:

```
1 3
2 100
3 2
```

If you call the `sort` utility for this file, it gives you this result:

```
1 100
2 2
3 3
```

The utility deduces that 100 is less than 2 and 3. It happens because `sort` compares strings but not digits. It converts each character of two lines to the corresponding ASCII code. Then the utility compares these codes one by one from left to right. When the first difference happens, the utility chooses the smaller string and puts it first. This string has the character with the smaller code.

The `-n` option forces `sort` to compare digits instead of strings. If you add this option, the utility converts all string into the numbers and compare them. Using this option, you get the correct sorting order like this:

```
1 2
2 3
3 100
```

Let's come back to our command:

```
du /usr/share -a -h | grep "\.html" | sort -h -r
```

It works well because of the `-h` option of the `sort` utility. This option forces the utility to convert the object sizes to integers. For example, it converts “2K” to “2048” integer. If `sort` meets the “2048” string, the utility treats it as an integer. This way, `sort` can process the `du` output.

You can combine pipelines with stream redirection. Suppose that you want to save the filtered and sorted `du` output to the file. The following command does it:

```
du /usr/share -a -h | grep "\.html" | sort -h -r > result.txt
```

The `result.txt` file gets the `sort` output here.

There is an option to split data streams when you combine pipelines and the redirection operator. For example, you want to write the output stream into the file and pass it to another utility at once. Bash does not have a mechanism for this task. However, you can call the `tee` utility for doing that. Here is an example:

```
du /usr/share -a -h | tee result.txt
```

The command prints the `du` output on the screen. It writes this output into the `result.txt` file at the same time. The `tee` utility duplicates its input stream to the specified file and the output stream. The utility overwrites the contents of `result.txt` if it exists. Use the `-a` option if you want to append data to the existing file.

Sometimes you need to check the data flow between commands in a pipeline. The `tee` utility helps you in this case. Just call the utility between the commands in the pipeline. Here is an example:

```
du /usr/share -a -h | tee du.txt | grep "\.html" | tee grep.txt | sort -h -r > result.txt
```

Each `tee` call stores the output of the previous pipeline command to the corresponding file. This intermediate output helps you to debug possible mistakes. The `result.txt` file contains the final result of the whole pipeline.

xargs

The `find` utility has the `-exec` parameter. It calls the specified command for each found object. This behavior resembles a pipeline: `find` passes its result to another program. These two mechanisms look similar, but their internals differ. Choose an appropriate one depending on your task.

Let's look at how the `find` utility performs the `-exec` action. The utility has a built-in interpreter. When it receives the `-exec` action on input, it calls the specified program there. The interpreter passes to the program whatever the `find` utility has found. Note that Bash is not involved in the `-exec` call. Therefore, you cannot use the following Bash features there:

- built-in Bash commands
- functions
- pipelines
- stream redirection
- conditional statements
- loops

Try to run the following command:

```
find ~ -type f -exec echo {} \;
```

The `find` utility calls the `echo` Bash built-in here. It works correctly. Why? Actually, `find` calls the `echo` utility. It has the same name as the Bash command. Unix environment provides several utilities that duplicate the Bash built-ins. You can find them in the `/bin` system directory. For example, there is the `/bin/echo` executable there.

Sometimes you need a Bash feature in the `-exec` action. There is a trick to get it. Run the shell explicitly and pass a command to it. Here is an example:

```
find ~ -type f -exec bash -c 'echo {}' \;
```

The previous command calls the `echo` utility. This command calls the `echo` Bash built-in. They do the same and print the `find` results on the screen.

Another option to process the `find` results is applying pipeline. Here is an example:


```
find ~ -type f | grep "bash"
```

The command's output looks like this:

```
1 /home/ilya.shpigor/.bashrc
2 /home/ilya.shpigor/.bash_history
3 /home/ilya.shpigor/.bash_logout
4 /home/ilya.shpigor/.bash_profile
```

The pipeline receives text data from the `find` utility. Then it transfers these data to the `grep` utility. Finally, `grep` prints the filenames where the pattern “bash” occurs.

The `-exec` action behaves in another way. No text data is transferred in this case. The `find` interpreter constructs a program call using the `find` results. It passes the paths of found files and directories to the program. These paths are not plain text.

You can use a pipeline and get the `-exec` action behavior. Apply the `xargs` utility for that.

Here is an example. Suppose that you want to find the pattern in the contents of the found files. The `grep` utility should receive file paths but not the plain text in this case. You can apply pipeline and `xarg` to solve this task. The solution looks like this:

```
find ~ -type f | xargs grep "bash"
```



This command cannot handle files whose names contain spaces and line breaks. The next section considers how to solve this problem.

Here is the command's output:

```
1 /home/ilya.shpigor/.bashrc:# ~/.bashrc: executed by bash(1) for interactive shells.
2 /home/ilya.shpigor/.bashrc:# The copy in your home directory (~/.bashrc) is yours, p\
3 lease
4 /home/ilya.shpigor/.bashrc:# User dependent .bashrc file
5 /home/ilya.shpigor/.bashrc:# See man bash for more options...
6 /home/ilya.shpigor/.bashrc:# Make bash append rather than overwrite the history on d\
7 isk
8 /home/ilya.shpigor/.bashrc:# When changing directory small typos can be ignored by b\
9 ash
10 ...
```

The `xargs` utility constructs a command. The utility takes two things on the input: parameters and text data from the input stream. The parameters come in the first place in the constructed command. Then all data from the input stream follows.

Let's come back to our example. Suppose that the `find` utility found the `~/ .bashrc` file. The pipeline passes the file path to the following `xargs` call:

```
xargs grep "bash"
```

The `xargs` utility receives two parameters in this call: “grep” and “bash”. Therefore, it constructs the command that starts with these two words. Here is the result:

```
grep "bash"
```

Then `xargs` do the second step. It takes text data from the input stream and converts them to the parameters for the constructed command. The pipeline passes the `~/ .bashrc` path to `xargs`. The utility uses it for making this command:

```
grep "bash" ~/ .bashrc
```

The `xargs` utility does not invoke Bash for executing the constructed command. It means that the command has the same restrictions as the `-exec` action of the `find` utility. No Bash built-ins and features are allowed there.

The `xargs` utility places the parameters made from the input stream at the end of the constructed command. In some cases, you want to change their position. For example, you want to place the parameters in the middle of the command. The `-I` option of `xargs` does that.

Here is an example. Suppose that you want to copy the found HTML files to the home directory. The `cp` utility does it. The only task is placing its parameters in the proper order when constructing the `cp` call. Use the `-I` option of `xargs` this way to get it:

```
find /usr/share/doc/bash -type f -name "*.html" | xargs -I % cp % ~
```

When you apply the `-I` option, you specify the place to insert parameters by the percent sign `%`. You can replace the percent sign with any string. Here is an example:

```
find /usr/share/doc/bash -type f -name "*.html" | xargs -I FILE cp FILE ~
```

The `xargs` utility receives several lines via the pipeline. It constructs the `cp` call per each received line. The utility creates the following two commands in our example:

- 1 `cp /usr/share/doc/bash/bash.html /home/ilya.shpigor`
- 2 `cp /usr/share/doc/bash/bashref.html /home/ilya.shpigor`

The `-t` option of `xargs` displays the constructed commands before executing them. Use it for checking the utility’s results and debugging. Here is an example of applying the option:

```
find /usr/share/doc/bash -type f -name "*.html" | xargs -t -I % cp % ~
```

We have considered several cases of using `find` with a pipeline. These examples are for educational purposes only. Do not apply them in practice! Use the `-exec` action of the `find` utility instead of pipelines. This way, you avoid issues when processing filenames with spaces and line breaks.

There are very few cases when combining `find` and a pipeline makes sense. One of these cases is the parallel processing of found files.

Here is an example. When you call the `cp` utility in the `-exec` action, it copies files one by one. It is inefficient if your CPU has several cores and the hard disk has a high access speed. You can speed up the operation by running it in several parallel processes. The `-P` parameter of the `xargs` utility does that. Specify the number of the processes in this parameter. They will execute the constructed command in parallel.

Suppose your computer's processor has four cores. Then you can copy files in four parallel processes. The following command does it:

```
find /usr/share/doc/bash -type f -name "*.html" | xargs -P 4 -I % cp % ~
```

This command copies four files at once. As soon as one of the parallel processes finishes, it handles the next file. This approach speeds up the processing of time-consuming tasks considerably. The performance gain depends on the configuration of your hardware.

Many GNU utilities can handle text data from the input stream. They work well in pipelines. Table 2-14 shows the most commonly used of these utilities.

Table 2-14. Utilities for processing the input stream

Utility	Description	Examples
<code>xargs</code>	It constructs a command from parameters and the input stream data.	<code>find . -type f -print0 xargs -0 cp -t ~</code>
<code>grep</code>	It searches for text that matches the specified pattern.	<code>grep -A 3 -B 3 "GNU" file.txt</code> <code>du /usr/share -a grep "\.html"</code>
<code>tee</code>	It redirects the input stream to the output stream and file at the same time.	<code>grep "GNU" file.txt tee result.txt</code>
<code>sort</code>	It sorts strings from the input stream in forward or reverse order (<code>-r</code>).	<code>sort file.txt</code> <code>du /usr/share sort -n -r</code>
<code>wc</code>	It counts the number of lines (<code>-l</code>), words (<code>-w</code>), letters (<code>-m</code>) and bytes (<code>-c</code>) in the specified file or input stream.	<code>wc -l file.txt</code> <code>info find wc -m</code>
<code>head</code>	It outputs the first bytes (<code>-c</code>) or lines (<code>-n</code>)	<code>head -n 10 file.txt</code>

Table 2-14. Utilities for processing the input stream

Utility	Description	Examples
	of a file or text from the input stream.	<code>du /usr/share sort -n -r head -10</code>
<code>tail</code>	It outputs the last bytes (-c) or lines (-n) of a file or text from the input stream.	<code>tail -n 10 file.txt</code> <code>du /usr/share sort -n -r tail -10</code>
<code>less</code>	It is the utility for navigating text from the input stream. Press the Q key to exit.	<code>less /usr/share/doc/bash/README</code> <code>du less</code>

Pipeline Pitfalls

The pipeline is a convenient Bash feature. You will apply it often when working with the shell. Unfortunately, you can easily make a mistake using the pipeline. Let's consider its pitfalls by examples.

You can expect that the same result from the following two commands:

- 1 `find /usr/share/doc/bash -name "*.html"`
- 2 `ls /usr/share/doc/bash | grep "\.html"`

These commands provide different results in some cases. The problem happens when you pass the filenames through the pipeline.

The root cause of the problem came from the POSIX standard. The standard allows all printable characters in the file and directory names. It means that spaces and line breaks are allowed. The only forbidden character is the **null character**²³³ (NULL). This rule can lead to unexpected consequences.

Here is an example. Create a file in the home directory. The filename should contain the line break. This is a control character that matches the `\n` **escape sequence**²³⁴ in the ASCII encoding. Call the `touch` utility to create the empty file this way:

```
touch ~/test\nfile.txt'
```

The `touch` utility updates the modification time of the file. It is a primary task of the utility. If the file does not exist, `touch` creates it. Such a secondary feature of a program is called the **side effect**²³⁵.

You need to create two extra files for our test. Call them `test1.txt` and `file1.txt`. The following command does that:

²³³https://en.wikipedia.org/wiki/Null_character

²³⁴https://en.wikipedia.org/wiki/Escape_sequence

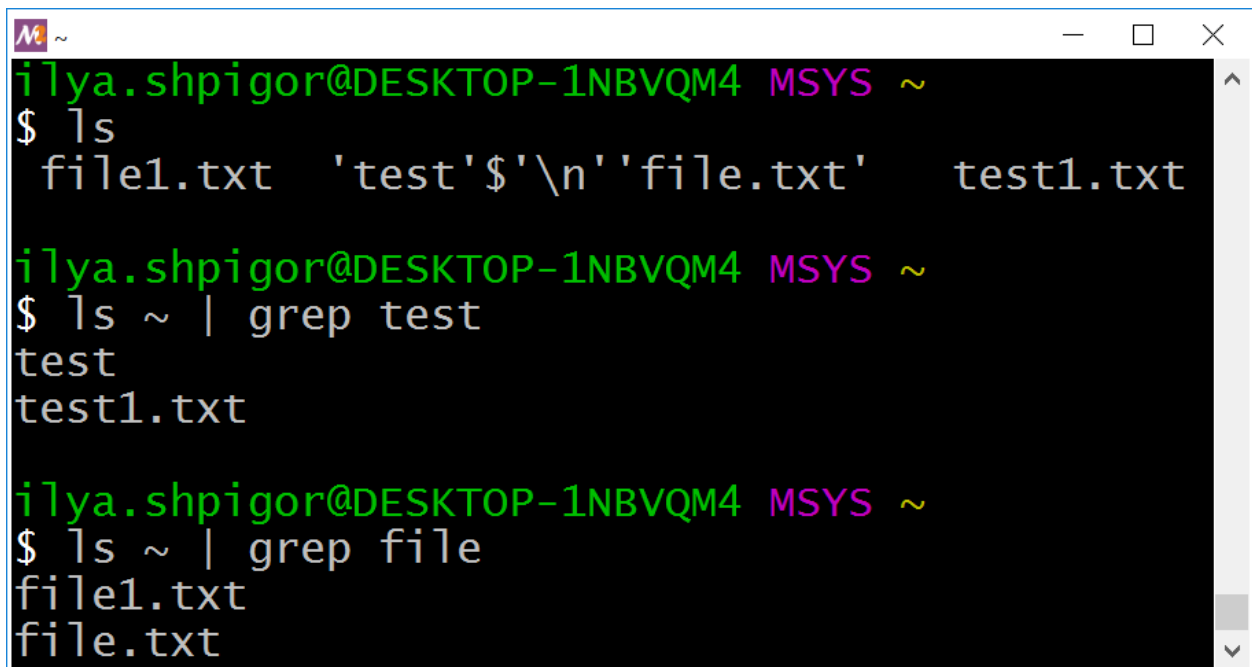
²³⁵[https://en.wikipedia.org/wiki/Side_effect_\(computer_science\)](https://en.wikipedia.org/wiki/Side_effect_(computer_science))

```
touch ~/test1.txt ~/file1.txt
```

Now call the `ls` utility for the home directory. Pass its output to `grep` using the pipeline. Here are the example commands:

- 1 `ls ~ | grep test`
- 2 `ls ~ | grep file`

Figure 2-27 shows the output of these commands.



```
ilya.shpigor@DESKTOP-1NBVQM4 MSYS ~
$ ls
file1.txt 'test'$'\n'file.txt test1.txt

ilya.shpigor@DESKTOP-1NBVQM4 MSYS ~
$ ls ~ | grep test
test
test1.txt

ilya.shpigor@DESKTOP-1NBVQM4 MSYS ~
$ ls ~ | grep file
file1.txt
file.txt
```

Figure 2-27. The result of combining the `ls` and `grep` utilities

Both commands truncate the `test\nfile.txt` filename.

Try to call `ls` without the pipeline. You will see that the utility prints the `'test'$'\n'file.txt'` filename properly. When you pass it via the pipeline, the escaping sequence `\n` is replaced by the line break. It leads to splitting the filename into two parts. Then `grep` handles these parts separately.

There is another potential problem. Suppose you search and copy the file. Its name has a space (for example, `test file.txt`). Then the following command fails:

```
ls ~ | xargs cp -t ~/tmp
```

Here `xargs` constructs the following call of the `cp` utility:

```
cp -t ~/tmp test file.txt
```

The command copies the `test` and `file.txt` files to the `~/tmp` path. However, none of these files exists. The reason for the error is the word splitting mechanism. Bash splits lines in words by the spaces. You can disable the mechanism using double quotes. Here is an example for our command:

```
ls ~ | xargs -I % cp -t ~/tmp "%"
```

It copies the “test file.txt” file properly.

Double quotes do not help if the filename has a line break. The only solution here is not to use `ls`. The `find` utility with the `-exec` action does this job right. Here is an example:

```
find . -name "*.txt" -exec cp -t tmp {} \;
```

It would be great not to use pipelines with file and directory names at all. However, you need it for solving some tasks. Combine the `find` and `xargs` utilities in this case. This approach works well if you call `find` with the `-print0` option. Here is an example:

```
find . -type f -print0 | xargs -0 -I % bsdtar -cf %.tar %
```

The `-print0` option changes the `find` output format. The default format is a list of the found objects. The separator between the objects is a line break. The `-print0` option changes the separator to the null character.

You have changed the `find` output format. Now you should change the `xargs` call accordingly. The `xarg` utility separates the input stream data by line breaks. The `-0` option changes the separator to the null character. This way, you reconcile the output and input formats of two utilities.

The `-Z` option changes the `grep` output format in the same manner. It replaces the line breaks with the null characters. Using the option, you can pass the `grep` output via the pipeline without errors. Here is an example:

```
grep -RlZ "GNU" . | xargs -0 -I % bsdtar -cf %.tar %
```

This command searches files that contain the “GNU” pattern. Then the pipeline passes the found filenames to the `xargs` utility. The utility constructs the `bsdtar` call for archiving these files.

Here are the general hints for using pipelines:

1. Be aware of spaces and line breaks when passing file and directory names via the pipeline.
2. Never process the `ls` output. Use the `find` utility with the `-exec` action instead.
3. Always use `-0` option when processing the object names by `xargs`. Pass only null character separated names to the utility.

Exercise 2-7. Pipelines and I/O streams redirection

Write a command to archive photos using the `bsdtar` utility.

If you are a Linux or macOS user, use the `tar` utility instead.

The photos are stored in the directory structure from exercises 2-6:

```
~/
  photo/
    2019/
      11/
      12/
    2020/
      01/
```

The photos of the same month should come into the same archive.

Your command should provide the following result:

```
~/
  photo/
    2019/
      11.tar
      12.tar
    2020/
      01.tar
```

Logical Operators

The pipeline allows you to combine several commands. These commands together make an algorithm with a **linear sequence**²³⁶. The computer executes actions of such an algorithm one by one without any conditions.

Suppose that you need a more complex algorithm. There, the result of the first command determines the next step. The computer does one action if the command succeeds. Otherwise, it does another action. Such a dependency is known as a **conditional algorithm**²³⁷. The pipeline does not fit in this case.

Here is an example of the conditional algorithm. You want to write a command to copy the directory. If this operation succeeds, the command writes the “OK” line in the log file. Otherwise, it writes the “Error” line there.

You can write the following command using the pipeline:

²³⁶<https://www.cs.utexas.edu/users/mitra/csSpring2017/cs303/lectures/algo.html>

²³⁷[https://en.wikipedia.org/wiki/Conditional_\(computer_programming\)](https://en.wikipedia.org/wiki/Conditional_(computer_programming))

```
cp -R ~/docs ~/docs-backup | echo "OK" > result.log
```

This command does not work properly. It writes the “OK” line to the `result.log` file regardless of the copying result. Even if the `docs` directory does not exist, you get the “OK” line. The log file reports that the operation succeeded, but it has failed in fact.

The `cp` utility result should define the `echo` output. You can get this behavior using the `&&` operator. Then the command becomes like this:

```
cp -R ~/docs ~/docs-backup && echo "OK" > result.log
```

Now `echo` prints the “OK” line when the `cp` utility succeeds. Otherwise, there is no output to the log file.

The `&&` operator performs the logical conjunction (AND). Its operands are Bash commands, which are actions. It differs from a regular AND that operates the Boolean expressions. These expressions are conditions.

Let’s have a look at how the logical conjunction deals with the Bash commands. The POSIX standard requires each program to return an **exit code**²³⁸ (or exit status) when it is done. The zero code means that the program finished successfully. Otherwise, the code takes a value from 1 to 255. Each Bash built-in returns the exit code too.

When you apply logical conjunction to the commands, the operator handles their exit codes. Bash executes the commands. Then the shell converts their exit codes into Boolean values. These values are the allowed operands of the logical operator.

Let’s go back to our example command:

```
cp -R ~/docs ~/docs-backup && echo "OK" > result.log
```

Suppose the `cp` utility completes successfully. It returns the zero code in this case. The zero code matches the “true” Boolean value. Therefore, the left-hand side (LHS) operand of the `&&` operator equals “true”. This information is not enough to deduce the result of the whole expression. It can be “true” or “false” depending on the right-hand side (RHS) operand. Then the `&&` operator has to execute the `echo` command. This command always succeeds and returns the zero code. Thus, the result of the `&&` operator equals “true”.

It is not clear how do we use the result of the `&&` operator in our example. The answer is we do not use it at all. Logical operators are needed to calculate Boolean expressions. However, they are often used in Bash for their side effect. This side effect is a strict order of operands evaluation.

Let’s consider the case when the `cp` utility fails in our example. It returns a non-zero exit code. This code is equivalent to the “false” value for Bash. In this case, the `&&` operator can deduce the value of the whole Boolean expression. It does not need to calculate the RHS operand. If at least one operand

²³⁸https://en.wikipedia.org/wiki/Exit_status

of the logical AND is “false”, the operator’s result equals “false”. Thus, the exit code of the `echo` command is not required. Then the `&&` operator does not execute it. This way, the “OK” line does not come to the log file.

We have considered the behavior of logical operators that is called the **short-circuit evaluation**²³⁹. It means calculation only those operands that are sufficient to deduce the value of the whole Boolean expression.



Bash stores an exit code of the last executed command in the environment variable. The variable name is the question mark. You can print its value with the `echo` command this way:

```
echo $?
```

We have done only the first part of our task. Now the command prints the “OK” line in the log file when copying succeeds. We need to handle the case of a failure too. If copying fails, the log file should get the “Error” line. You can add this behavior with the `||` operator, which does logical OR.

When adding the OR operator, our command looks like this:

```
cp -R ~/docs ~/docs-backup && echo "OK" > result.log || echo "Error" > result.log
```

This command implements the conditional algorithm that we need. If the `cp` utility finishes successfully, the log file gets the “OK” line. Otherwise, it gets the “Error” line. Let’s consider how it works in detail.

First, we would make our command simpler for reading. Let’s denote all operands by Latin letters. The “A” letter matches the `cp` call. The “B” letter marks the first `echo` call with the “OK” line. The “C” letter is the second `echo` call. Then we can rewrite our command this way:

```
A && B || C
```

The `&&` and `||` operators have the same priorities. Bash calculates Boolean expressions from the left to the right side. The operators are called **left-associative**²⁴⁰ in this case. Given this, we can rewrite our expression this way:

```
(A && B) || C
```

²³⁹https://en.wikipedia.org/wiki/Short-circuit_evaluation

²⁴⁰https://en.wikipedia.org/wiki/Operator_associativity

Adding parentheses does not change the calculation order. First, Bash evaluates the expression (A && B). Then, it calculates the “C” operand if it is necessary.

If “A” equals “true”, the && operator calculates its RHS operand “B”. This is the echo command that prints the “OK” line to the log file. Next, Bash processes the || operator. Its LHS operand (A && B) equals “true”. When calculating the OR operator, it is enough to get the “true” value for at least one operand. Then you can conclude that the operator’s result equals “true”. Therefore, Bash skips the RHS operand “C” in our case. It leads that the “Error” line does not come to the log file.

If the “A” value is “false”, the expression (A && B) equals “false” too. In this case, Bash skips the operand “B”. It is enough to have one operand equals “false” to deduce the “false” result of the AND operator. Then the “OK” line does not come to the log file. Next, Bash handles the || operator. The shell already knows that its LHS operand equals “false”. Thus, it should evaluate the RHS operand “C” for deducing the whole expression. It leads to executing the second echo command. Then the “Error” line comes to the log file.

The principle of short-circuits evaluation is not obvious. You would need some time to figure it out. Please do your best for that. Every modern programming language supports Boolean expressions. Therefore, understanding the rules of their evaluation is essential.

Combining Commands

We already know how to combine Bash commands with pipelines and logical operators. There is a third way to do that. You can put a semicolon as a delimiter between the commands. In this case, Bash executes them one by one from left to right without any conditions. You get the linear sequence algorithm in this case.

Here is an example. Suppose that you want to copy two directories to different target paths. A single cp call cannot do it. But you can combine two calls into one command like this:

```
cp -R ~/docs ~/docs-backup ; cp -R ~/photo ~/photo-backup
```

This command calls the cp utility twice. The second call does not depend on the result of copying the docs directory. Even if it fails, Bash copies the photo directory anyway.

You can do the same with the pipeline this way:

```
cp -R ~/docs ~/docs-backup | cp -R ~/photo ~/photo-backup
```

Here Bash executes both cp calls one by one too. It means that the linear sequence algorithm is the same for our two commands: with the semicolon and pipeline.

However, a semicolon and pipeline behave differently in general. When you use the semicolon, two commands do not depend on each other completely. When you use the pipeline, there is dependency. The output stream data of the first command comes to the input stream of the second command. In some cases, it changes the behavior of your algorithm.

Compare the following two commands:

- 1 `ls /usr/share/doc/bash | grep "README" * -`
- 2 `ls /usr/share/doc/bash ; grep "README" * -`

The `-` option of `grep` appends data from the input stream to the utility's parameters.

Figure 2-28 shows the results of both commands.

```

ilya.shpigor@DESKTOP-1NBVQM4 MSYS ~
$ ls /usr/share/doc/bash | grep "README" * -
xz.txt:      README          This file
(standard input):README

ilya.shpigor@DESKTOP-1NBVQM4 MSYS ~
$ ls /usr/share/doc/bash ; grep "README" * -
bash.html    CHANGES  FAQ       NEWS     RBASH
bashref.html COMPAT    INTRO    POSIX   README
xz.txt:      README          This file
  
```

Figure 2-28. Results of commands with pipeline and semicolon

Even the behavior of the `ls` utility differs in these two commands. When using the pipeline, `ls` prints nothing on the screen. Instead, it redirects its output to the `grep` input.

Let's consider the output of the commands. The second parameter of `grep` is the "*" pattern. It forces the utility to process all files in the current directory first. `grep` finds the "README" word in the `xz.txt` file. Then it prints this line on the screen:

```
xz.txt: README This file
```

When file processing is done, `grep` handles the input stream data from the `ls` utility. This data also contains the "README" word. Then `grep` prints the following line:

```
(standard input):README
```

This way, the `grep` utility processes two things at once:

- Files of the current directory.
- Data from the input stream.

When using the semicolon, the `ls` utility prints its result on the screen. Then Bash calls the `grep` utility. It processes all files of the current directory. Next, `grep` checks its input stream. There is no data there. This way, `grep` finds the "README" word in the `xz.txt` file only.

Exercise 2-8. Logical operators

Write the command that implements the following algorithm:

1. Copy the README file with the Bash documentation to the home directory.
2. Archive the copied ~/README file.
3. Delete the copied ~/README file.

Each step takes place only if the previous one succeeds.

Write the result of each step to the log file result.txt.

Bash Scripts

We have learned the basics of how to operate the file system using the shell. It is time to apply our knowledge and come from the standalone commands to programs. These programs written in Bash are called **scripts**. We will learn how to write them.

Development Tools

You have used the interactive mode of Bash in the previous chapter. The workflow of this mode looks like this:

1. You type a command in the terminal window.
2. The Bash process loads your command in RAM.
3. The interpreter executes the command.
4. Bash removes your command from RAM.

If you want to write a program, RAM is not the appropriate place to store it. This is a temporary memory. Whenever you shut down the computer, RAM is cleared.

When you write a program, you should store it on the disk drive. The disk drive is long-term information storage. Then you need a special program to create and edit source code files. This program is called a **source code editor**²⁴¹.

Let's consider source code editors that work well with Bash.

Source Code Editor

Potentially, you can write Bash scripts in any text editor. Even the standard Windows application called **Notepad**²⁴² can work this way. However, text editors are inconvenient for writing the source code. Notepad does not have any features for doing that. Meanwhile, these features increase your productivity significantly.

Today you can find plenty of free and proprietary source code editors. Some of them are more widespread than others. The popularity of the editor does not mean that it fits you perfectly. You should try several programs and choose one that you like.

Here there is a list of three popular source code editors. You can start with them. If no one fits you, please look for alternatives on the Internet.

²⁴¹https://en.wikipedia.org/wiki/Source-code_editor

²⁴²https://en.wikipedia.org/wiki/Microsoft_Notepad



MS Office and its open-source counterpart [LibreOffice](#)²⁴³ are not suitable for writing source code at all. They save files in the binary or [XML-like](#)²⁴⁴ format. Interpreters and compilers cannot read them. These programs expect text files on input. Text data is the format of all source code editors.

[Notepad++](#)²⁴⁵ is a fast and minimalistic source code editor. It is available for free. You can use it on Windows only. If your OS is macOS or Linux, please consider other editors. The latest Notepad++ version is available on the [official website](#)²⁴⁶.

[Sublime Text](#)²⁴⁷ is a proprietary [cross-platform](#)²⁴⁸ source code editor. Cross-platform means that the program runs on several OSes and hardware configurations. Sublime Text works well on Windows, Linux and macOS. You can use it for free without buying a license. Download the editor on the [official website](#)²⁴⁹.

[Visual Studio Code](#)²⁵⁰ is a free cross-platform source code editor from Microsoft. It works on Windows, Linux and macOS. You do not need to buy a license for using the editor. Download it on the [official website](#)²⁵¹.

All three editors have the following features for working with source code:

- [Syntax highlighting](#)²⁵².
- [Autocomplete](#)²⁵³.
- Support of commonly used [character encodings](#)²⁵⁴.

It is possible to edit the source code without these features. However, they make it easier to read and edit the program. They also help you to get used to the Bash syntax.

Launching the Editor

There are several ways to run the source code editor. The first option is using the GUI of your OS. Launch the editor via the Start menu or the desktop icon. It is the same way you run any other program.

The second option is using the command-line interface. This approach is more convenient in some cases. Here is an example of when you would need it. You call the `find` utility for searching several

²⁴³<https://en.wikipedia.org/wiki/LibreOffice>

²⁴⁴<https://en.wikipedia.org/wiki/XML>

²⁴⁵<https://en.wikipedia.org/wiki/Notepad%2B%2B>

²⁴⁶<https://notepad-plus-plus.org/downloads>

²⁴⁷https://en.wikipedia.org/wiki/Sublime_Text

²⁴⁸https://en.wikipedia.org/wiki/Cross-platform_software

²⁴⁹<https://www.sublimetext.com>

²⁵⁰https://en.wikipedia.org/wiki/Visual_Studio_Code

²⁵¹<https://code.visualstudio.com/>

²⁵²https://en.wikipedia.org/wiki/Syntax_highlighting

²⁵³<https://en.wikipedia.org/wiki/Autocomplete>

²⁵⁴https://en.wikipedia.org/wiki/Character_encoding

files. You can pass the `find` output to the source code editor input and open all found files. It is possible because most modern editors support CLI.

There are three ways to run an application in Bash:

1. By the name of the executable.
2. By the absolute path.
3. By the relative path.

The first approach is the most convenient. You need to add the installation path of the application to the `PATH` variable. Then Bash can find the program's executable when you call it.

Let's consider how to run the Notepad++ editor by the executable name. The program has the following installation path by default:

```
C:\Program Files (86)\Notepad++
```

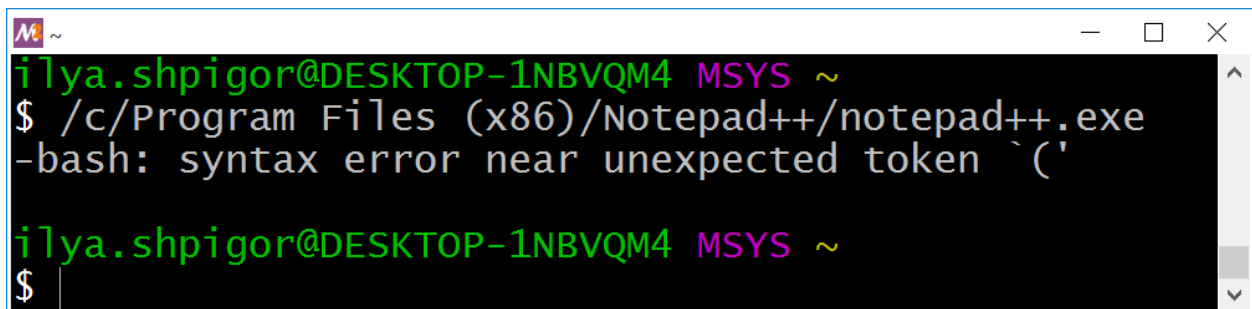


Clarify the installation path of your editor in the properties of its desktop icon or the Start menu item.

When you work in the MSYS2 environment, the Notepad++ installation path looks like this:

```
/c/Program Files (x86)/Notepad++
```

Try to run the editor using this absolute path. Figure 3-1 shows that it does not work. Bash reports about the syntax error in this case.

A screenshot of a terminal window with a black background and green text. The prompt is `ilya.shpigor@DESKTOP-1NBVQM4 MSYS ~`. The user enters the command `$ /c/Program Files (x86)/Notepad++/notepad++.exe`. The terminal outputs the error message `-bash: syntax error near unexpected token `('`. The prompt returns to `ilya.shpigor@DESKTOP-1NBVQM4 MSYS ~` and `$` is visible on the next line.

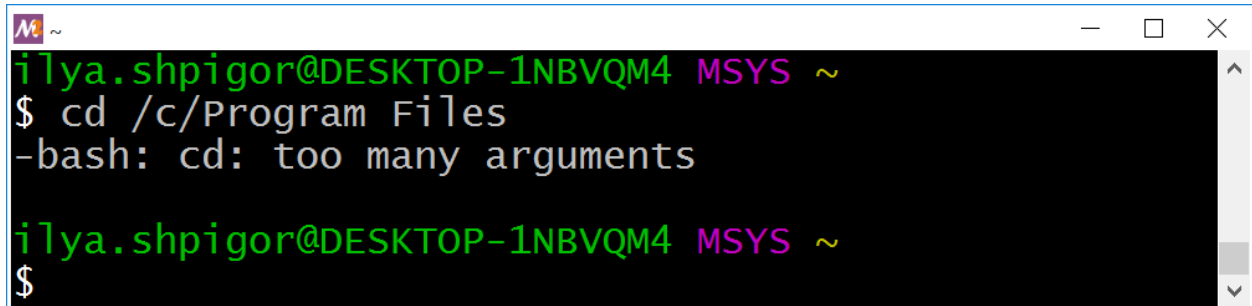
```
ilya.shpigor@DESKTOP-1NBVQM4 MSYS ~
$ /c/Program Files (x86)/Notepad++/notepad++.exe
-bash: syntax error near unexpected token `('
ilya.shpigor@DESKTOP-1NBVQM4 MSYS ~
$
```

Figure 3-1. Result of launching Notepad++

This command has several problems. We will investigate them one by one. The `cd` Bash built-in can give you the first hint about what is going wrong. Call `cd` this way:

```
cd /c/Program Files
```

Figure 3-2 shows the result.



```
ilya.shpigor@DESKTOP-1NBVQM4 MSYS ~
$ cd /c/Program Files
-bash: cd: too many arguments
ilya.shpigor@DESKTOP-1NBVQM4 MSYS ~
$
```

Figure 3-2. Result of the `cd` command

Bash complains that you have passed too many parameters to `cd`. This command expects only one parameter, which is a path. It looks like you provided two paths instead of one here. This mistake happens because of the word splitting mechanism. Bash separated the path by the space into two parts: “/c/Program” and “Files”.

You have two options to suppress the word splitting mechanism:

1. Enclose the path in double quotes:

```
cd "/c/Program Files"
```

2. Escape all spaces using the backslash:

```
cd /c/Program\ Files
```

When you suppress word splitting, Bash executes the `cd` command properly.

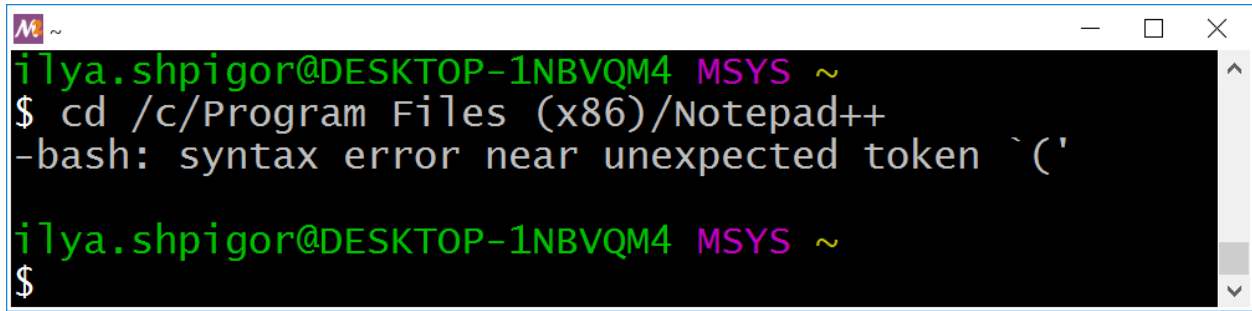
Now try to navigate the `/c/Program Files (x86)` path. The following command does not work:

```
cd /c/Program Files (x86)
```

We found out that the issue happens because of word splitting. You can suppress it by escaping the spaces this way:

```
cd /c/Program\ Files\ (x86)
```

Figure 3-3 shows that this command still fails.



```
ilya.shpigor@DESKTOP-1NBVQM4 MSYS ~
$ cd /c/Program Files (x86)/Notepad++
-bash: syntax error near unexpected token `('
ilya.shpigor@DESKTOP-1NBVQM4 MSYS ~
$
```

Figure 3-3. Result of the `cd` command

This is the same error message as Bash has printed when launching Notepad++ in Figure 3-1. This problem happens because of the parentheses. They are part of the Bash syntax. It means that the shell treats them as a language construct. We met this problem when grouping conditions of the `find` utility. Escaping or double quotes solves this issue too. Here are possible solutions for our case:

- 1 `cd /c/Program\ Files\ \ (x86\)`
- 2 `cd "/c/Program Files (x86)"`

Using double quotes is simpler than escaping. Apply them to launch the Notepad++ this way:

```
"/c/Program Files (x86)/Notepad++/notepad++.exe"
```

Now Bash launches the editor properly.

Launching Notepad++ by the absolute path is inconvenient. You should type a long command in this case. Launching the editor by the name of the executable is much better. Let's change the `PATH` Bash variable for making that work.

Add the following line at the end of the `~/ .bash_profile` file:

```
PATH="/c/Program Files (x86)/Notepad++:${PATH}"
```

Restart the MSYS2 terminal. Now the following command launches Notepad++:

```
notepad++.exe
```

There is one more option to launch the editor from the shell. Instead of changing the `PATH` variable, you can declare an **alias**. The alias is a Bash mechanism. It replaces the command you typed with another one. This way, you can abbreviate long lines.

We have the following command for launching Notepad++:

```
"/c/Program Files (x86)/Notepad++/notepad++.exe"
```

Let's declare the alias for this command. The `alias` Bash built-in does this job. Call it this way for our example:

```
alias notepad++="/c/Program\ Files\ \ (x86\)/Notepad++/notepad++.exe"
```

This command declares the alias with the “notepad++” name. Now Bash replaces the “notepad++” command by the absolute path to the Notepad++ executable.

Using the alias has one problem. You should declare it whenever launching the terminal window. There is a way to automate this declaration. Just add our `alias` command at the end of the `~/ .bashrc` file. Bash executes this file at every terminal startup. Then you get declared alias in each new terminal window.

Now you can open the source code files in Notepad++ using the shell. Here is an example to open the `test.txt` file:

```
notepad++ test.txt
```

If the `test.txt` file does not exist, Notepad++ shows you the dialog to create it.

Background Mode

Suppose that you run a GUI application in the terminal window. Then you cannot use this window for typing the Bash commands. The GUI program controls it and prints the diagnostic messages there. The terminal window becomes available again when the application finishes.

You can run the GUI application in the **background mode**. Then the terminal window stays available, and you can use it normally.

Add the ampersand `&` at the end of a Bash command to launch it in the background mode. Here is an example:

```
notepad++ test.txt &
```

After this command, you can type text in the terminal window. The only problem is the error messages from Notepad++. The editor still prints them here. They make it inconvenient to use this terminal window.

You can detach the running GUI application from the terminal window completely. Do it with the `disown` Bash built-in. Call `disown` with the `-a` option this way:

```
1 notepad++ test.txt &
2 disown -a
```

Now Notepad++ does not print any messages in the terminal. The `disown` call has one more effect. It allows you to close the terminal window and keep the editor working. Without the `disown` call, Notepad++ finishes when you close the terminal.

You can combine Notepad++ and `disown` calls into one command. It looks like this:

```
notepad++ test.txt & disown -a
```

The `-a` option of the `disown` command detaches all programs that work in the background. If you skip this option, you should specify the **process identifier**²⁵⁵ (PID) of the program to detach. PID is a unique number that OS assigns to each new process.

Suppose that you want to call `disown` for the specific program. You should know its PID. Bash prints the PID of the background process when you launch it. Here is an example:

```
notepad++ test.txt &
[1] 600
```

The second line has two numbers. The second number 600 is PID. The first number “[1]” is the job ID. You can use it to switch the background process to the foreground mode. The `fg` command does it this way:

```
fg %1
```

If you want to detach the Notepad++ process from our example, call `disown` this way:

```
disown 600
```

If you want to list all programs that work in the background, use the `jobs` Bash built-in. When you call it with the `-l` option, it prints both job IDs and PIDs. Use it this way:

```
jobs -l
```

This command lists all background processes that you have launched in the current terminal window.

You can call Notepad++ and detach it from the terminal in a single command. In this case, you should use the special Bash variable called `$_`. It stores the PID of the last launched command. Pass this PID to the `disown` call, and you are done. Here is an example of how to apply this approach:

²⁵⁵https://en.wikipedia.org/wiki/Process_identifier

```
notepad++ test.txt & disown $!
```

Why Do We Need Scripts?

We learned how to write complex Bash commands using pipelines and logical operators. The pipeline combines several commands into one. You get a linear sequence algorithm this way. If you add logical operators there, you get the conditional algorithm. These operators allow you to handle special cases and choose a proper reaction for them.

The shell command that implements the conditional algorithm can be as complicated as a real program. What is the difference between them? Why do we need scripts that are Bash programs? Let's figure out answers to these questions.

Backup Command

We need an example to consider Bash scripts features. Let's write the command that creates a backup of your photos on the [external hard drive](#)²⁵⁶. The command consists of two actions: archiving and copying.

Suppose that you store all your photos in the `~/photo` directory. The mount point of the external drive is `/d`. Then the following command creates an archive of the photos on the external drive:

```
bsdtar -cjf ~/photo.tar.bz2 ~/photo && cp -f ~/photo.tar.bz2 /d
```

Here the logical AND connects the archiving and copying commands. Therefore, the `cp` call happens only when the `bsdtar` utility succeeds. This utility creates an archive called `photo.tar.bz2`. It contains all files of the `~/photo` directory.



In our example, creating the backup happens in two steps: archiving and copying. It is done for demonstration purposes. You can do the same thing by the single `bsdtar` call:

```
bsdtar -cjf /d/photo.tar.bz2 ~/photo
```

Suppose that you run the backup command automatically. For example, it launches every day by schedule. If some error happens, you do not have a chance to read its message. You need a log file to get this possibility. Here is an example of the `bsdtar` call that writes its status to the file:

²⁵⁶https://en.wikipedia.org/wiki/Hard_disk_drive#EXTERNAL

```

1 bsdtar -cjf ~/photo.tar.bz2 ~/photo &&
2 echo "bsdtar - OK" > results.txt ||
3 echo "bsdtar - FAILS" > results.txt

```

You can split a Bash command into multiple lines. There are two ways for doing that:

1. Add the line break right after the logical operator (&& or ||).
2. Add the line break after the backslash .

We applied the first option in the last `bsdtar` call. The second option looks like this:

```

1 bsdtar -cjf ~/photo.tar.bz2 ~/photo \
2 && echo "bsdtar - OK" > results.txt \
3 || echo "bsdtar - FAILS" > results.txt

```

You would need the status of the `cp` call as well. Therefore, we should write it to the log file. Here is the command for that:

```

1 cp -f ~/photo.tar.bz2 /d &&
2 echo "cp - OK" >> results.txt ||
3 echo "cp - FAILS" >> results.txt

```

Now we can combine the `bsdtar` and `cp` calls into a single command. The logical AND should connect these calls. The straightforward solution looks like this:

```

bsdtar -cjf ~/photo.tar.bz2 ~/photo &&
  echo "bsdtar - OK" > results.txt ||
  echo "bsdtar - FAILS" > results.txt &&
cp -f ~/photo.tar.bz2 /d &&
  echo "cp - OK" >> results.txt ||
  echo "cp - FAILS" >> results.txt

```

Let's check if this command works correctly. We can replace each command call with a Latin letter. Then we get a convenient form of the Boolean expression. The expression looks like this:

```
B && O1 || F1 && C && O2 || F2
```

The “B” and “C” letters represent the `bsdtar` and `cp` calls. “O1” is the `echo` call that prints “bsdtar - OK” line in the log file. “F1” is the `echo` call for printing “bsdtar - FAIL” line. Similarly, “O2” and “F2” are the commands for logging the `cp` result.

If the `bsdtar` call succeeds, the “B” operand of our expression equals “true”. Then Bash performs the sequence of the following steps:

1. B
2. O1
3. C
4. O2 or F2

If the `bsdtar` fails, the “B” operand equals false. Then Bash does the following steps:

1. B
2. F1
3. C
4. O2 or F2

It means that the shell calls the `cp` utility even when the archiving fails. It does not make sense.

Unfortunately, the `bsdtar` utility makes things even more confusing. It creates an empty archive if it cannot access the target directory or files. Then the `cp` utility copies the empty archive successfully. These operations lead to the following output in the log file:

```
1 bsdtar - FAILS
2 cp - OK
```

Such output confuses you. It does not clarify what went wrong.

Here is our expression again:

```
B && O1 || F1 && C && O2 || F2
```

Why does Bash call the `cp` utility when `bsdtar` fails? It happens because the `echo` command always succeeds. It returns zero code, which means “true”. Thus, the “O1”, “F1”, “O2” and “F2” operands of our expression are always “true”.

Let’s fix the issue caused by the `echo` call exit code. We should focus on the `bsdtar` call and corresponding `echo` commands. They match the following Boolean expression:

```
B && O1 || F1
```

We can enclose the “B” and “O1” operands in brackets this way:

```
(B && O1) || F1
```

It does not change the expression’s result.

We got a logical OR between the “(B && O1)” and “F1” operands. The “F1” operand always equals “true”. Therefore, the whole expression is always “true”. The value of “(B && O1)” does not matter. We want to get another behavior. If the “(B && O1)” operand equals “false”, the entire expression should be “false”.

One possible solution is inverting the “F1” operand. The logical NOT operator does that. We get the following expression this way:

```
B && O1 || ! F1 && C && O2 || F2
```

Let's check the behavior that we got. If the "B" command fails, Bash evaluates "F1". It always equals "false" because of negation. Then Bash skips the "C" and "O2" commands. It happens because there is a logical AND between them and "F1". Finally, Bash comes to the "F2" operand. The shell needs its value. Bash knows that the LHS operand of the logical OR equals "false". Therefore, it needs to evaluate the RHS operand to deduce the result of the whole expression.

We can make the expression clearer with the following parentheses:

```
(B && O1 || ! F1 && C && O2) || F2
```

Now it is evident that Bash executes the "F2" action when the parenthesized expression equals "false". Otherwise, it cannot deduce the final result.

The last command writes this output into the log file:

```
1 bsdtar - FAILS
2 cp - FAILS
```

This output looks better than the previous one. Now the `cp` utility does not copy an empty archive.

The current result still has room for improvement. Imagine that you extended the backup command. Then it contains 100 actions. If an error occurs at the 50th action, all the remaining operations print their failed results into the log file. Such output makes it complicated to find the problem. The better solution here is to terminate the command right after the first error occurred. Parentheses can help us to reach this behavior. Here is a possible grouping of the expression's operands:

```
(B && O1 || ! F1) && (C && O2 || F2)
```

Let's check what happens if the "B" operand is false. Then Bash executes the "F1" command. The negation inverts the "F1" result. Therefore, the entire LHS expression equals "false". Here is the LHS expression:

```
(B && O1 || ! F1)
```

Then the short-circuit evaluation happens. It prevents calculating the RHS operand of the logical AND. Then Bash skips all commands of the RHS expression. Here is the RHS expression:

```
(C && O2 || F2)
```

We got the proper behavior of the backup command.

We can add one last improvement. The "F2" operand should be inverted. Then the whole expression equals "false" if the "C" command fails. Then the entire backup command fails if `bsdtar` or `cp` call fails. Inverting "F2" operand provides the proper non-zero exit status in the error case.

Here is the final version of our expression with all improvements:

```
(B && 01 || ! F1) && (C && 02 || ! F2)
```

Let's come back to the real Bash code. The corrected backup command looks like this:

```
1 (bsdtar -cjf ~/photo.tar.bz2 ~/photo &&
2   echo "bsdtar - OK" > results.txt ||
3   ! echo "bsdtar - FAILS" > results.txt) &&
4 (cp -f ~/photo.tar.bz2 /d &&
5   echo "cp - OK" >> results.txt ||
6   ! echo "cp - FAILS" >> results.txt)
```

We spent some time writing this command. However, another person would need much more time to read it and understand it correctly. It happens in programming often. This situation is a severe problem for big projects. Therefore, please train yourself to make your code clean and evident from the beginning. Code cleanliness is more important than a high speed of writing it.

Poor Technical Solution

Our backup command became long and complex after applying all improvements. Therefore, you should store it somewhere. Otherwise, you have to type the command in the terminal window each time. Typing is a bad idea because you can make a mistake or forget something.

Bash has an option to store frequently used commands. The history file saves everything you executed in the terminal. The file is unique for each user and has the `~ / .bash_history` path. When you press the Ctrl+R keystroke in the terminal window, Bash calls the quick search over the history. You can quickly find the required command there.

Can you store the backup command permanently in the history file? This solution seems to be reliable and convenient. Please do not jump to conclusions. Let's take a look at its possible problems.

First, the history file has a limited size. It saves 500 most recently executed commands by default. When this number exceeds, each new command overwrites the oldest one in the file. Therefore, you can lose the backup command accidentally.

You can think about increasing the capacity of the history file. Then the question arises. Which size would be enough? Whatever size you choose, there is a risk of exceeding it. This problem leads to the idea of making the history file unlimited. Then it saves all commands without overwriting anything.

It seems you find a way to store the backup command effectively. The history file with unlimited size does it. Could this decision lead to any problems?

Suppose you use Bash for a couple of years. All commands you executed during this time came to the `.bash_history` file. If you run the same command twice, it appears twice in the file. Therefore, the history size will reach hundreds of megabytes in two years. You do not need most of these commands. Only a small portion of them are significant for regular usage. It leads to inefficient use of your disk drive space.

You might argue that storing two hundred megabytes of the history file is not a problem for modern computers. Yes, it is true. However, there is another overhead that you missed. When you press Ctrl+R, Bash searches the command in the entire `.bash_history` file. The larger it is, the longer the search takes. Over time, you will wait several seconds, even using a powerful computer.

When the history file grows, the searching time increases. There are two reasons for that. First, Bash should process more lines in the history to find your request. Second, the file has many commands that have the same first letters. It leads you to type more letters after pressing Ctrl+R to find the right command. At some point, the history file search becomes inconvenient. That is the second problem with our solution.

What else could go wrong? Suppose that you got new photos. You placed them in the `~/Documents/summer_photo` directory. Our backup command can handle the `~/photo` path only. It cannot archive files from `~/Documents/summer_photo`. Thus, you should write a new command for doing that. The complexity of extending features is the third problem of our solution.

You may have several backup commands for various purposes. The first one archives your photos. The second one archives your documents. It would be hard to combine them. You have to write the third command that includes all required actions.

We can conclude that a history file is a bad option for the long-term storage of commands. There is the same reason for all our problems. We misuse the history file mechanism. It was not intended for storing information permanently. As a result, we came up with a poor technical solution.

Everybody can come up with a poor technical solution. Professionals with extensive experience did such a mistake often. It happens for various reasons. The lack of knowledge played a role in our case. We got how Bash works in the shell mode. Then we applied this experience to the new task. The problem happened because we did not take into account all the requirements.

Here is the complete list of the requirements for our task:

1. The backup command should have a long-term storage.
2. It should be a way to call the command quickly.
3. It should be a possibility to extend the command by new features.
4. The command should be able to combine with other commands.

First, let's evaluate our knowledge of Bash. They are not enough to meet all these requirements. All the mechanisms we know do not fit here. Can a Bash script help us? I propose to explore its features. Then we can check if it is suitable for our task.

Bash Script

Let's create a Bash script that does our backup command. Here are the steps for doing that:

1. Open the source code editor and create a new file. If you have integrated Notepad++ into Bash, run the following command:

```
notepad++ ~/photo-backup.sh
```

2. Copy the backup command into the file:

```
1 (bsdtar -cjf ~/photo.tar.bz2 ~/photo &&  
2   echo "bsdtar - OK" > results.txt ||  
3   ! echo "bsdtar - FAILS" > results.txt) &&  
4 (cp -f ~/photo.tar.bz2 /d &&  
5   echo "cp - OK" >> results.txt ||  
6   ! echo "cp - FAILS" >> results.txt)
```

3. Save the file in the home directory with the photo-backup.sh name.
4. Close the editor.

Now you have the Bash script file. Call the Bash interpreter and pass the script name there in the first parameter. Here is an example of this command:

```
bash photo-backup.sh
```

You have run your first script. Any script is a sequence of Bash commands. The file on the disk drive stores them. When Bash runs a script, it reads and executes the file commands one by one. Conditional and loop statements can change this order of execution.

It is inconvenient to call Bash interpreter explicitly when running the script. Instead, you can specify its relative or absolute path. This approach works if you do the following steps to prepare the script:

1. Allow any user to execute the script by the following command:

```
chmod +x ~/photo-backup.sh
```

2. Open the script in an editor.
3. Add the following line at the beginning of the file:

```
#!/bin/bash
```

4. Save the modified file.
5. Close the editor.

Now you can run the script by its relative or absolute path. Do it in one of the following ways:

```
1 ./photo-backup.sh
2 ~/photo-backup.sh
```

Let's consider our preparation steps for launching the script. The first thing that prevents it from running is permissions. When you create a new file, it gets the following permissions by default:

```
-rw-rw-r--
```

This line means that the owner and his group can read and modify the file. Everyone else can only read it. No one can execute the file.

The `chmod` utility changes the permissions of the specified file. If you call it with the `+x` option, the utility allows everyone to execute the file. It gets the following permissions in this case:

```
-rwxrwxr-x
```

When you run the script, your shell tries to interpret its lines. You may switch your shell from Bash to another one. It can be the [Csh](#)²⁵⁷ for example. In this case, you cannot execute our script. It happens because Bash and Csh have different syntax. They use different language constructions for the same things. We wrote the script in the Bash language. Therefore, the Bash interpreter should execute it.

There is an option to specify the interpreter that should execute the script. To do that, add the **shebang**²⁵⁸ at the beginning of the script file. Shebang is a combination of the number sign and exclamation mark. It looks like this:

```
#!
```

Add the absolute path to the interpreter after the shebang. It looks like this in our case:

```
#!/bin/bash
```

Now the Bash interpreter always executes the script. It happens even if you use another shell for typing commands.

The `file` utility prints the type of the specified file. If the script does not have the shebang, the utility defines it as a regular text file. Here is an example output:

```
~/photo-backup.sh: ASCII text
```

If you add the shebang, the utility defines this file as the Bash script:

²⁵⁷<https://ru.wikipedia.org/wiki/Csh>

²⁵⁸[https://en.wikipedia.org/wiki/Shebang_\(Unix\)](https://en.wikipedia.org/wiki/Shebang_(Unix))

~/photo-backup.sh: Bourne-Again shell script, ASCII text executable

The Bash interpreter has the same path `/bin/bash` for most Linux systems. However, this path differs for some Unix systems (for example, FreeBSD). It can be a reason why your script does not work there. The following shebang solves this problem:

```
#!/usr/bin/env bash
```

Here we call the `env` utility. It searches the path of the Bash executable in the list of the `PATH` variable.

Commands Sequence

Listing 3-1 demonstrates the current version of our script.

Listing 3-1. The script for making the photos backup

```
1 #!/bin/bash
2 (bsdtar -cjf ~/photo.tar.bz2 ~/photo &&
3   echo "bsdtar - OK" > results.txt ||
4   ! echo "bsdtar - FAILS" > results.txt) &&
5 (cp -f ~/photo.tar.bz2 /d &&
6   echo "cp - OK" >> results.txt ||
7   ! echo "cp - FAILS" >> results.txt)
```

The script contains one command, which is too long. This makes it hard to read and modify. You can split the command into two parts. Listing 3-2 shows how it looks like.

Listing 3-2. The script with two commands

```
1 #!/bin/bash
2
3 bsdtar -cjf ~/photo.tar.bz2 ~/photo &&
4   echo "bsdtar - OK" > results.txt ||
5   ! echo "bsdtar - FAILS" > results.txt
6
7 cp -f ~/photo.tar.bz2 /d &&
8   echo "cp - OK" >> results.txt ||
9   ! echo "cp - FAILS" >> results.txt
```

Unfortunately, the behavior of the script has changed. Now the logical AND does not take place between the `bsdtar` and `cp` commands. Therefore, Bash always tries to copy files even if archiving has failed. This is wrong.

The script should stop if the `bsdtar` call fails. We can reach this behavior with the `exit` Bash built-in. It terminates the script when called. The command receives the exit code as the parameter. The script returns this code on termination.

Listing 3-3 shows the script with the `exit` call.

Listing 3-3. The script with the `exit` call

```
1 #!/bin/bash
2
3 bsdtar -cjf ~/photo.tar.bz2 ~/photo &&
4   echo "bsdtar - OK" > results.txt ||
5   (echo "bsdtar - FAILS" > results.txt ; exit 1)
6
7 cp -f ~/photo.tar.bz2 /d &&
8   echo "cp - OK" >> results.txt ||
9   ! echo "cp - FAILS" >> results.txt
```

We changed the command that calls the `bsdtar` utility. It looked like this before:

```
B && O1 || ! F1
```

It became like this after adding the `exit` call:

```
B && O1 || (F1 ; E)
```

The “E” letter means the `exit` command here.

If `bsdtar` returns an error, Bash evaluates the RHS operand of the logical OR. It is equal to “(F1; E)”. We removed the negation of the `echo` command because its result is not necessary anymore. Bash calls `exit` after `echo`. We expect that this call terminates the script.

Unfortunately, the `exit` call does not terminate the script. It happens because parentheses create a [child process](https://en.wikipedia.org/wiki/Child_process)²⁵⁹. The child Bash process is called **subshell**. It executes the commands specified in parentheses. When they are done, Bash continues executing the parent process. The parent process is the one that spawned the subshell.

The `exit` call finishes the subshell in Listing 3-3. Bash calls the `cp` utility after that. To solve this problem, you should replace the parentheses with braces. Bash executes the commands in braces in the current process. The subshell is not spawned in this case.

Listing 3-4 shows the corrected version of the script.

²⁵⁹https://en.wikipedia.org/wiki/Child_process

Listing 3-4. The fixed script with the exit call

```
1 #!/bin/bash
2
3 bsdtar -cjf ~/photo.tar.bz2 ~/photo &&
4   echo "bsdtar - OK" > results.txt ||
5   { echo "bsdtar - FAILS" > results.txt ; exit 1 ; }
6
7 cp -f ~/photo.tar.bz2 /d &&
8   echo "cp - OK" >> results.txt ||
9   ! echo "cp - FAILS" >> results.txt
```

Notice the semicolon before the closing brace. It is mandatory here. Also, spaces after the opening brace and before the closing one are required.

Our problem has another solution. It is more elegant than calling the `exit` command. Suppose you want to terminate the script after the first failed command. The `set` Bash built-in can do that. It changes the parameters of the interpreter. Call the command with the `-e` option like this:

```
set -e
```

You can specify the same option when starting the Bash. Do it this way:

```
bash -e
```

The `-e` option has [several pitfalls](#)²⁶⁰. For example, it changes the behavior of the current Bash process only. The subshells it spawns work as usual.

Bash executes each command of a pipeline or logical operator in a separate subshell. Therefore, the `-e` option does not affect these commands. It means that the `set` command does not work well in our case.

Changing Parameters

Suppose you have moved your photos from the `~/photo` directory to `~/Documents/Photo`. If you want to support the new path in the backup script, you should change its code. Listing 3-5 shows how the new script looks like.

²⁶⁰<http://mywiki.woledge.org/BashFAQ/105>

Listing 3-5. The script with the new path

```
1 #!/bin/bash
2
3 bsdtar -cjf ~/photo.tar.bz2 ~/Documents/Photo &&
4   echo "bsdtar - OK" > results.txt ||
5   { echo "bsdtar - FAILS" > results.txt ; exit 1 ; }
6
7 cp -f ~/photo.tar.bz2 /d &&
8   echo "cp - OK" >> results.txt ||
9   ! echo "cp - FAILS" >> results.txt
```

Every time you move the photos from one directory to another, you have to change the script. It is inconvenient. A better solution would be to make a universal script that can handle any directory. Such a script should receive the path to photos as an input parameter.

When you run a Bash script, you can pass command-line parameters there. It works the same way as for any GNU utility. Specify the parameters separated by a space after the script name. Bash will pass them to the script. Here is an example:

```
./photo-backup.sh ~/Documents/Photo
```

This command runs our script with the `~/Documents/Photo` input parameter. You can read it via the `$1` variable in the script. If the script receives more parameters, read them via the variables `$2`, `$3`, `$4`, etc. These names match the numbers of the parameters. Variables of this type are called **positional parameters**.

There is a special positional parameter `$0`. It stores the path to the launched script. It equals `./photo-backup.sh` in our example.

Let's handle the input parameter in our script. Listing 3-6 shows how it looks like after the change.

Listing 3-6. The script uses the positional parameter

```
1 #!/bin/bash
2
3 bsdtar -cjf ~/photo.tar.bz2 "$1" &&
4   echo "bsdtar - OK" > results.txt ||
5   { echo "bsdtar - FAILS" > results.txt ; exit 1 ; }
6
7 cp -f ~/photo.tar.bz2 /d &&
8   echo "cp - OK" >> results.txt ||
9   ! echo "cp - FAILS" >> results.txt
```

The `$1` variable stores the path to the photos. We use it in the `bsdtar` call. There are double quotes around the variable name. They prevent the word splitting mechanism.

Suppose you want to archive photos from the `~/photo` album path. Then you call the script this way:

```
./photo-backup.sh "~/photo album"
```

Suppose that you skip quotes around the variable name when calling in the script. Then the `bsdtar` call looks like this:

```
bsdtar -cjf ~/photo.tar.bz2 ~/photo album &&  
echo "bsdtar - OK" > results.txt ||  
{ echo "bsdtar - FAILS" > results.txt ; exit 1 ; }
```

In this case, the `bsdtar` utility receives the `~/photo album` string in parts. It gets two parameters instead of one: `~/photo` and `album`. These directories do not exist. Therefore, the script fails.

It is not enough to put parameters in quotes when calling a script. You should quote all occurrences of the corresponding variable name in the script. It happens because of the way how the Bash runs a program.

Suppose that you call a script from the shell. Then Bash spawns a child process to execute it. The child process does not receive quotes from the command line because Bash removes them. Therefore, you should add quotes again inside the script.

Now our backup script can handle the input parameter. What are the benefits of this solution? It provides you a universal script for making backups. The script can process any paths and types of input files: documents, photos, videos, source code, etc.

Adding the parameter processing to our script leads to one problem. Suppose you call it twice for making backups of photos and documents this way:

- 1 ./photo-backup.sh ~/photo
- 2 ./photo-backup.sh ~/Documents

The first command creates the `~/photo.tar.bz2` archive and copies it to the D disk. Then the second command does the same and overwrites the existing `/d/photo.tar.bz2` file. This way, you lose the result of the first command.

To solve this problem, you should pick different names for the created archive. This way, you avoid filename conflicts. The simplest approach is to name the archive the same way as the target directory with the files to backup. Listing 3-7 shows how this solution looks like.

Listing 3-7. The script with the unique archive name

```
1 #!/bin/bash
2
3 bsdtar -cjf "$1".tar.bz2 "$1" &&
4   echo "bsdtar - OK" > results.txt ||
5   { echo "bsdtar - FAILS" > results.txt ; exit 1 ; }
6
7 cp -f "$1".tar.bz2 /d &&
8   echo "cp - OK" >> results.txt ||
9   ! echo "cp - FAILS" >> results.txt
```

Now the script picks a unique name for the archive. Call it this way, for example:

```
./photo-backup.sh ~/Documents
```

This command creates the `~/Documents.tar.bz2` archive and copies it to the D disk. In this case, the filename does not conflict with the photo archive called `/d/photo.tar.bz2`.

You can make one more improvement to the script. Call the `mv` utility instead of `cp`. It deletes the temporary archive in the home directory. Listing 3-8 shows the final version of the script.

Listing 3-8. The script with removing the temporary archive

```
1 #!/bin/bash
2
3 bsdtar -cjf "$1".tar.bz2 "$1" &&
4   echo "bsdtar - OK" > results.txt ||
5   { echo "bsdtar - FAILS" > results.txt ; exit 1 ; }
6
7 mv -f "$1".tar.bz2 /d &&
8   echo "cp - OK" >> results.txt ||
9   ! echo "cp - FAILS" >> results.txt
```

Now we get the universal backup script. Its old name `photo-backup.sh` does not fit anymore. The new version can copy any data. Let's rename it to `make-backup.sh`.

Combination with Other Commands

At the moment, you can run our backup script by its absolute or relative path. If you integrate it into Bash, you can call it by the name. This is a convenient option when you use the script in pipelines or logical operators.

These are three ways to integrate some script into Bash:

1. Add the script's path to the `PATH` variable. Edit the `~/.bash_profile` file for that.
2. Define the alias with an absolute path to the script. Do that in the `~/.bashrc` file.
3. Copy the script to the `/usr/local/bin` directory. The `PATH` variable contains this path by default. If there is no such directory in your `MSYS2` environment, create it.

We have learned the first two ways when preparing your source code editor. The third way is very straightforward. You can do it on your own.



If you need to remove a declared alias, call the `unalias` Bash built-in. For example, this call removes the `make-backup.sh` alias:

```
unalias make-backup.sh
```

Suppose that you have integrated the backup script with Bash in one of three ways. Then you can launch it by name like this:

```
make-backup.sh ~/photo
```

You can combine the script with other commands using pipelines and logical operators. It works the same way as for any Bash built-in or GNU utility.

Here is an example. Suppose you need to backup all PDF documents of the `~/Documents` directory. You can find them by the following `find` call:

```
find ~/Documents -type f -name "*.pdf"
```

Then you can apply our script to archive and copy each found file. Here is the command for that:

```
find ~/Documents -type f -name "*.pdf" -exec make-backup.sh {} \;
```



There is the escaped semicolon at the end of the `find` call. It means that the action is done for each found file separately.

This command works well. It creates an archive of each PDF file and copies it to the D disk. However, this approach is inconvenient. It would be better to collect all PDF files into one archive. Let's try the following command for that:

```
find ~/Documents -type f -name *.pdf -exec make-backup.sh {} +
```

The command should pass all found files into the single `make-backup.sh` call. Unfortunately, it does not work as expected. It produces an archive with the first found PDF file only. Where are the rest of the documents? Let's take a look at the `bsdtar` call inside the script. It looks like this:

```
bsdtar -cjf "$1".tar.bz2 "$1"
```

The problem happens because we process the first positional parameter only. The `$1` variable stores it. The `bsdtar` call ignores other parameters in variables `$2`, `$3`, etc. They contain the rest results of the `find` utility. This way, we cut off all results except the first one.

If you replace the `$1` variable with `$@`, you solve the problem. Bash stores all script parameters in `$@`. The corrected `bsdtar` call looks like this:

```
bsdtar -cjf "$1".tar.bz2 "$@"
```

The `bsdtar` utility now processes all script parameters. Note that the archive name still matches the first `$1` parameter. It should be one word. Otherwise, `bsdtar` fails.

Listing 3-9 shows the corrected version of the backup script. It handles an arbitrary number of input parameters.

Listing 3-9. The script with an arbitrary number of input parameters

```
1  #!/bin/bash
2
3  bsdtar -cjf "$1".tar.bz2 "$@" &&
4    echo "bsdtar - OK" > results.txt ||
5    { echo "bsdtar - FAILS" > results.txt ; exit 1 ; }
6
7  mv -f "$1".tar.bz2 /d &&
8    echo "cp - OK" >> results.txt ||
9    ! echo "cp - FAILS" >> results.txt
```

Bash has an alternative variable for `$@`. It is called `$*`. If you put it in double quotes, Bash interprets its value as a single word. It interprets the `$@` variable as a set of words in the same case.

Here is an example to explain the difference between the `$@` and `$*` variables. Suppose you call the backup script this way:

```
make-backup.sh "one two three"
```

In the script, Bash replaces the `"$"` construct with the following word:

```
"one two three"
```

Here is the replacement for the `"$@"` construct:

```
"one" "two" "three"
```



Always use `$@` instead of `$*`. The only exception is when a single word must represent all input parameters.

Scripts Features

While solving the backup task, we considered the basic features of the Bash scripts. Let's make a summary for them.

Here are the requirements for the backup task:

1. The backup command should have a long-term storage.
2. It should be a way to call the command quickly.
3. It should be a possibility to extend the command by new features.
4. The command should be able to combine with other commands.

The final version of the `make-backup.sh` script meets all these requirements. Here are the solutions for them:

1. The hard disk stores the script file. It is long-term memory.
2. The script is easy to integrate with Bash. Then you can call it quickly.
3. The script is a sequence of commands. Each one starts on a new line. You can read and edit it easily. Thanks to parameterization, you can generalize the script for solving tasks of the same type.
4. Due to integration with Bash, you can combine the script with other commands.

If your task requires any of these features, write a Bash script for that.

Variables and Parameters

We already met Bash variables several times in this book. You have learned the list of system paths in the `PATH` variable. Then you have used positional parameters in the backup script. It is time to get a good grasp on the topic.

Let's start with the meaning of the “variable” term in programming. The variable is an area of memory where some value is stored. In most cases, this is short-term memory (RAM, CPU cache and registers).

The first generation of programming languages (for example, [assembler](#)²⁶¹) has minimal support of variables. When using such a language, you should refer to a variable by its address. If you want to read or write its value, you have to specify its memory address.

When working with memory addresses, you might get into trouble. Suppose you work on a computer with 32-bit processors. Then any memory address has a length of 4 bytes. It is the number from 0 to 4294967295. This number is twice larger for 64-bit processors. It is inconvenient to remember and operate with such big numbers. That is why modern programming languages allow you to replace a variable address with its name. A compiler or interpreter translates this name into a memory address automatically. These programs “remember” large numbers instead of you this way.

When should you apply variables? Our experience with `PATH` and positional parameters has shown that variables store some data. It is needed for one of the following purposes:

1. Transfer information from one part of a program or system to another.
2. Store the intermediate result of a calculation for later use.
3. Save the current state of the program or system. This state may determine its future behavior.
4. Set a constant value to be used repeatedly later.

A typical programming language has a special type of variable for each of these purposes. The Bash language follows this rule too.

Classification of variables

The Bash interpreter has two operation modes: interactive (shell) and non-interactive (scripting). Variables solve similar tasks in each mode. However, the contexts of these tasks are different. Therefore, there are more features to classify variables in Bash than in other languages.

Let’s simplify the terminology for convenience. It is not entirely correct, but it helps to avoid confusion. When we talk about scripts, we use the “variable” term. When we talk about shell and command-line arguments, we use the “parameter” term. These terms are often used synonymously.

There are four attributes for classifying variables in Bash. Table 3-1 explains them.

Table 3-1. Variable Types in Bash

Classification Attribute	Variable Types	Definition	Examples
Declaration mechanism	User-defined variables	The user sets them.	<code>filename="README.txt"</code> <code>; echo "\$filename"</code>
	Internal variables	The interpreter sets them. It needs them to work correctly.	<code>echo "\$PATH"</code>

²⁶¹https://en.wikipedia.org/wiki/Assembly_language

Table 3-1. Variable Types in Bash

Classification Attribute	Variable Types	Definition	Examples
	Special parameters	The interpreter sets them for the user. The user can read them but not write.	<code>echo "\$?"</code>
Scope ²⁶²	Environment or global variables	They are available in any instance of the interpreter. The <code>env</code> utility lists them.	<code>echo "\$PATH"</code>
	Local variables	They are available in a particular instance of the interpreter only.	<code>filename="README.txt"</code> <code>; echo "\$filename"</code>
Content type	String	It stores a string.	<code>filename="README.txt"</code>
	Integer	It stores an integer.	<code>declare -i</code> <code>number=10/2 ; echo "\$number"</code>
	Indexed array	It stores a numbered list of lines.	<code>cities=("London" "New York" "Berlin") ;</code> <code>echo "\${cities[1]}"</code> <code>cities[0]="London" ;</code> <code>cities[1]="New York"</code> <code>; cities[2]="Berlin"</code> <code>; echo "\${cities[1]}"</code>
	Associative array ²⁶³	It is a data structure with elements that are key-value pairs. Each key and value are strings.	<code>declare -A cities=(</code> <code>["Alice"]="London"</code> <code>["Bob"]="New York"</code> <code>["Eve"]="Berlin") ;</code> <code>echo "\${cities[Bob]}"</code>
Changeability	Constants	The user cannot delete them. They store values that cannot be changed.	<code>readonly</code> <code>CONSTANT="ABC" ; echo "\$CONSTANT"</code> <code>declare -r</code> <code>CONSTANT="ABC" ; echo "\$CONSTANT"</code>

²⁶²[https://en.wikipedia.org/wiki/Scope_\(computer_science\)](https://en.wikipedia.org/wiki/Scope_(computer_science))

²⁶³https://en.wikipedia.org/wiki/Associative_array

Table 3-1. Variable Types in Bash

Classification Attribute	Variable Types	Definition	Examples
	Variables	The user can delete them. They store values that can be changed.	<code>filename="README.txt"</code>

We will consider each type of variable in this section.

Declaration Mechanism

User-Defined Variables

The purpose of user-defined variables is obvious from their name. You declare them for your own purposes. Such variables usually store intermediate results of the script, its state and frequently used constants.

To declare the user-defined variable, specify its name, put an equal sign, and type its value.

Here is an example. Suppose that you want to declare a variable called `filename`. It stores the `README.txt` filename. The variable declaration looks like this:

```
filename="README.txt".
```

Spaces before and after the equal sign are not allowed. It works in other programming languages but not in Bash. For example, the following declaration causes an error:

```
filename = "README.txt"
```

Bash misinterprets this line. It assumes that you call the command with the `filename` name. Then you pass there two parameters: `=` and `"README.txt"`.

When declaring a variable, you can apply Latin letters, numbers and the underscore in its name. The name must not start with a number. Letter case is important. It means that `filename` and `FILENAME` are two different variables.

Suppose you have declared a variable `filename`. Then Bash allocates the memory area for that. It writes the `README.txt` string there. You can read this value back using the variable name. When you do that, Bash should understand your intention. If you put a dollar sign before the variable name, it would be a hint for Bash. Then it treats the word `filename` as the variable name.

When you reference the variable in a command or script, it looks like this:

```
$filename
```

Bash handles words with a dollar sign in a special way. When it encounters such a word, it runs the **parameter expansion** mechanism. The mechanism replaces all occurrences of a variable name by its value. Here is the example command:

```
cp $filename ~
```

The command looks like this after the parameter expansion:

```
cp README.txt ~
```

Bash performs nine kinds of expansions before executing each command. They are done in a strict order. Please try to remember this order. If you miss it, you can get an error.

Here is an example of a mistake that happens because of expansions order. Suppose that you manipulate the “my file.txt” file in the script. For the sake of convenience, you put the filename into a variable. Its declaration looks like this:

```
filename="my file.txt"
```

Then you use the variable in the cp call. Here is the copying command:

```
cp $filename ~
```

Bash does word splitting after the parameter expansion. They are two different expansion mechanisms. When both of them are done, the cp call looks like this:

```
cp my file.txt ~
```

This command leads to the error. Bash passes two parameters to the cp utility: “my” and “file.txt”. These files do not exist.

Another error happens if the variable’s value contains a special character. For example, you declare and use the filename variable this way:

```
1 filename="*file.txt"
2 rm $filename
```

The rm utility deletes all files ending in file.txt. The globbing mechanism causes such behavior. It happens because Bash does globbing after the parameter expansion. Then it substitutes files of the current directory whose names match the “*file.txt” pattern. It leads to unexpected results. Here is an example of the rm call that you can get this way:


```
rm report_file.txt myfile.txt msg_file.txt
```

When referencing a variable, always apply double quotes. They prevent unwanted Bash expansions. The quotes solve problems of both our examples:

```
1 filename1="my file.txt"
2 cp "$filename1" ~
3
4 filename2="*file.txt"
5 rm "$filename2"
```

Thanks to the quotes, Bash inserts the variables' values as they are:

```
1 cp "my file.txt" ~
2 rm "*file.txt"
```

We already know several [Bash expansions](#)²⁶⁴. Table 3-2 gives the full picture. It shows the complete list of expansions and their order of execution.

Table 3-2. Bash expansions

Order of Execution	Expansion	Description	Example
1	Brace Expansion ²⁶⁵	It generates a set of strings by the specified pattern with braces.	echo a{d,c,b}e
2	Tilde Expansion ²⁶⁶	Bash replaces the tilde by the value of the HOME variable.	cd ~
3	Parameter Expansion ²⁶⁷	Bash replaces parameters and variables by their values.	echo "\$PATH"
4	Arithmetic Expansion ²⁶⁸	Bash replaces arithmetic expressions by their results.	echo \$((4+3))
5	Command Substitution ²⁶⁹	Bash replaces commands with their output.	echo \$(cat README.txt)

²⁶⁴<http://www.gnu.org/savannah-checkouts/gnu/bash/manual/bash.html#Shell-Expansions>

²⁶⁵https://www.gnu.org/software/bash/manual/html_node/Brace-Expansion.html

²⁶⁶https://www.gnu.org/software/bash/manual/html_node/Tilde-Expansion.html

²⁶⁷https://www.gnu.org/software/bash/manual/html_node/Shell-Parameter-Expansion.html

²⁶⁸https://www.gnu.org/software/bash/manual/html_node/Arithmetic-Expansion.html

²⁶⁹https://www.gnu.org/software/bash/manual/html_node/Command-Substitution.html

Table 3-2. Bash expansions

Order of Execution	Expansion	Description	Example
6	Process Substitution ²⁷⁰	Bash replaces commands with their output. Unlike Command Substitution, it is done asynchronously ²⁷¹ . The command's input and output are bound to a temporary file.	<code>diff <(sort file1.txt) <(sort file2.txt)</code>
7	Word Splitting ²⁷²	Bash splits command-line arguments into words and passes them as separate parameters.	<code>cp file1.txt file2.txt</code> <code>~</code>
8	Filename Expansion ²⁷³ (globbing)	Bash replaces patterns with filenames.	<code>rm ~/delete/*</code>
9	Quote Removal	Bash removes all unshielded characters , ' " that were not derived from one of the expansions.	<code>cp "my file.txt" ~</code>

Exercise 3-1. Testing the Bash expansions

Run the example of each Bash expansion from Table 3-2 in the terminal. Figure out how the final command turned out. Come up with your own examples.

Let's come back to the parameter expansion. When you put the dollar sign before a variable name, you use the short form of the expansion. Its full form looks this way:

```
${filename}
```

Use this form to avoid ambiguity. Ambiguity can happen when the text follows the variable name. Here is an example of such a case:

²⁷⁰https://www.gnu.org/software/bash/manual/html_node/Process-Substitution.html

²⁷¹[https://en.wikipedia.org/wiki/Asynchrony_\(computer_programming\)](https://en.wikipedia.org/wiki/Asynchrony_(computer_programming))

²⁷²https://www.gnu.org/software/bash/manual/html_node/Word-Splitting.html

²⁷³https://www.gnu.org/software/bash/manual/html_node/Filename-Expansion.html

```

1 prefix="my"
2 name="file.txt"
3 cp "$prefix_$name" ~

```

Here Bash tries to find and insert the variable called “prefix_”. It happens because the interpreter appends the underscore to the variable name. You can solve this kind of problem if you apply the full form of the parameter expansion. Do it this way:

```
cp "${prefix}_${name}" ~
```

If you prefer to use the short form of the expansion, you have another option. Enclose each variable name in double quotes. Then Bash will not confuse them and nearby text. Here is an example:

```
cp "$prefix"_"$name" ~
```

The full form of the parameter expansion has several features. They help you to handle cases when a variable is undefined. For example, you can insert the specified value in this case. Here is an example:

```
cp file.txt "${directory:-~}"
```

Here Bash checks if the `directory` variable is defined and has a non-empty value. If it is, Bash performs a regular parameter expansion. Otherwise, it inserts the value that follows the minus character. It is the home directory path in our example.

Table 3-3 shows all variations of the parameter expansion.

Table 3-3. The full form of the parameter expansion

Variation	Description
<code>\${parameter:-word}</code>	If the “parameter” variable is not declared or has an empty value, Bash inserts the specified “word” value instead. Otherwise, it inserts the variable’s value.
<code>\${parameter:=word}</code>	If a variable is not declared or has an empty value, Bash assigns it the specified “word” value. Then it inserts this value. Otherwise, Bash inserts the variable’s value. You cannot override positional and special parameters this way.
<code>\${parameter:?word}</code>	If the variable is not declared or has an empty value, Bash prints the specified “word” value in the error stream. Then, it terminates the script with a non-zero exit status. Otherwise, Bash inserts the variable’s value.
<code>\${parameter:+word}</code>	If the variable is not declared or has an empty value, Bash skips the expansion. Otherwise, it inserts the specified “word” value.

Exercise 3-2. The full form of the parameter expansion

Write a script that searches for files with the TXT extension in the current directory.

The script ignores subdirectories.

Copy or move all found files to the home directory.

When calling the script, you can choose whether to copy or move the files.

If you do not specify the action, the script copies the files.

Internal Variables

You can declare variables for your own purposes. Bash also can do that. These variables are called **internal** or **shell variables**. You can change values for some of them.

Internal variables solve two tasks:

1. They pass information from the shell to the applications it runs.
2. They store the current state of the interpreter.

There are two groups of internal variables:

1. Bourne Shell variables.
2. Bash variables.

The first group came from Bourne Shell. Bash needs it for compatibility with the POSIX standard. Table 3-4 shows the frequently used variables of this group.

Table 3-4. Bourne Shell variables

Name	Value
HOME	The home directory of the current user. Bash uses this variable for tilde expansion and processing the <code>cd</code> call without parameters.
IFS ²⁷⁴	It contains a list of delimiter characters. The word splitting mechanism uses them to split the strings into words. The default delimiters are space, <code>tab</code> ²⁷⁵ and a line break.
PATH	It contains the list of paths where Bash looks for utilities and programs. Colons separate the paths in the list.
PS1	It is a command prompt. The prompt can include control characters ²⁷⁶ . Bash replaces them with specific values (for example, the current user's name).
SHELLOPTS	The list of shell options ²⁷⁷ . They change the operating mode of the interpreter. Colons separate the options in the list.

²⁷⁴<http://mywiki.woledge.org/IFS>

²⁷⁵https://en.wikipedia.org/wiki/Tab_key#Tab_characters

²⁷⁶https://www.gnu.org/software/bash/manual/html_node/Controlling-the-Prompt.html#Controlling-the-Prompt

²⁷⁷https://www.gnu.org/software/bash/manual/html_node/The-Shopt-Builtin.html#The-Shopt-Builtin

The second group of internal variables is Bash specific. Table 3-5 shows them. This list is incomplete. There are some more variables, but they are rarely used.

Table 3-5. Bash variables

Name	Value
BASH	The full path to the Bash executable file. This file corresponds to the current Bash process.
BASHOPTS	The list of Bash specific shell options ²⁷⁸ . They change the operating mode of Bash. Colons separate the options in the list.
BASH_VERSION	The version of the running Bash interpreter.
GROUPS	The list of groups to which the current user belongs.
HISTCMD	The index of the current command in history. It shows you how many items are there.
HISTFILE	The path to the file that stores the command history. The default path is <code>~/.bash_history</code> .
HISTFILESIZE	The maximum number of lines allowed in the command history. The default value is 500.
HISTSIZE	The maximum number of entries allowed in the command history. The default value is 500.
HOSTNAME	The computer name as a node of the network. Other hosts can reach your computer by this name.
HOSTTYPE	The string describing the hardware platform where Bash is running.
LANG	Locale settings ²⁷⁹ for the user interface. They define the user's language, region and special characters. Some settings are overridden by variables <code>LC_ALL</code> , <code>LC_COLLATE</code> , <code>LC_CTYPE</code> , <code>LC_MESSAGES</code> , <code>LC_NUMERIC</code> , <code>LC_TYPE</code> .
MACHTYPE	The string describing the system where Bash is running. It includes information from the <code>HOSTTYPE</code> and <code>OSTYPE</code> variables.
OLDPWD	The previous directory that the <code>cd</code> command has set.
OSTYPE	The string describing of the OS where Bash is running.
POSIXLY_CORRECT	If this variable is defined, Bash runs in the POSIX compatible mode ²⁸⁰ .
PWD	The current directory that the <code>cd</code> command has set.

²⁷⁸https://www.gnu.org/software/bash/manual/html_node/The-Shopt-Builtin.html#The-Shopt-Builtin

²⁷⁹[https://en.wikipedia.org/wiki/Locale_\(computer_software\)](https://en.wikipedia.org/wiki/Locale_(computer_software))

²⁸⁰https://www.gnu.org/software/bash/manual/html_node/Bash-POSIX-Mode.html#Bash-POSIX-Mode

Table 3-5. Bash variables

Name	Value
RANDOM	Each time you read this variable, Bash returns a random number between 0 and 32767. When you write the variable there, Bash assigns a new initializing number (seed²⁸¹) to the pseudorandom number generator²⁸² .
SECONDS	The number of seconds elapsed since the current Bash process started.
SHELL	The path to the shell executable for the current user. Each user can use his own shell program.
SHLVL	The nesting level of the current Bash instance. This variable is incremented by one each time you start Bash from the shell or script.
UID	The ID number of the current user.

The internal variables are divided into three groups depending on the allowed actions with them. These are the groups:

1. Bash assigns a value to a variable at startup. It remains unchanged throughout the session. You can read it, but changing is prohibited. Examples: BASHOPTS, GROUPS, SHELLOPTS, UID.
2. Bash assigns a default value to a variable at startup. Your actions or other events change this value. You can re-assign some values explicitly, but this can disrupt the interpreter. Examples: HISTCMD, OLDPWD, PWD, SECONDS, SHLVL.
3. Bash assigns a default value to the variable at startup. You can change it. Examples: HISTFILESIZE, HISTSIZE.

Special Parameters

Bash declares special parameters and assigns values to them. It handles them the same way as shell variables.

Special parameters pass information from the shell to the launched application and vice versa. A positional parameter is an example of this kind of Bash variable.

Table 3-6 shows frequently used special parameters.

²⁸¹https://en.wikipedia.org/wiki/Random_seed

²⁸²https://en.wikipedia.org/wiki/Pseudorandom_number_generator

Table 3-6. Bash Special Parameters

Name	Value
\$*	The string with all positional parameters passed to the script. Parameters start with the \$1 variable but not with \$0. If you skip the double quotes (\$*), Bash inserts each positional parameter as a separate word. With double quotes (“\$*”), Bash handles it as one quoted string. The string contains all the parameters separated by the first character of the internal variable IFS.
\$@	The list of strings that contains all positional parameters passed to the script. Parameters start with the \$1 variable. If you skip double quotes (\$@), Bash handles each array’s element as an unquoted string. Word splitting happens in this case. With double quotes (“\$@”), Bash handles each element as a quoted string without word splitting.
\$#	The number of positional parameters passed to the script.
\$1, \$2...	They contain the value of the corresponding positional parameter. \$1 matches the first parameter. \$2 matches the second one, etc. These numbers are given in the decimal system.
\$?	The exit status of the last executed command in the foreground mode. If you have executed a pipeline, the parameter stores the exit status of the last command in this pipeline.
\$-	It contains options for the current interpreter instance.
\$\$	The process ID of the current interpreter instance. If you read it in the subshell, Bash returns the PID of the parent process.
#!	The process ID of the last command launched in the background mode.
\$0	The name of the shell or script that is currently running.

You cannot change special Bash parameters directly. For example, the following redeclaration of \$1 does not work:

```
1="new value"
```

If you want to change positional parameters, use the `set` command. It redeclares all parameters at once. There is no option to change a single positional parameter only. Here is the general form of the `set` call:

```
set -- NEW_VALUE_OF_$1 NEW_VALUE_OF_$2 NEW_VALUE_OF_$3...
```

What to do if you need to change a single positional parameter? Here is an example. Suppose you call the script with four parameters like this:

```
./my_script.sh arg1 arg2 arg3 arg4
```

You want to replace the third parameter `arg3` with the `new_arg3` value. The following `set` call does that:

```
set -- "${@:1:2}" "new_arg3" "${@:4}"
```

Let's consider this command in detail. Bash replaces the first argument “\$” with the first two elements of the `$@` array. It leads that `$1` and `$2` parameters get their previous values. Then there is the new value for the parameter `$3`. Now it equals “`new_arg3`”. The “\$” value comes at the end. Here Bash inserts all elements of the `$@` array starting from `$4`. It means that all these parameters get their previous values.

All special parameters from Table 3-6 are available in the POSIX-compatible mode of Bash.

Scope

Environment Variables

Any software system has **scopes** that group variables. A scope is a part of a program or system where the variable name remains associated with its value. There you can convert the variable name into its address. Outside the scope, the same name can point to another variable.

A scope is called **global** if it spreads to the whole system. Here is an example. Suppose that the variable called `filename` is in the global scope. Then you can access it by its name from any part of the system.

Bash keeps all its internal variables in the global scope. They are called **environment variables**. It means that all internal variables are environment variables. You can declare your variable in the global scope too. Then it becomes a new environment variable.

Why does Bash store variables in the global scope? It happens because Unix has a special set of settings. They affect the behavior of the applications that you run. An example is locale settings. They dictate how each application should adapt its interface. Applications receive Unix settings through environment variables.

Suppose one process spawns a child process. The child process inherits all environment variables of the parent. This way, all utilities and applications launched from the shell inherit its environment variables. This mechanism allows all programs to receive global Unix settings.

The child process can change its environment variables. When it spawns another process, it inherits the changed variables. However, when the child changes its environment variables, it does not affect the corresponding variables of the parent process.

The `export` built-in command declares an environment variable. Here is an example of doing that:


```
export BROWSER_PATH="/opt/firefox/bin"
```

You can declare the variable and then add it to the global scope. Call the `export` command this way:

```
1 BROWSER_PATH="/opt/firefox/bin"
2 export BROWSER_PATH
```

Sometimes you need to declare the environment variables for the specific application only. List the variables and their values before the application call in this case. Here is an example:

```
MOZ_WEBRENDER=1 LANG="en_US.UTF-8" /opt/firefox/bin/firefox
```

This command launches the Firefox browser and passes it the `MOZ_WEBRENDER` and `LANG` variables. They can differ from the global Unix settings.

The last example works well in Bash. If you use another shell, you need another approach. Suppose that you use Bourne Shell. Then you can pass variables to the application using the `env` utility. Here is an example of doing that:

```
env MOZ_WEBRENDER=1 LANG="en_US.UTF-8" /opt/firefox/bin/firefox
```

If you call the `env` utility without parameters, it prints all declared environment variables for the current interpreter process. Call it in your terminal this way:

```
env
```

The `export` Bash built-in and the `env` utility print the same thing when called without parameters. Use `export` instead of `env`. There are two reasons for that. First, the `export` sorts its output. Second, it adds double quotes to the values of all variables. They prevent you from making a mistake if some values have line breaks.

All names of environment variables contain uppercase letters only. Therefore, it is a good practice to name local variables in lower case. It prevents you from accidentally using one variable instead of another.

Local Variables

We have considered the user-defined variables. You can declare them in several ways. Depending on your choice, the new variable comes to the **local scope** or global scope (environment).

There are two ways to declare the global scope variable:

1. Add the `export` command to the variable declaration.

2. Pass the variable to the program when launching it. You can do it with the `env` utility when using a shell other than Bash.

If you do not apply any of these ways, your variable comes to the local scope. A variable of this kind is called a **local variable**. It is available in the current instance of the interpreter. A child process (except a subshell) does not inherit it.

Here is an example. Suppose that you declare the `filename` variable in the terminal window this way:

```
filename="README.txt"
```

Now you can print its value in the same terminal window. The following `echo` command does that:

```
echo "$filename"
```

The same `echo` command works well in a subshell. You can try it. Spawn the subshell by adding the parentheses around the Bash command. It looks like this:

```
(echo "$filename")
```

The child process does not get the local `filename` variable. Let's check it. Start a child process by calling the Bash interpreter explicitly. Do it this way:

```
bash -c 'echo "$filename"'
```

The `-c` parameter passes a command that the Bash child process executes. A similar Bash call occurs implicitly when you run a script from the shell.

We enclose the `echo` call in the single quotes when passing it to the `bash` command. The quotes disable all Bash expansions for the string inside. This behavior differs from the double quotes. They disable all expansions except the command substitution and parameter expansion. If we apply double quotes in our `bash` call, the parameter expansion happens. Then Bash inserts the variable's value in the call. This way, we will get the following command:

```
bash -c "echo README.txt"
```

We are not interested in this command. Instead, we want to check how the child process reads the local variable. Therefore, the parent process should not insert its value into the `bash` call.

If you change a local variable in the subshell, its value stays the same in the parent process. The following commands confirm this rule:

```
1 filename="README.txt"
2 (filename="CHANGELOG.txt")
3 echo "$filename"
```

If you execute them, you get the “README.txt” output. It means that changing the local variable in the subshell does not affect the parent process.

When you declare a local variable, it comes to the shell’s variables list. The list includes all local and environment variables that are available in the current interpreter process. The `set` command prints this list when called without parameters. Here is an example of how to find the `filename` variable there:

```
set | grep filename=
```

The `grep` utility prints the following string with the `filename` value:

```
filename=README.txt
```

It means that the `filename` variable is in the list of shell variables.

Variable Content Type

Variable Types

It is common practice to use the **static type system**²⁸³ in compiled programming languages (such as C). When using this system, you decide how to store the variable in memory. You should specify the variable type when declaring it. Then the compiler allocates memory and picks one of the predefined formats to store this type of variable.

Here is an example of how the static type system works. Suppose you want to declare a variable called `number`. You should specify its type in the declaration. You choose the unsigned integer type, which has a size of two bytes. Then the compiler allocates exactly two bytes of memory for this variable.

When the application starts, you assign the 203 value to the variable. It is equal to 0xCB in hexadecimal. Then the variable looks this way in the memory:

```
00 CB
```



Modern computers use the binary form to store information in the memory. Here we use the hexadecimal format instead for clarity.

²⁸³https://en.wikipedia.org/wiki/Type_system#Static_type_checking

One byte is enough to store the 203 value. However, you forced the compiler to reserve two bytes for that. The unused byte stays zeroed. No one can use it in the scope of the number variable. If the variable has a global scope, the byte is reserved and unused while the application works.

Suppose that you have assigned the 14037 value to the variable. It is equal to 0x36D5 in hexadecimal. Then it looks like this in the memory:

```
36 D5
```



The CPU determines the byte order when storing data in the computer memory. This byte order is called **endianness**²⁸⁴. It is **big-endian** in our example. The alternative order is **little-endian**.

Now you want to store the 107981 (0x1A5CD) value in the variable. This number does not fit into two bytes. The variable's size is defined in the declaration. The compiler cannot extend it automatically afterward. Therefore, it writes only part of the 107981 value into the variable. It looks like this in the memory:

```
A5 CD
```

The compiler discarded the first digit of the number. If you read the variable, you get 42445 (0xA5CD). It means that you lose the original 107981 value. You cannot recover it anymore. This problem is called **integer overflow**²⁸⁵.

Here is another example of the static type system. Suppose you want to store the username in a variable called `username`. You declare this variable and assign it the string type. When doing that, you should specify the maximum length of the string. It can be ten characters, for example.

After declaring the variable, you write the "Alice" name there. If you use the C compiler, the string looks this way in memory:

```
41 6C 69 63 65 00 00 00 00 00
```



The C compiler uses the ASCII encoding for characters by default. When using it, you can clarify the hexadecimal code of each letter in the **ASCII table**²⁸⁶.

Six bytes are enough to store the string "Alice". The first five bytes store characters. The last sixth byte stores the null character (00). It marks the end of the string. However, the compiler has reserved ten bytes for the variable. It fills the unused memory with zeros or random values.

²⁸⁴<https://en.wikipedia.org/wiki/Endianness>

²⁸⁵https://en.wikipedia.org/wiki/Integer_overflow

²⁸⁶<http://www.asciitable.com>

Dynamic type system²⁸⁷ is an alternative to the static system. It uses another approach to choose how to store a variable in memory. This choice happens whenever you assign the new value to the variable. Together with the value, the variable gets new **metadata**²⁸⁸. The metadata defines the variable type. They can change during the application work. Thus, the variable's representation in memory changes too. Most interpreted programming languages use the dynamic type system (for example, Python).



Metadata is additional information about some object or data. The **library catalog**²⁸⁹ is an example of metadata. It has a card for each book. The card contains the author, title of the work, publisher, year of publication, and the number of pages. Thus, the card contains metadata about the book.

Strictly speaking, Bash does not have the type system at all. It is not a language with a static or dynamic type system. Bash stores all **scalar variables**²⁹⁰ in memory as strings.

The scalar variable stores data of a **primitive type**²⁹¹. These data are the minimal building blocks to construct more complex **composite types**²⁹². The scalar variable is just a name for the memory address where its value is stored.

Here is an example of how Bash represents scalar variables in memory. Suppose you made the following variable declaration:

```
declare -i number=42
```

Bash stores the number variable in memory as the following string:

```
34 32 00
```

Any language with the type system needs one byte to store this integer. But Bash needs three bytes. The first two bytes store each character of the integer. The characters are 4 and 2. The third byte stores the null character.

The Bourne Shell language has the scalar variables only. Bash introduces two new composite types: **indexed array**²⁹³ and **associative array**²⁹⁴.

The indexed array is a numbered set of strings. There each string corresponds to the sequence number. Bash stores such an array as a **linked list**²⁹⁵ in memory. A linked list is a data structure that

²⁸⁷https://en.wikipedia.org/wiki/Type_system#Dynamic_type_checking_and_runtime_type_information

²⁸⁸<https://en.wikipedia.org/wiki/Metadata>

²⁸⁹https://en.wikipedia.org/wiki/Library_catalog

²⁹⁰[https://en.wikipedia.org/wiki/Variable_\(computer_science\)](https://en.wikipedia.org/wiki/Variable_(computer_science))

²⁹¹https://en.wikipedia.org/wiki/Primitive_data_type

²⁹²https://en.wikipedia.org/wiki/Composite_data_type

²⁹³https://en.wikipedia.org/wiki/Array_data_structure

²⁹⁴https://en.wikipedia.org/wiki/Associative_array

²⁹⁵https://en.wikipedia.org/wiki/Linked_list

consists of nodes. Each node contains data and the memory address of the next node. Node data are strings in this case.

The associative array is a more complicated thing. It is a set of elements. Each element consists of two strings. The first one is called “key”. The second is called “value”. When you want to access the array’s element, you should specify its key. It works the same as for the indexed array, where you specify the element’s index. The keys are unique. It means that the array cannot have two elements with the same keys. Bash stores associative array as a **hash-table**²⁹⁶ in memory.

Why are Bash “arrays” called arrays? Actually, they are linked lists and hash tables. A real array is the data structure whose elements are stored in memory one after another. Each element has a sequential number called an **index** or identifier. Bash “arrays” do not store their elements sequentially in memory. Thus, they are not arrays according to the definition.

Here is an example of how a real array stores its elements in memory. Suppose you have an array with numbers from five to nine. Each element takes one byte. Then the size of the array is five bytes. It looks this way in memory:

```
05 06 07 08 09
```

The indexing of arrays’ elements starts with zero. It means that the index of the first element equals 0. The second index equals 1 and so on. In our example, the first element with the 0 index equals integer 5. The second element equals 6. Elements follow each other in memory. Their indexes match the memory offset from the beginning of the array. Thus, the element with the third index has three bytes offset. Its value equals integer 8.

Let’s come back to the question about naming the Bash “arrays”. Only the authors of the language can answer it. However, we can guess. The name “array” gives you a hint of how to work with such a variable. When you have experience with another language, you know how to operate with a regular array. This way, you can start using Bash “arrays” immediately. You do not need to know how Bash stores these “arrays” internally.

Attributes

The Bash language does not have a type system. It stores all scalar variables in memory as strings. At the same time, Bash has arrays. They are composite types because an array is a combination of strings.

When you declare a variable in Bash, you should choose if it is scalar or composite. You make this choice by specifying metadata for the variable. Such metadata is called **attributes**. The attributes also define the constancy and scope of a variable.

The `declare` Bash built-in specifies the variable attributes. When you call it without parameters, `declare` prints all local and environment variables. The `set` command prints the same output.

The `declare` command has the `-p` option. The option adds variables attributes to the output data.

²⁹⁶https://en.wikipedia.org/wiki/Hash_table

If you need information on a particular variable, pass its name to the `declare` command. Here is an example of the `PATH` variable:

```
declare -p PATH
```

The `declare` command also prints information about declared [subroutines](#)²⁹⁷. They are called **functions** in Bash. A function is a program fragment or an independent block of code that performs a certain task.

Suppose you are interested in function declarations but not in variables. Then apply the `-f` option of the `declare` command. It filters out variables from the output. Here is the `declare` call in this case:

```
declare -f
```

You can specify the function name right after the `-f` option. Then the `declare` command prints information about it. Here is an example of the function `quote`:

```
declare -f quote
```

This command displays the declaration of the `quote` function. The function takes a string on the input and encloses it in single quotes. If the string already contains the single quotes, the function escapes them. You can call `quote` in the same way as any Bash built-in. Here is an example:

```
quote "this is a 'test' string"
```

The `declare` call without the `-p` option does not print a function declaration. It means that the following command outputs nothing:

```
declare quote
```

We have learned how to get information about already declared variables and functions using `declare`. Now let's find out how this command sets attributes for new variables.

Table 3-7 shows the frequently used options of the `declare` command.

²⁹⁷<https://en.wikipedia.org/wiki/Subroutine>

Table 3-7. The `declare` command options and the corresponding variable attributes

Option	Definition
-a	The declared variable is an indexed array.
-A	The declared variable is an associative array.
-g	It declares a variable in the global scope of the script. The variable does not come to the environment.
-i	It declares an integer variable. When you assign it a value, Bash treats it as an arithmetic expression.
-r	It declares a constant. The constant cannot change its value after declaration.
-x	It declares an environment variable.

Here are several examples of how to declare variables with attributes. First, let's compare integer and string variables. Execute the following two commands in the terminal window:

```
1 declare -i sum=11+2
2 text=11+2
```

We declared two variables named `sum` and `text`. The `sum` variable has the integer attribute. Therefore, its value equals 13 that is the sum of 11 and 2. The `text` variable is equal to the "11+2" string.

Bash stores both variables as strings in memory. The `-i` option does not specify the variable's type. Instead, it limits the allowed values of the variable.

Try to assign a string to the `sum` variable. You can do it in one of the following ways:

```
1 declare -i sum="test"
2 sum="test"
```

Each of these commands sets the `sum` value to zero. It happens because the variable has the integer attribute. Therefore, it cannot be equal to some string.

Suppose you have declared an integer variable. Then you do not need any Bash expansion for arithmetic operations on it. The following commands do correct calculations:

```
1 sum=sum+1      # 13 + 1 = 14
2 sum+=1         # 14 + 1 = 15
3 sum+=sum+1     # 15 + 15 + 1 = 31
```

Here the calculation results come after the hash symbol. Bash ignores everything after this symbol. Such lines are called **comments**²⁹⁸.

²⁹⁸[https://en.wikipedia.org/wiki/Comment_\(computer_programming\)](https://en.wikipedia.org/wiki/Comment_(computer_programming))



The usefulness of comments is the subject of endless debates in the programming community. They are needed to explain the code. However, some people consider that comments are a sign of incomprehensible, poorly written code. If you just started learning to program, comment your code without doubts. Explain the complex constructions in your scripts. Otherwise, you can forget what do they mean afterward.

Now execute the same commands with the string variable. You will get the following results:

```
1 text=text+1      # "text+1"
2 text+=1          # "text+1" + "1" = "text+11"
3 text+=text+1    # "text+11" + "text" + "1" = "text+11text+1"
```

Here Bash concatenates strings instead of doing arithmetic calculations. If you want to operate on integers instead, you should use the arithmetic expansion. Here is an example of this expansion:

```
1 text=11
2 text=$((text + 2)) # 11 + 2 = 13
```

When you apply the `-r` option of the `declare` built-in, you get a constant. Such a call looks this way:

```
declare -r filename="README.txt"
```

Whenever you change or delete the value of the `filename` constant, Bash prints an error message. Therefore, both following commands fail:

```
1 filename="123.txt"
2 unset filename
```



Use the `unset` command for removing variables. This command cannot remove constants.

The `-x` option of the `declare` command declares an environment variable. It provides the same result as if you apply the `export` built-in in the variable declaration. Thus, the following two commands are equivalent:

```
1 export BROWSER_PATH="/opt/firefox/bin"
2 declare -x BROWSER_PATH="/opt/firefox/bin"
```

A good practice is to use the `export` command instead of `declare` with the `-x` option. This improves the code readability. You do not need to remember what the `-x` option means. For the same reason, you should prefer the `readonly` command instead of `declare` with the `-r` option. Both built-ins declare a constant, but `readonly` is easier to remember.

The `readonly` command declares a variable in the global scope of a script. The `declare` built-in with the `-r` option has another result. If you call it in a function body, you declare a local variable. It is not available outside the function. Use the `-g` option to get the same behavior as `readonly`. Here is an example:

```
declare -gr filename="README.txt"
```

Indexed Arrays

Bourne Shell has scalar variables only. The interpreter stores them as strings in memory. Working with such variables is inconvenient in some cases. Therefore, developers have added arrays to the Bash language. When do you need an array?

Strings have a serious limitation. When you write a value to the scalar variable, it is a single unit. For example, you save a list of filenames in the variable called `files`. You separate them by spaces. As a result, the `files` variable stores a single string from the Bash point of view. It can lead to errors.

The root cause of the problem came from the POSIX standard. It allows any characters in filenames except the null character (NULL). NULL means the end of a filename. The same character means the end of a string in Bash. Therefore, a string variable can contain NULL at the end only. It turns out that you have no reliable way to separate filenames in a string. You cannot use NULL, but any other delimiter character can occur in the names.

You cannot process results of the `ls` utility reliable because of the delimiter problem. The utility cannot use NULL as a separator for names of files and directories in its output. It leads to a recommendation to avoid parsing of the `ls` output. Another advice is to not use `ls` in variable declarations this way:

```
files=$(ls Documents/*.txt)
```

This declaration writes all TXT files of the `Documents` directory to the `files` variable. If there are spaces or line breaks in the filenames, you cannot separate them properly anymore.

Bash arrays solve the delimiter problem. An array stores a list of separate units. You can always read them in their original form. Therefore, use an array to store filenames instead of a string. Here is a better declaration of the `files` variable:

```
declare -a files=(Documents/*.txt)
```

This command declares and **initializes** the array named `files`. Initializing means assigning values to the array's elements. You can do that in the declaration or after it.

When you declare a variable, Bash can deduce if it is an array. This mechanism works when you skip the `declare` built-in. Bash adds the appropriate attribute automatically. Here is an example:

```
files=(Documents/*.txt)
```

This command declares the indexed array `files`.

Suppose that you know all array elements in advance. In this case, you can assign them explicitly in the declaration. It looks like this:

```
files=( "/usr/share/doc/bash/README" "/usr/share/doc/flex/README.md" "/usr/share/doc/\
xz/README" )
```

When assigning array elements, you can read them from other variables. Here is an example:

```
1 bash_doc="/usr/share/doc/bash/README"
2 flex_doc="/usr/share/doc/flex/README.md"
3 xz_doc="/usr/share/doc/xz/README"
4 files=( "$bash_doc" "$flex_doc" "$xz_doc" )
```

This command writes values of variables `bash_doc`, `flex_doc` and `xz_doc` to the `files` array. If you change these variables after this declaration, it does not affect the array.

When declaring an array, you can specify an index for each element explicitly. Do it this way:

```
1 bash_doc="/usr/share/doc/bash/README"
2 flex_doc="/usr/share/doc/flex/README.md"
3 xz_doc="/usr/share/doc/xz/README"
4 files=( [0]="$bash_doc" [1]="$flex_doc" [5]="/usr/share/doc/xz/README" )
```

Here there are no spaces before and after each equal sign. Remember this rule: when you declare any variable in Bash, you do not put spaces near the equal sign.

Instead of initializing the entire array at once, you can assign its elements separately. Here is an example:

```
1 files[0]="$bash_doc"
2 files[1]="$flex_doc"
3 files[5]="/usr/share/doc/xz/README"
```

There are gaps in the array indexes in the last two examples. It is not a mistake. Bash allows arrays with such gaps. They are called **sparse arrays**.

Suppose that you have declared an array. Now there is a question of how to read its elements. The following parameter expansion prints all of them:

```
$ echo "${files[@]}"  
/usr/share/doc/bash/README /usr/share/doc/flex/README.md /usr/share/doc/xz/README
```

You see the echo command at the first line. Its output comes on the next line.

It can be useful to print indexes of elements instead of their values. For doing that, add an exclamation mark in front of the array name in the parameter expansion. Here is an example:

```
1 $ echo "${!files[@]}"  
2 0 1 5
```

You can calculate an element index using some formula. Specify the formula in square brackets when accessing the array. The following commands read and write the fifth element:

```
1 echo "${files[4+1]}"  
2 files[4+1]="/usr/share/doc/xz/README"
```

You can use variables in the formula. Bash accepts both integer and string variables there. Here is another way to access the fifth element of the array:

```
1 i=4  
2 echo "${files[i+1]}"  
3 files[i+1]="/usr/share/doc/xz/README"
```

You can insert the sequential array elements at once. Specify the starting index, colon and the number of elements in the parameter expansion. Here is an example:

```
1 $ echo "${files[@]:1:2}"  
2 /usr/share/doc/flex/README.md /usr/share/doc/xz/README
```

This echo call prints two elements, starting from the first. The elements' indexes are not important in this case. You get the filenames with indexes 1 and 5.

Starting with version 4, Bash provides the `readarray` built-in. It is also known as `mapfile`. The command reads the contents of a text file into an indexed array. Let's see how to use it.

Suppose you have the file named `names.txt`. It contains names of several persons:

```
1 Alice  
2 Bob  
3 Eve  
4 Mallory
```

You want to create an array with strings of this file. The following command does that:

```
readarray -t names_array < names.txt
```

The command writes all lines of the `names.txt` file to the `names_array` array.

Exercise 3-3. Declaration of arrays

Try all the following variants of the array declarations:

1. Using the `declare` command.
2. Without the `declare` command.
3. The globbing mechanism provides values for array elements.
4. Specify all array elements in the declaration.
5. Specify the elements separately after the array declaration.
6. Assign the values of the existing variables to array elements.
7. Read the array elements from a text file.

Print the array contents using the `echo` command for each case.

We have learned how to declare and initialize indexed arrays. Here are some more examples of using them. Suppose the `files` array contains a list of filenames. You want to copy the first file in the list. The following `cp` call does that:

```
cp "${files[0]}" ~/Documents
```



Most programming languages number the elements of arrays and characters of strings from zero, but not from one. Bash follows this rule too.

When reading an array element, you always apply the full form of the parameter expansion with curly brackets. Put the index of the element in square brackets after the variable name.

When you put the `@` symbol instead of the element's index, Bash inserts all array elements. Here is an example:

```
cp "${files[@]}" ~/Documents
```

You need to get an array size in some cases. Put the `#` character in front of its name. Then specify the `@` symbol as the element index. For example, the following parameter expansion gives you the size of the `files` array:

```
echo "${#files[@]}"
```

When reading array elements, always apply double quotes. They prevent errors caused by word splitting.

Call the `unset` Bash built-in if you need to remove an array element. Here is an example of removing the fourth element:

```
unset 'files[3]'
```

You can suppose that this command has the wrong element index. The command is correct. Remember about numbering array elements from zero. Also, single quotes are mandatory here. They turn off all Bash expansions.

The `unset` command can clear the whole array if you call it this way:

```
unset files
```

Associative Arrays

We have considered indexed arrays. Their elements are strings. Each element has an index that is a positive integer. The indexed array gives you a string by its index.

The developers introduced associative arrays in the 4th version of Bash. These arrays use strings as element indexes instead of integers. This kind of string-index is called **key**. The associative array gives you a string-value by its string-index. When do you need this feature?

Here is an example. Suppose you need a script that stores the list of contacts. The script adds a person's name, email or phone number to the list. Let's omit the person's last name for simplicity. When you need these data back, the script prints them on the screen.

You can solve the task using the indexed array. This solution would be inefficient for searching for the required contact. The script should traverse over all array elements. It compares each element with the person's name that you are looking for. When the script finds the right person, it prints his contacts on the screen.

An associative array makes searching for contacts faster. The script should not pass through all array elements in this case. Instead, it gives the key to the array and gets the corresponding value back.

Here is an example of declaring and initializing the associative array with contacts:

```
declare -A contacts=(["Alice"]="alice@gmail.com" ["Bob"]="(697) 955-5984" ["Eve"]="(\n245) 317-0117" ["Mallory"]="mallory@hotmail.com")
```

There is only one way to declare an associative array. For doing that, you should use the `declare` command with the `-A` option. Bash cannot deduce the array type without it, even if you specify string-indexes. Therefore, the following command declares the indexed array:

```
contacts=(["Alice"]="alice@gmail.com" ["Bob"]="(697) 955-5984" ["Eve"]="(245) 317-01\
17" ["Mallory"]="mallory@hotmail.com")
```

Let's check of how this indexed array looks like. The following `declare` call prints it:

```
1 $ declare -p contacts
2 declare -a contacts='([0]="mallory@hotmail.com")'
```

You see the indexed array with one element. This result happened because Bash converts all string-indexes to zero value. Then every next contact in the initialization list overwrites the previous one. This way, the zero-index element contains contacts of the last person in the initialization list.

You can specify elements of an associative array separately. Here is an example:

```
1 declare -A contacts
2 contacts["Alice"]="alice@gmail.com"
3 contacts["Bob"]="(697) 955-5984"
4 contacts["Eve"]="(245) 317-0117"
5 contact["Mallory"]="mallory@hotmail.com"
```

Suppose that you have declared an associative array. Now you can access its elements by their keys. The key is the name of a person in our example. The following command reads the contacts by the person's name:

```
1 $ echo "${contacts["Bob"]}"
2 (697) 955-5984
```

If you put the `@` symbol as the key, you get all elements of the array:

```
1 $ echo "${contacts[@]}"
2 (697) 955-5984 mallory@hotmail.com alice@gmail.com (245) 317-0117
```

If you add the exclamation mark before the array name, you get the list of all keys. It is the list of persons in our example:

```
1 $ echo "${!contacts[@]}"
2 Bob Mallory Alice Eve
```

Add the `#` character before the array name to get its size:

```
1 $ echo "${#contacts[@]}"
2 4
```

Let's apply the associative array to the contacts script. The script receives a person's name via the command-line parameter. Then it prints an email or phone number of that person.

Listing 3-10 shows the script for managing the contacts.

Listing 3-10. The script for managing the contacts

```
1 #!/bin/bash
2
3 declare -A contacts=(
4   ["Alice"]="alice@gmail.com"
5   ["Bob"]="(697) 955-5984"
6   ["Eve"]="(245) 317-0117"
7   ["Mallory"]="mallory@hotmail.com")
8
9 echo "${contacts["$1"]}"
```

If you need to edit some person's data, you should open the script in a code editor and change the array initialization.

The `unset` Bash built-in deletes an associative array or its element. It works this way:

```
1 unset contacts
2 unset 'contacts[Bob]'
```

Bash can insert several elements of an associative array in the same way as it does for an indexed array. Here is an example:

```
1 $ echo "${contacts[@]:0:2}"
2 mallory@hotmail.com (245) 317-0117
```

Here you get the first two elements of the array.

There is one problem with inserting several elements of an associative array. Their order in memory does not match their initialization order. The **hash function**²⁹⁹ calculates a numerical index in the memory of each element. The function takes a string-key on input and returns a unique integer on output. Because of this feature, inserting several elements of the associative array is a bad practice.

²⁹⁹https://en.wikipedia.org/wiki/Hash_function

Conditional Statements

We met the conditional statements the first time when learning the `find` utility. Then we found out that Bash has its own logical operators AND (`&&`) and OR (`||`). This language has other options to make branches.

We will consider the `if` and `case` operators in this section of the book. You will use them frequently when writing Bash scripts. These operators provide similar behavior. However, each of them fits better for some specific tasks.

If Statement

Imagine that you are writing a one-line command. Such a command is called **one-liner**. You are trying to make it as compact as possible because a short command is faster to type. Also, compactness gives you less chance to make a mistake when typing.

Now imagine that you are writing a script. The hard drive stores it. You call the script regularly and change it rarely. The code compactness is not important in this case. Instead, you try to make the code simple for reading and changing.

The `&&` and `||` operators fit well for one-liners. When you are writing scripts, Bash gives you better options. Actually, it depends on the particular case. Sometimes you can use logical operators in the script and keep its code clean. However, they lead to hard-to-read code in most cases. Therefore, it is better to replace them with the `if` and `case` statements.

Here is an example. Have a look at Listing 3-9 again. It shows the backup script. You can see this `bsdtar` call there:

```
1 bsdtar -cjf "$1".tar.bz2 "$@" &&  
2   echo "bsdtar - OK" > results.txt ||  
3   { echo "bsdtar - FAILS" > results.txt ; exit 1 ; }
```

When writing this script, we have tried to make it better for reading. This motivation forced us to split calls of the `bsdtar` and `mv` utilities. This solution is still not enough. The `bsdtar` call is too long and complicated for reading. Therefore, it is easy to make a mistake when modifying it. Such error-prone code is called **fragile**. You get it whenever making a poor technical solution.

Let's improve the `bsdtar` call. The first step for improving the code is writing its algorithm in a clean way. Here is the algorithm for our case:

1. Read a list of files and directories from the `$@` variable.
2. Archive and compress the files and directories.
3. If the archiving succeeds, write the "bsdtar - OK" line into the log file.
4. If an error occurred, write the line "bsdtar - FAILS" into the log file and terminate the script.

The last step is the most confusing one. When `bsdtr` succeeds, the script does one action only. When an error happens, there are two actions. These actions are combined into the single **command block**³⁰⁰ by **curly brackets**³⁰¹. This code block looks too complicated.

The `if` statement executes command blocks on specific conditions. The statement looks this way in the general form:

```
1 if CONDITION
2 then
3     ACTION
4 fi
```

You can write the `if` statement in one line. For doing that, add semicolons before `then` and `fi` like this:

```
if CONDITION; then ACTION; fi
```

The `CONDITION` and `ACTION` here mean a single command or a block of commands. If the exit status of the `CONDITION` equals zero, Bash executes the `ACTION`.

Here is an example of the `if` statement:

```
1 if cmp file1.txt file2.txt &> /dev/null
2 then
3     echo "Files file1.txt and file2.txt are identical"
4 fi
```

The `cmp` utility call works as the `CONDITION` here. The utility compares the contents of two files. If they differ, `cmp` prints the position of the first distinct character. The exit status is non-zero in this case. If the files are the same, the utility returns the zero status.

When you call a utility or command in the `if` condition, its exit status matters only. Therefore, we redirect the `cmp` output to the `/dev/null`³⁰² file. It is a special system file. OS deletes all data that you write there. This operation always succeeds.

If the contents of the `file1.txt` and `file2.txt` files match, the `cmp` utility returns the zero status. Then the `if` condition equals “true”. The `echo` command prints the message in this case.

We have considered a simple `if` statement with a single `CONDITION` and `ACTION`. When the condition is met, `if` performs the `ACTION`. There are cases when you want to choose one of two possible `ACTIONS` using the `CONDITION`. The `if-else` statement solves this task. Here is the statement in the general form:

³⁰⁰[https://en.wikipedia.org/wiki/Block_\(programming\)](https://en.wikipedia.org/wiki/Block_(programming))

³⁰¹https://www.gnu.org/software/bash/manual/html_node/Command-Grouping.html

³⁰²https://en.wikipedia.org/wiki/Null_device

```
1  if CONDITION
2  then
3    ACTION_1
4  else
5    ACTION_2
6  fi
```

If you write the `if-else` statement in one line, it looks this way:

```
if CONDITION; then ACTION_1; else ACTION_2; fi
```

Bash executes `ACTION_2` if the `CONDITION` returns the non-zero exit status. The condition is “false” in this case. Otherwise, Bash executes the `ACTION_1`.

You can extend the `if-else` statement by the `elif` blocks. Such a block adds an extra `CONDITION` and the corresponding `ACTION`. Bash executes the `ACTION` if the `CONDITION` equals “true”.

Here is an example of the `if-else` statement. Suppose you want to choose one of three actions depending on the value of a variable. The following `if` statement does that:

```
1  If CONDITION_1
2  then
3    ACTION_1
4  elif CONDITION_2
5  then
6    ACTION_2
7  else
8    ACTION_3
9  fi
```

There is no limitation on the number of `elif` blocks in the statement. You can add as many blocks as you need.

Let’s improve our example of file comparison. You need to print the message in both cases: when the files match and when they do not. The following `if-else` statement does that:

```
1  if cmp file1.txt file2.txt &> /dev/null
2  then
3    echo "Files file1.txt and file2.txt are the same."
4  else
5    echo "Files file1.txt and file2.txt differ."
6  fi
```

It is time to come back to our backup script. There the `echo` call combined with `exit` makes a block of commands. Bash executes it depending on the result of the `bsdtar` utility. Whenever you meet a code block and condition, it is a hint to apply the `if` statement.

If you apply the `if-else` statement to check the `bsdtar` result, you get the following code:

```
1 if bsdtar -cjf "$1".tar.bz2 "$@"
2 then
3   echo "bsdtar - OK" > results.txt
4 else
5   echo "bsdtar - FAILS" > results.txt
6   exit 1
7 fi
```

Do you agree that it is easier to read the code now? You can simplify it even more. The **early return**³⁰³ pattern will help you with that. Replace the `if-else` statement with `if` this way:

```
1 if ! bsdtar -cjf "$1".tar.bz2 "$@"
2 then
3   echo "bsdtar - FAILS" > results.txt
4   exit 1
5 fi
6
7 echo "bsdtar - OK" > results.txt
```

This code behaves the same as one with the `if-else` statement. You can see that the `bsdtar` result was inverted. If the utility fails, the `if` condition equals “true”. Then Bash prints the “bsdtar - FAILS” message to the log file and terminates the script. Otherwise, Bash skips the whole command block of the `if` statement. Then the further `echo` call prints the “bsdtar - OK” message to the log file.

The early return pattern is a useful technique that makes your code cleaner and easier to read. The idea behind it is to terminate the program as early as possible when an error appears. This solution allows you to avoid the nested `if` statements.

An example will demonstrate the benefits of the early return pattern. Imagine the algorithm that does five actions. Each action depends on the result of the previous one. If any action fails, the algorithm stops. You can implement this algorithm with the nested `if` statements like this:

³⁰³<https://medium.com/swlh/return-early-pattern-3d18a41bba8>

```
1  if ACTION_1
2  then
3    if ACTION_2
4    then
5      if ACTION_3
6      then
7        if ACTION_4
8        then
9          ACTION_5
10       fi
11      fi
12     fi
13    fi
```

These nested statements look confusing. Suppose that you need to handle the errors. It means that you should add the `else` block for each `if` statement. It will make this code even harder to read.

The nested `if` statements make code bulky and incomprehensible. It is a serious problem. The early return pattern solves it. If you apply the pattern to our example algorithm, you get the following code:

```
1  if ! ACTION_1
2  then
3    # error handling
4  fi
5
6  if ! ACTION_2
7  then
8    # error handling
9  fi
10
11 if ! ACTION_3
12 then
13   # error handling
14 fi
15
16 if ! ACTION_4
17 then
18   # error handling
19 fi
20
21 ACTION_5
```

This is the same algorithm. Its behavior did not change. Bash performs the same five actions one by

one. If any of them fails, the algorithm stops. However, the code looks different. The early return pattern made it simpler and clearer.

Suppose that each action of the algorithm corresponds to one short command. The `exit` command handles all errors. Also, you do not need an output to the log file. You can replace the `if` statement with the `||` operator in this case. Then your code remains simple and clear. It will look this way:

```
1 ACTION_1 || exit 1
2 ACTION_2 || exit 1
3 ACTION_3 || exit 1
4 ACTION_4 || exit 1
5 ACTION_5
```

There is only one case when the `&&` and `||` operators are more expressive than the `if` statement. It happens when you operate short commands for doing actions and error handling.

Let's rewrite the backup script using the `if` statement. Listing 3-11 shows the result.

Listing 3-11. The backup script with the early return pattern

```
1 #!/bin/bash
2
3 if ! bsdtar -cjf "$1".tar.bz2 "$@"
4 then
5     echo "bsdtar - FAILS" > results.txt
6     exit 1
7 fi
8
9 echo "bsdtar - OK" > results.txt
10
11 mv -f "$1".tar.bz2 /d &&
12     echo "cp - OK" >> results.txt ||
13     ! echo "cp - FAILS" >> results.txt
```

We have replaced the `&&` and `||` operators in the `bsdtar` call with the `if` statement. It did not change the behavior of the script.

Logical operators and the `if` statement are not equivalent in general. An example will show you the difference between them. Suppose there is an expression of three commands A, B and C:

```
A && B || C
```

You can suppose that the following `if-else` statement gives the same behavior:

```
if A
then
  B
else
  C
fi
```

It looks like Bash does the same in both cases. If A is “true”, then Bash executes B. Otherwise, it executes C. This assumption is wrong. When you apply the logical operator, you get another behavior. If A is “true”, then Bash executes B. Then the B result defines if C execution happens. If B is “true”, Bash skips C. If B is “false”, it executes C. Thus, execution of C depends on both A and B. There is no such dependence in the `if-else` statement.

Exercise 3-4. The `if` statement

Here is the Bash command:

```
( grep -RlZ "123" target | xargs -0 cp -t . && echo "cp - OK" || ! echo "cp - FAILS" \
) && ( grep -RLZ "123" target | xargs -0 rm && echo "rm - OK" || echo "rm - FAILS" \
)
```

It looks for the string "123" in the files of the directory named "target". If the file contains the string, it is copied to the current directory. If there is no string in the file, it is removed from the target directory.

Make the script from this command.

Replace the `&&` and `||` operators with the `if` statements.

Operator `[[`

We got acquainted with the `if` statement. It calls a Bash built-in or utility in the condition.

Let's consider the options that you have for making an `if` condition. Suppose that you want to check if a text file contains some phrase. When the phrase presents, you should print a message in the log file.

You can combine the `if` statement and the `grep` utility to solve the task. Put the `grep` call in the `if` condition. If the utility succeeds, it returns zero exit status. In this case, the `if` condition equals “true” and Bash prints the message.

The `grep` utility prints its result to the output stream. You do not need it when calling `grep` in the `if` condition. You can get rid of the utility's output using the `-q` option. Then you get the following `if` statement:

```

1 if grep -q -R "General Public License" /usr/share/doc/bash
2 then
3   echo "Bash has the GPL license"
4 fi

```

The `grep` utility works well when you deal with files. But what should you do when you want to compare two strings or numbers? Bash has the `[[` operator for that. The double square brackets are the **reserved word**³⁰⁴ of the interpreter. Bash handles them on its own without calling a utility.



The Bourne shell does not have the `[[` operator. The POSIX standard does not have it too. Therefore, if you should follow the standard, use the obsolete `test`³⁰⁵ operator or its synonym `[`. Never use `test` in Bash. It has limited capabilities comparing with the operator `[[`. Also, its syntax is error-prone.

Let's start with a simple example of using the `[[` operator. You need to compare two strings. The following `if` condition does that:

```

1 if [[ "abc" = "abc" ]]
2 then
3   echo "The strings are equal"
4 fi

```

Write a script with this code and run it. It will show you the message that the strings are equal. This kind of check is not useful. Usually, you want to compare some variable with a string. The `[[` operator compares them this way:

```

1 if [[ "$var" = "abc" ]]
2 then
3   echo "The variable equals the \"abc\" string"
4 fi

```

Double quotes are optional in this condition. Bash skips globbing and word splitting when it substitutes a variable in the `[[` operator. The quotes prevent problems if the variable value or string contains spaces. Here is an example of such a case:

³⁰⁴https://en.wikipedia.org/wiki/Reserved_word

³⁰⁵<http://mywiki.woledge.org/BashFAQ/031>


```

1 if [[ "$var" = abc def ]]
2 then
3     echo "The variable equals the \"abc def\" string"
4 fi

```

This `if` condition causes the error because of word splitting. Always apply quotes when working with strings. This helps you to avoid such problems. Here is the corrected `if` condition for our example:

```

1 if [[ "$var" = "abc def" ]]
2 then
3     echo "The variable equals the \"abc def\" string"
4 fi

```

The `[[` operator can compare two variables with each other. The following `if` condition does that:

```

1 if [[ "$var" = "$filename" ]]
2 then
3     echo "The variables are equal"
4 fi

```

Table 3-8 shows all kinds of string comparisons that the `[[` operator performs.

Table 3-8. String comparisons using the `[[` operator

Operation	Description	Example
>	The string on the left side is larger than the string on the right side in the lexicographic order ³⁰⁶ .	<code>[["bb" > "aa"]] && echo "The "bb" string is larger than "aa"</code>
<	The string on the left side is smaller than the string on the right side in the lexicographic order.	<code>[["ab" < "ac"]] && echo "The "ab" string is smaller than "ac"</code>
= or ==	The strings are equal.	<code>[["abc" = "abc"]] && echo "The strings are equal"</code>
!=	The strings are not equal.	<code>[["abc" != "ab"]] && echo "The strings are not equal"</code>
-z	The string is empty.	<code>[[-z "\$var"]] && echo "The string is empty"</code>
-n	The string is not empty.	<code>[[-n "\$var"]] && echo "The string is not empty"</code>

³⁰⁶https://en.wikipedia.org/wiki/Lexicographic_order

Table 3-8. String comparisons using the [[operator

Operation	Description	Example
-v	The variable is set to any value.	[[-v var]] && echo "The string is set"
= or ==	Search a pattern on the right side in a string on the left side. You should not enclose the pattern in quotes here.	[["\$filename" = READ*]] && echo "The filename starts with "READ""
!=	Check that a pattern on the right side does not occur in a string on the left side. You should not enclose the pattern in quotes here.	[["\$filename" != READ*]] && echo "The filename does not start with "READ""
=~	Search a regular expression ³⁰⁷ on the right side in a string on the left side.	[["\$filename" =~ ^READ.*]] && echo "The filename starts with "READ""

You can use logical operations AND, OR and NOT in the [[operator. They combine several Boolean expressions into a single condition. Table 3-9 explains how they work.

Table 3-9. Logical operations in the [[operator

Operation	Description	Example
&&	Logical AND.	[[-n "\$var" && "\$var" < "abc"]] && echo "The string is not empty and it is smaller than "abc""
	Logical OR.	[["abc" < "\$var" -z "\$var"]] && echo "The string is larger than "abc" or it is empty"
!	Logical NOT.	[[! "abc" < "\$var"]] && echo "The string is not larger than "abc""

You can group Boolean expressions using parentheses in the [[operator. Here is an example:

```
[[ (-n "$var" && "$var" < "abc") || -z "$var" ]] && echo "The string is not empty and less than \"abc\" or the string is empty"
```

Comparing strings is one feature of the [[operator. Besides that, it can check files and directories for various conditions. Table 3-10 shows operations for doing that.

³⁰⁷<https://mywiki.woledge.org/RegularExpression>

Table 3-10. Operations for checking files and directories in the `[[` operator

Operation	Description	Example
-e	The file exists.	<code>[[-e "\$filename"]] && echo "The \$filename file exists"</code>
-f	The specified object is a regular file. It is not a directory or device file ³⁰⁸ .	<code>[[-f "~/README.txt"]] && echo "The README.txt is a regular file"</code>
-d	The specified object is a directory.	<code>[[-d "/usr/bin"]] && echo "The /usr/bin is a directory"</code>
-s	The file is not empty.	<code>[[-s "\$filename"]] && echo "The \$filename file is not empty"</code>
-r	The specified file exists, and you can read it.	<code>[[-r "\$filename"]] && echo "The \$filename file exists. You can read it"</code>
-w	The specified file exists, and you can write it.	<code>[[-w "\$filename"]] && echo "The \$filename file exists. You can write into it"</code>
-x	The specified file exists, and you can execute it.	<code>[[-x "\$filename"]] && echo "The \$filename file exists. You can execute it"</code>
-N	The file exists. It was modified since you read it last time.	<code>[[-N "\$filename"]] && echo "The \$filename file exists. It was modified"</code>
-nt	The file on the left side is newer than the file on the right side. Either the file on the left side exists and the file on the right side does not.	<code>[["\$file1" -nt "\$file2"]] && echo "The \$file1 file is newer than \$file2"</code>
-ot	The file on the left side is older than the file on the right side. Either the file on the right side exists and the file on the left side does not.	<code>[["\$file1" -ot "\$file2"]] && echo "The \$file1 file is older than \$file2"</code>
-ef	There are paths or hard links to the same file on the left and right sides. You cannot compare hard links if your file system does not support them.	<code>[["\$file1" -ef "\$file2"]] && echo "The \$file1 and \$file2 files match"</code>

The `[[` operator can compare integers. Table 3-11 shows operations for doing that.

³⁰⁸https://en.wikipedia.org/wiki/Device_file

Table 3-11. Integer comparisons using the `[[` operator

Operation	Description	Example
<code>-eq</code>	The number on the left side equals the number on the right side.	<code>[["\$var" -eq 5]] && echo "The variable equals 5"</code>
<code>-ne</code>	The numbers are not equal.	<code>[["\$var" -ne 5]] && echo "The variable is not equal to 5"</code>
<code>-gt</code>	Greater (>).	<code>[["\$var" -gt 5]] && echo "The variable is greater than 5"</code>
<code>-ge</code>	Greater or equal.	<code>[["\$var" -ge 5]] && echo "The variable is greater than or equal to 5"</code>
<code>-lt</code>	Less (<).	<code>[["\$var" -lt 5]] && echo "The variable is less than 5"</code>
<code>-le</code>	Less or equal.	<code>[["\$var" -le 5]] && echo "The variable is less than or equal to 5"</code>

Table 3-11 raises questions. Two letters mark each comparison operation. It is harder to remember them than usual comparison signs: `<`, `>`, and `=`. Why doesn't the `[[` operator use the comparison signs? We should have a look at the operator's history to answer this question.

The `[[` operator came to Bash to replace the obsolete `test` built-in. The first version of Bourne shell in 1979 did not have `test`. However, programmers needed the feature to compare strings, files and integers. Therefore, Unix developers have added the `test` utility for that purpose. This utility became built-in since the System III shell version in 1981. This change did not affect the `test` syntax. The reason for that is backward compatibility. Programmers have written a lot of code by 1981. This code has used the old `test` syntax. Therefore, the new System III shell version had to support it.

Let's take a look at the `test` syntax. When it was a utility, the format of its input parameters had to follow Bourne shell rules. For example, here is a typical `test` call to compare the `var` variable with the number five:

```
test "$var" -eq 5
```

This command does not raise any questions. We pass the following three parameters to the `test` utility:

1. The value of the `var` variable.
2. The `-eq` option.
3. The number 5.

We can use the `test` call in the `if` condition this way:

```
1 if test "$var" -eq 5
2 then
3     echo "The variable equals 5"
4 fi
```

The Bourne shell introduces the `[` operator as a synonym for the `test` built-in. The only difference between them is the mandatory closing parenthesis `]`. The `test` operator does not need it, but the operator does.

Using the `[` operator, we can rewrite the previous `if` condition this way:

```
1 if [ "$var" -eq 5 ]
2 then
3     echo "The variable equals 5"
4 fi
```

The `[` operator improves the code readability. Thanks to the operator, the `if` statement in the Bourne shell looks like in other programming languages. Problems happen because the `[` and `test` are equivalent. It is easy to lose sight of this fact. Mostly it happens when you have experience in using other languages. This mismatch between expected and real behavior leads to errors.

One of the most common mistakes of using the `[` operator is the missing space between the bracket and the following character. Then the `if` condition becomes like this:

```
1 if ["$var" -eq 5]
2 then
3     echo "The variable equals 5"
4 fi
```

If you replace the `[` bracket with the `test` call, the error becomes obvious:

```
1 if test"$var" -eq 5
2 then
3     echo "The variable equals 5"
4 fi
```

The space between a command name and its parameters is mandatory in both Bash and Bourne shells.

Let's come back to the question about comparison signs in the `[[` operator. Imagine the following test call:

```
test "$var" > 5
```

The `>` symbol is a short form of the redirect operator `1>`. Therefore, the `test` call does the following steps:

1. It calls the `test` built-in and passes the `var` variable there.
2. It redirects the `test` output to the file named `5` in the current directory.

We expect another behavior, right? The `>` symbol should be a comparison sign. Such errors are easy to make and hard to detect. Shell developers want to prevent them. Therefore, they introduced two-letter comparison operations. The `[[` Bash operator inherited these operations. It was done for backward compatibility with the Bourne shell.

Suppose that the `[[` operator replaces two-letter operations with comparison signs. You have the legacy code written in Bourne shell. You want to port it to Bash. The legacy code has the following `if` statement:

```
1 if [ "$var1" -gt 5 -o 4 -lt "$var2" ]
2 then
3   echo "The var1 variable is greater than 5 or var2 is less than 4"
4 fi
```

Here you should replace the `-gt` operation to `>` and `-lt` to `<`. It is easy to make a mistake while doing that. It is much simpler to add an extra parenthesis at the beginning and end of the Boolean expression. This idea answers our question.

You can use comparison signs for strings when working with the `[[` operator. Why is there no backward compatibility issue in this case? The first version of the `test` utility did not support the lexicographic comparison of strings. Therefore, the utility did not have comparison signs `<` and `>`. They appeared in the extension of POSIX standard later. The standard allows comparison signs for strings only. It was too late to add them for numbers because of the legacy code amount. According to the standard, you should escape comparison signs like this: `/<` and `/>`. Then these signs came to the `[[` operator in Bash. You do not need to apply escape symbols for them there.

Exercise 3-5. The `[[` operator

Write a script to compare two directories named `"dir1"` and `"dir2"`.

The script should print all files from one directory that absent in another one.

Case Statement

A program that follows a conditional algorithm chooses its actions depending on the values of variables. If some variable has one value, the program does one thing. Otherwise, it does something else. The condition statements of programming language provide such a behavior.

We have considered the `if` statement. There is another conditional statement in Bash called `case`. It is more convenient than `if` in some cases.

An example will help us to compare `if` and `case` statements. Suppose that you are writing a script for archiving documents. The script has three operating modes:

1. Archiving without compression.
2. Archiving with compression.
3. Unarchiving.

You can choose the mode by passing a command-line option to the script. Table 3-12 shows the possible options.

Table 3-12. Options of the archiving script

Option	Operating mode
-a	Archiving without compression
-c	Archiving with compression
-x	Unarchiving



Always follow the [POSIX convention](#)³⁰⁹ and its [GNU extension](#)³¹⁰ when choosing options and parameters format for your scripts. Then users can learn them faster.

You can check the script option using the `if` statement. Listing 3-12 shows how this solution looks like.

Listing 3-12. The script for archiving documents

```

1  #!/bin/bash
2
3  operation="$1"
4
5  if [[ "$operation" == "-a" ]]
6  then
7      bsdtar -c -f documents.tar ~/Documents
8  elif [[ "$operation" == "-c" ]]
9  then
10     bsdtar -c -j -f documents.tar.bz2 ~/Documents
11 elif [[ "$operation" == "-x" ]]
12 then
13     bsdtar -x -f documents.tar*
14 else

```

³⁰⁹https://www.gnu.org/software/libc/manual/html_node/Argument-Syntax.html

³¹⁰https://www.gnu.org/prep/standards/html_node/Command_Line-Interfaces.html

```
15 echo "Invalid option"
16 exit 1
17 fi
```

The `$1` position parameter keeps the script option. It is always better to store it in a well-named variable. It is the `operation` variable in our example. Depending on its value, the `if` statement chooses parameters for the `bsdtar` call.

Now let's replace the `if` statement with the `case` one. Listing 3-13 shows the result.

Listing 3-13. The script for archiving documents

```
1 #!/bin/bash
2
3 operation="$1"
4
5 case "$operation" in
6   "-a")
7     bsdtar -c -f documents.tar ~/Documents
8     ;;
9
10  "-c")
11    bsdtar -c -j -f documents.tar.bz2 ~/Documents
12    ;;
13
14  "-x")
15    bsdtar -x -f documents.tar*
16    ;;
17
18  *)
19    echo "Invalid option"
20    exit 1
21    ;;
22 esac
```

Suppose that you saved the script in the `archiving-case.sh` file. Now you can call it in one of the following ways:

```
1 ./archiving-case.sh -a
2 ./archiving-case.sh -c
3 ./archiving-case.sh -x
```

If you pass any other parameter to the script, it prints the error message and terminates. The same happens if you skip the parameter.



Always call the `exit` command when handling errors in your script. The script should return the non-zero exit status in this case.

The `case` statement compares a string with a list of patterns. Each pattern has a corresponding code block. Bash executes this block when its pattern and the string match each other.

Each case block consists of the following elements:

1. A pattern or a list of patterns separated by vertical bars.
2. Right parenthesis.
3. A code block.
4. Two semicolons. They mark the end of the code block.

Bash checks patterns of the `case` blocks one by one. If the string matches the first pattern, Bash executes its code block. Then it skips other patterns and executes the command that follows the `case` statement.

The `*` pattern without quotes matches any string. It is usually placed at the end of the `case` statement. The corresponding code block handles cases when none of the patterns match the string. It usually indicates an error.

At first sight, it may seem that the `if` and `case` statements are equivalent. They are different but allow you to achieve the same behavior in some cases.

Let's compare the `if` and `case` statements of Listings 3-12 and 3-13. First, we will write them in a general form. Here is the result for the `if` statement:

```
1  if CONDITION_1
2  then
3    ACTION_1
4  elif CONDITION_2
5  then
6    ACTION_2
7  elif CONDITION_3
8  then
9    ACTION_3
10 else
11   ACTION_4
12 fi
```

The `case` statement looks this way:

```
1 case STRING in
2   PATTERN_1)
3     ACTION_1
4     ;;
5
6   PATTERN_2)
7     ACTION_2
8     ;;
9
10  PATTERN_3)
11    ACTION_3
12    ;;
13
14  PATTERN_4)
15    ACTION_4
16    ;;
17 esac
```

The differences between the constructs are evident now. First, the `if` condition checks the results of a Boolean expression. The `case` statement compares a string with several patterns. Therefore, it makes no sense to pass a Boolean expression to the `case` condition. Doing that, you handle two cases only: when the expression is “true” and “false”. The `if` statement is more convenient for such checking.

The second difference between `if` and `case` is the number of conditions. Each branch of the `if` statement evaluates an individual Boolean expression. They are independent of each other in general. The expressions check the same variable in our example, but this is a particular case. The `case` statement checks one string that you pass to it.

The `if` and `case` statements are fundamentally different. You cannot exchange one for another in your code. Use an appropriate statement depending on the nature of your checking. The following questions will help you to make the right choice:

- How many conditions should you check? Use `if` for checking several conditions.
- Would it be enough to check one string only? Use `case` when the answer is yes.
- Do you need compound Boolean expressions? Use `if` when the answer is yes.

When you use the `case` statement, you can apply one of two possible delimiters between the code blocks:

1. Two semicolons `;;`.
2. Semicolons and ampersand `;&`.

The ampersand delimiter is allowed in Bash, but it is not part of the POSIX standard. When Bash meets this delimiter, it executes the following code block without checking its pattern. It can be useful when you want to start executing an algorithm from a specific step. Also, the ampersand delimiter allows you to avoid code duplication in some cases.

Here is an example of a code duplication problem. Suppose that you have the script that archives PDF documents and copies the resulting file. An input parameter of the script chooses an action to do. For example, the `-a` option means archiving and `-c` means copying. The script should always copy the archiving result. You get a code duplication in this case.

Listing 3-14 shows the archiving script. The `case` statement there has two `cp` calls that are the same.

Listing 3-14. The script for archiving and copying PDF documents

```
1  #!/bin/bash
2
3  operation="$1"
4
5  case "$operation" in
6    "-a")
7      find Documents -name "*.pdf" -type f -print0 | xargs -0 bsdtar -c -j -f document\
8  s.tar.bz2
9      cp documents.tar.bz2 ~/backup
10     ;;
11
12    "-c")
13     cp documents.tar.bz2 ~/backup
14     ;;
15
16    *)
17     echo "Invalid option"
18     exit 1
19     ;;
20 esac
```

You can avoid code duplication by adding the `&` separator between the `-a` and `-c` code blocks. Listing 3-15 shows the changed script.

Listing 3-15. The script for archiving and copying PDF documents

```
1  #!/bin/bash
2
3  operation="$1"
4
5  case "$operation" in
6    "-a")
7      find Documents -name "*.pdf" -type f -print0 | xargs -0 bsdtar -c -j -f document\
8  s.tar.bz2
9      ;&
10
11   "-c")
12     cp documents.tar.bz2 ~/backup
13     ;;
14
15   *)
16     echo "Invalid option"
17     exit 1
18     ;;
19  esac
```

The `;&` delimiter can help you in some cases. However, use it carefully. You can easily confuse the delimiters when reading the code. This way, you may misread `;;` instead of `;&` and misunderstand the case statement.

Alternative to the Case Statement

The case statement and the associative array solve a similar task. The array makes the relationship between data (key-value). The case statement does the same between data and commands (value-action).

Usually, it is more convenient to handle data than code. Data are easier for modifying and checking for correctness. Therefore, it is worth to replace the case statement with an associative array in some cases. Converting data into code is easy to do in Bash comparing with other programming languages.

Here is an example of replacing case with an array. Suppose that you want to write a wrapper script for the archiving utilities. It receives several command-line parameters. The first one defines if the script calls the `bsdtar` or `tar` utility.

Listing 3-16 shows the script. It handles the command-line parameters in the case statement.

Listing 3-16. The wrapper script for the archiving utilities

```
1  #!/bin/bash
2
3  utility="$1"
4
5  case "$utility" in
6    "-b"|"--bsdtar")
7      bsdtar "${@:2}"
8      ;;
9
10   "-t"|"--tar")
11     tar "${@:2}"
12     ;;
13
14   *)
15     echo "Invalid option"
16     exit 1
17     ;;
18 esac
```

Here you see three code blocks in the `case` statement. The script executes the first block when the utility variable matches the string `-b` or `--bsdtar`. The script executes the second block when the variable matches `-t` or `--tar`. The third block handles an invalid input parameter.

Here is an example of how you can launch the script:

```
./tar-wrapper.sh --tar -cvf documents.tar.bz2 Documents
```

This call forces the script to choose the `tar` utility for archiving the `Documents` directory. If you want to use the `bsdtar` utility instead, replace the `--tar` option with `-b` or `--bsdtar` this way:

```
./tar-wrapper.sh -b -cvf documents.tar.bz2 Documents
```

The script handles the first positional parameter on its own. It passes all the following parameters to the archiving utility. We use the `$@` parameter for doing that. It is not an array, but it supports the array-like syntax for accessing several elements. The archiving utility receives all elements of the `$@` parameter starting from the second one.

Now let's rewrite our wrapper script using the associative array. First, we should consider the Bash mechanisms for converting data into commands. If you want to apply such a mechanism, you should store the command and its parameters into the variable. Then Bash expands the variable somewhere in the script and executes the command.

Here is an example of how to convert data into a command. For the first time, we will do it in the shell but not in the script. The first step is declaring the variable like this:

```
ls_command="ls"
```

Now the `ls_command` variable stores the command to call the `ls` utility. You can use the variable this way:

```
ls_command
```

This command calls the `ls` utility without parameters. How does it work? Bash inserts the value of the `ls_command` variable. Then the command becomes like this:

```
ls
```

Bash executes the resulting `ls` command after the parameter expansion.

Why don't we use double quotes when expanding the `ls_command` variable? One small change would help us to answer this question. Let's add an option to the `ls` utility call. Here is the `ls_command` variable declaration for this case:

```
ls_command="ls -l"
```

The parameter expansion with double quotes leads to the error now:

```
1 $ "$ls_command"
2 ls -l: command not found
```

Double quotes cause the problem because they prevent word splitting. Therefore, the command looks this way after the parameter expansion:

```
"ls -l"
```

Bash should call the utility named `"ls -l"` for processing this command. As you remember, the POSIX standard allows spaces in filenames. Therefore, `"ls -l"` is the correct name for an executable. Removing the quotes solves this problem. We meet one of the rare cases when you do not need double quotes for the parameter expansion.

It can happen that you still need double quotes when reading the command from the variable. This issue has a solution. Use the `eval` built-in in this case. It constructs the command from its input parameters. Then Bash does word splitting for the resulting command regardless of double quotes.

Here is the `eval` call for processing our `ls_command` variable:

```
eval "$ls_command"
```



Many Bash manuals claim that it is a bad practice to use `eval` and store commands in variables. It can lead to serious errors and vulnerabilities. This idea is correct in general. Be careful and never pass user-entered data to `eval`.

Now we can rewrite our wrapper script using an associative array. Listing 3-17 shows the result.

Listing 3-17. The wrapper script for the archiving utilities

```

1  #!/bin/bash
2
3  option="$1"
4
5  declare -A utils=(
6    ["-b"]="bsdtar"
7    ["--bsdtar"]="bsdtar"
8    ["-t"]="tar"
9    ["--tar"]="tar")
10
11 if [[ -z "$option" || ! -v utils["$option"] ]]
12 then
13   echo "Invalid option"
14   exit 1
15 fi
16
17 ${utils["$option"]} "${@:2}"

```

Here, the `utils` array stores matching between the script's options and utility names. Using the array, we construct a call of the right utility. The following command does that:

```
${utils["$option"]} "${@:2}"
```

Bash reads the utility name from the `utils` array. The `option` variable provides the element's key. If you pass the wrong option to the script, the corresponding key does not present in `utils`. Then Bash inserts an empty string after the parameter expansion. It leads to an error. The `if` statement checks the `option` variable and prevents this error.

The `if` statement checks two Boolean expressions:

1. The `option` variable is not empty.
2. The `utils` array has a key that equals the `option` variable.

The second expression uses the `-v` option of the `[[` operator. It checks if the variable has been declared. This option has one pitfall. If you have declared the variable and assigned an empty string, the expression result equals true. Please remember about this behavior.

Our example with the wrapper script shows that replacing the `case` statement with the associative array makes your code cleaner. Always consider if this option fits your case when writing scripts.

Exercise 3-6. The `case` statement

There are two configuration files in the user's home directory:

`".bashrc-home"` and `".bashrc-work"`.

Write a script to switch between them.

You can do that by copying one of the files to the path `"~/bashrc"` or creating a symbolic link.

Solve the task with the `"case"` statement first.

Then replace the `"case"` statement with the associative array.

Arithmetic Expressions

Bash allows you to do calculations on integers. You can apply the following operations there: addition, subtraction, multiplication and division. Besides that, Bash provides bitwise and logical operators. You will use them often when programming.

Bash does not support floating-point arithmetic. If you need this arithmetic in your scripts, please use the `bc`³¹¹ or `dc`³¹² calculator.

Integer Representation

The first question we should consider is integers' representation in the computer's memory. This knowledge helps you to get mathematical operations in Bash.

You already know `integers`³¹³ from mathematic. They do not have a fractional component and can be positive or negative. Programming languages with the static type system provide an `integer`³¹⁴ type of variables. You should use this type whenever you need to operate integers.

You can be very precise in programming and specify if the integer variable is positive only. The integer of this kind is called **unsigned**. If it can become positive and negative, the variable is called **signed**.

There are three common ways of representing integers in computer memory:

- **Sign-magnitude representation**³¹⁵ (SMR)
- **Ones' complement**³¹⁶

³¹¹[https://en.wikipedia.org/wiki/Bc_\(programming_language\)](https://en.wikipedia.org/wiki/Bc_(programming_language))

³¹²[https://en.wikipedia.org/wiki/Dc_\(computer_program\)](https://en.wikipedia.org/wiki/Dc_(computer_program))

³¹³<https://en.wikipedia.org/wiki/Integer>

³¹⁴[https://en.wikipedia.org/wiki/Integer_\(computer_science\)](https://en.wikipedia.org/wiki/Integer_(computer_science))

³¹⁵https://en.wikipedia.org/wiki/Signed_number_representations#Sign-and-magnitude_method

³¹⁶https://en.wikipedia.org/wiki/Signed_number_representations#Ones'_complement

- **Two's complement**³¹⁷

Sign-Magnitude Representation

All numbers in the computer's memory are represented in binary form. It means that the computer stores them as a sequence of zeros and ones. A number representation defines how to interpret this sequence.

First, we consider the simplest numbers representation that is the sign-magnitude representation or SMR. There are two options to use it:

1. To store positive integers (unsigned).
2. To store both positive and negative integers (signed).

The computer allocates a fixed block of memory for any number. When you choose the first option of SMR, all allocated memory bits are used in the same way. They store the value of the number. Table 3-13 shows several examples of how it looks like.

Table 3-13. Sign-magnitude representation of the unsigned integers

Decimal	Hexadecimal	SMR
0	0	0000 0000
5	5	0000 0101
60	3C	0011 1100
110	6E	0110 1110
255	FF	1111 1111

Suppose that the computer allocates one byte of memory for a number. Then you can store unsigned integers from 0 to 255 there using SMR.

The second option of SMR allows you to store signed integers. In this case, the highest bit keeps the integer's sign. Therefore, there are fewer bits left for the value of the number.

Here is an example. Suppose that a computer allocates one byte to store the signed integer. One bit is reserved to indicate the positive or negative sign. Then you have seven bits only to store the number itself.

Table 3-14 shows the sign-magnitude representation of several signed integers.

³¹⁷https://en.wikipedia.org/wiki/Signed_number_representations#Two's_complement

Table 3-14. The sign-magnitude representation of the signed integers

Decimal	Hexadecimal	SMR
-127	FF	1111 1111
-110	EE	1110 1110
-60	BC	1011 1100
-5	85	1000 0101
-0	80	1000 0000
0	0	0000 0000
5	5	0000 0101
60	3C	0011 1100
110	6E	0110 1110
127	7F	0111 1111

The highest (leftmost) bit of all negative numbers equals one. It equals zero for positive numbers. Because of the sign, it is impossible to store numbers greater than 127 in one byte. The minimum allowed negative number is -127 for the same reason.

There are two reasons why SMR is not widespread nowadays:

1. Arithmetic operations on negative numbers complicate the processor architecture. A processor module for adding positive numbers is not suitable for negative numbers.
2. There are two representations of zero: positive (0000 0000) and negative (1000 0000). It complicates the comparison operation because these values are not equal in memory.

Take your time and try to get well how the SMR works. Without getting it, you won't understand the other two ways of representing integers.

Ones' Complement

SMR has two disadvantages. They led to technical issues when computer engineers had used this representation in practice. Therefore, the engineers started looking for an alternative approach to store numbers in memory. This way, they came to ones' complement representation.

The first problem of SMR is related to operations on negative numbers. The ones' complement solves it. Let's consider this problem in detail.

The example will explain to you what exactly happens when you operate negative numbers in SMR. Suppose that you want to add integers 10 and -5. First, you should write them in SMR. Assume that each integer occupies one byte in computer memory. Then you represent them this way:

```
10 = 0000 1010
-5 = 1000 0101
```

Now the question arises. How does the processor add these two numbers? Any modern processor has a standard module called **adder**³¹⁸. It adds two numbers in a bitwise manner. If you apply it for our task, you get the following result:

$$10 + (-5) = 0000\ 1010 + 1000\ 0101 = 1000\ 1111 = -15$$

This result is wrong. It means that the adder cannot add numbers in SMR. The calculation mistake happens because the adder handles the highest bit of the number wrongly. This bit stores the integer's sign.

There are two ways for solving the problem:

1. Add a special module to the processor. It should process operations on negative integers.
2. Change the integer representation in memory. The representation should fit the adder logic when it operates negative integers.

The development of computer technology followed the second way. It is cheaper than complicating the processor architecture.

The ones' complement reminds SMR. The sign of the integer occupies the highest bit. The remaining bits store the number. The difference with SMR is all bits of a negative number are inverted. It means zeros become ones, and ones become zeros. Bits of positive numbers are not inverted.

Table 3-15 shows the ones' complement representation of some numbers.

Table 3-15. The ones' complement of the signed integers

Decimal	Hexadecimal	Ones' Complement
-127	80	1000 0000
-110	91	1001 0001
-60	C3	1100 0011
-5	FA	1111 1010
-0	FF	1111 1111
0	0	0000 0000
5	5	0000 0101
60	3C	0011 1100
110	6E	0110 1110
127	7F	0111 1111

The memory capacity is the same when you switch from SMR to the ones' complement. One byte can store numbers from -127 to 127 in both cases.

How did inverting the bits of negative numbers solve the adder problem? Let's come back to our example with adding 10 and -5. First, you should represent them in the ones' complement this way:

³¹⁸[https://en.wikipedia.org/wiki/Adder_\(electronics\)](https://en.wikipedia.org/wiki/Adder_(electronics))

```
10 = 0000 1010
-5 = 1111 1010
```

When you apply the standard adder, you get the following result:

```
10 + (-5) = 0000 1010 + 1111 1010 = 1 0000 0100
```

The addition led to the overflow because the highest one does not fit into one byte. It is discarded in this case. Then the result becomes like this:

```
0000 0100
```

The discarded highest one affects the final result. You need a second calculation step to take it into account. There you add the discarded value to the result this way:

```
0000 0100 + 0000 0001 = 0000 0101 = 5
```

You got the correct result of adding numbers 10 and -5.

If the addition results in a negative number, the second calculation step is unnecessary. Here is an example of adding numbers -7 and 2. First, write them in the ones' complement representation:

```
-7 = 1111 1000
2 = 0000 0010
```

Then add the numbers:

```
-7 + 2 = 1111 1000 + 0000 0010 = 1111 1010
```

The highest bit equals one. It means that you got a negative number. Therefore, you should skip the second calculation step.

Let's check if the result is correct. You can convert it from ones' complement to SMR for convenience. Invert bits of the number for doing that. The highest bit should stay unchanged. Here is the result:

```
1111 1010 -> 1000 0101 = -5
```

This is the correct result of adding -7 and 2 and.

The ones' complement solves the first of two SMR's problems. Now the CPU adder can operate any signed integers. This solution has one disadvantage. Addition requires two steps when you get a positive number in the result. This drawback slows down computations.

SMR has another problem with zeros. It represents zero in two ways. The ones' complement does not solve it.

Two's Complement

The two's complement solves both problems of SMR. First, it allows the CPU adder to operate negative numbers in one step. You need two steps for that when using the ones' complement. Second, the two's complement has only one way to represent zero.

Positive integers in the two's complement and SMR look the same. The highest bit equals zero there. The remaining bits store the number.

Negative integers in the two's complement have the highest bit equal to one. The rest bits are inverted the same way as in the ones' complement. The only difference is you need to add one to the negative number after inverting its bits.

Table 3-16 shows the two's complement representation of some numbers.

Table 3-16. The two's complement of the signed integers

Decimal	Hexadecimal	Two's Complement
-128	80	1000 0000
-127	81	1000 0001
-110	92	1001 0010
-60	C4	1100 0100
-5	FB	1111 1011
0	0	0000 0000
5	5	0000 0101
60	3C	0011 1100
110	6E	0110 1110
127	7F	0111 1111

The memory capacity increases by one unit when you switch from SMR to the two's complement. It happens because there is only one way to represent zero. Therefore, one byte can store the numbers from -128 to 127.

Here is an example of adding numbers 14 and -8. First, you should write them in the two's complement this way:

```
14 = 0000 1110
-8 = 1111 1000
```

Now you can add the numbers:

```
14 + (-8) = 0000 1110 + 1111 1000 = 1 0000 0110
```

The addition leads to the overflow. The highest one does not fit into a single byte. The rest bits make a positive number. It means that you should discard the highest one. This way, you get the following result:

```
0000 0110 = 6
```

When addition gives a negative result, you should not discard the highest bit. Here is an example. You want to add numbers -25 and 10. They look this way in the two's complement:

```
-25 = 1110 0111
10 = 0000 1010
```

This is the result of the addition:

```
-25 + 10 = 1110 0111 0000 1010 = 1111 0001
```

Now you should convert the result to decimal. First, convert it from two's complement to the ones' complement. Second, convert the result to SMR. You get the following sequence of conversions:

```
1111 0001 - 1 = 1111 0000 -> 1000 1111 = -15
```

When converting from the ones' complement to SMR, you invert all bits except the highest one. This way, you get the correct result of adding -25 and 10.

The two's complement allows the CPU adder to operate negative numbers. Moreover, such calculations require a single step only. Therefore, there is no performance loss, unlike the ones' complement case.

The two's complement solves the problem of zero representation. It has only one way to represent it. Zero is the number with all bits zeroed. Therefore, you do not have issues with comparing numbers anymore.

All modern computers use the two's complement representation to store numbers in memory.

Exercise 3-7. Arithmetic operations with numbers in the two's complement representation

Represent the following integers in the two's complement and add them:

```
* 79 + (-46)
* -97 + 96
```

Represent the following two-byte integers in the two's complement and add them:

```
* 12868 + (-1219)
```

Converting Numbers

You have learned how a computer represents numbers in memory. Would you need this knowledge in practice?

Modern programming languages take care of converting numbers to the correct format. For example, you declare a signed integer variable in decimal notation. You do not need to worry about how the computer stores it in memory. It stores all numbers in two's complement representation automatically.

There are cases when you want to treat a variable as a set of bits. You declare it as a positive integer in this case. You should operate it in hexadecimal. Please do not convert this variable to decimal. This way, you avoid the problems of converting numbers.

The issue arises when you want to read data from some device. Such a task often occurs in the **system software development**³¹⁹. Specialists of this domain deal with device drivers, OS kernels and their modules, system libraries and network protocol stack.

An example will demonstrate you a problem. Suppose that you are writing a driver for a peripheral device. It measures air temperature cyclically and sends the results to the CPU. Your task is to interpret these data correctly. Unfortunately, the computer cannot do it for you. It happens because the computer and device represent the numbers differently. Therefore, you should write a code that does the conversion. You need to know numbers representation for doing that.

There is another task that requires you to know the two's complement. I am talking about debugging your program. Suppose that the program gives you an unexpectedly large number in the result. If you know the two's complement, you can guess that an integer overflow happens. This helps you find the root cause of the problem.

Operator ((

Bash performs integer arithmetic in **math context**.



The syntax of math context resembles the C language. The idea behind it is to make Bash easier for learning for programmers who have experience with the C language. Most users of the first Unix versions knew C.

Suppose that you want to store a result of adding two numbers in a variable. You need to declare it using the `-i` attribute and assign a value in the declaration. Here is an example:

```
declare -i var=12+7
```

When processing this declaration, the variable gets value 19 instead of the “12+7” string. It happens because the `-i` attribute forces Bash to apply the mathematical context when handling the variable.

³¹⁹https://en.wikipedia.org/wiki/System_software

There is an option to apply the mathematical context besides the variable declaration. Call the `let` built-in for doing that.

Suppose that you declared the variable without the `-i` attribute. Then the `let` built-in allows you to calculate an arithmetic expression and assign its result to the variable. Here is an example:

```
let text=5*7
```

The `text` variable gets integer 35 after executing this command.

When declaring a variable using the `-i` attribute, you do not need the `let` command. You can calculate the arithmetic expression without it this way:

```
declare -i var  
var=5*7
```

Now the `var` variable equals 35.

Declaring a variable using the `-i` attribute creates the mathematical context implicitly. The `let` built-in does it explicitly. Avoid implicit mathematical contexts whenever it is possible. They can lead to errors. The `-i` attribute does not affect the way of how Bash stores the variable in memory. Instead, it forces Bash to convert strings into numbers every time you operate the variable.

The `let` command allows you to treat a string variable as an integer variable. This way, you can do the following assignments:

```
1 let var=12+7  
2 let var="12 + 7"  
3 let "var = 12 + 7"  
4 let 'var = 12 + 7'
```

All four commands give you the same result. They assign number 19 to the `var` variable.

The `let` built-in takes parameters on input. Each of them should be a valid arithmetic expression. If there are spaces, Bash splits the expression into parts because of word splitting. In this case, `let` computes each part separately. It can lead to errors.

The following command demonstrates the issue:

```
let var=12 + 7
```

Here Bash applies word splitting. It produces three expressions that the `let` built-in receives on input. These are the expressions:

- `var=12`

- +
- 7

When calculating the second expression, `let` reports the error. The plus sign means the arithmetic addition. The addition requires two operands. There are no operands at all in our case.

If you pass several correct expressions to the `let` built-in, it evaluates them one by one. Here are the examples:

```
1 let a=1+1 b=5+1
2 let "a = 1 + 1" "b = 5 + 1"
3 let 'a = 1 + 1' 'b = 5 + 1'
```

The results of all three commands are the same. The `a` variable gets number 2. The `b` variable gets number 6.

If you need to prevent word splitting of the `let` parameters, apply single or double quotes.

The `let` built-in has a synonym that is the `((` operator. Bash skips word splitting when handling everything inside the parentheses. Therefore, you do not need quotes there. Always use the `((` operator instead of the `let` built-in. This way, you will avoid mistakes.



The relationship of the `((` operator and the `let` command resembles that of `test` and the `[[` operator. You should use operators rather than commands in both cases.

The `((` operator has two forms. The first one is called the **arithmetic evaluation**³²⁰. It is a synonym for the `let` built-in. The arithmetic evaluation looks like this:

```
((var = 12 + 7))
```

Here the double opening parentheses replace the `let` keyword. When using the arithmetic evaluation, you need double closing parentheses at the end. When the evaluation succeeds, it returns zero exit status. It returns exit status equals one when it fails. After calculating the arithmetic evaluation, Bash replaces it with its exit status.

The second form of the `((` operator is called the **arithmetic expansion**³²¹. It looks like this:

```
var=$((12 + 7))
```

³²⁰https://wiki.bash-hackers.org/syntax/ccmd/arithmetic_eval

³²¹<https://wiki.bash-hackers.org/syntax/expansion/arith>

Here you put a dollar sign before the `((` operator. In this case, Bash calculates the arithmetic expression and replaces it by its value.



The second form of the `((` operator is a part of the POSIX standard. Use it for writing portable code. The first form of the operator is available in Bash, ksh and zsh interpreters only.

You can skip the dollar sign before variable names inside the `((` operator. Bash evaluates them correctly in this case. For example, the following two expressions are equivalent:

```
1 a=5 b=10
2 result=$(( $a + $b ))
3 result=$(( a + b ))
```

Both expressions assign number 15 to the `result` variable.

Do not use the dollar sign inside the `((` operator. It makes your code clearer.



Bash has an obsolete form of the arithmetic expansion that is the `[$]` operator. Never use it. Besides that, there is the GNU utility called `expr`. It calculates arithmetic expressions too. Bash uses it when launching old Bourne Shell scripts. You should never use `expr`.

Table 3-17 shows the operations that you can perform in the arithmetic expression.

Table 3-17. The operations of the arithmetic expression

Operation	Description	Example
Calculations		
*	Multiplication	echo "\$((2 * 9)) = 18"
/	Division	echo "\$((25 / 5)) = 5"
%	The remainder of division	echo "\$((8 % 3)) = 2"
+	Addition	echo "\$((7 + 3)) = 10"
-	Subtraction	echo "\$((8 - 5)) = 3"
**	Exponentiation	echo "\$((4**3)) = 64"
Bitwise operations		

Table 3-17. The operations of the arithmetic expression

Operation	Description	Example
~	Bitwise NOT	echo "\$((~5)) = -6"
<<	Bitwise left shift	echo "\$((5 << 1)) = 10"
>>	Bitwise right shift	echo "\$((5 >> 1)) = 2"
&	Bitwise AND	echo "\$((5 & 4)) = 4"
	Bitwise OR	echo "\$((5 2)) = 7"
^	Bitwise XOR ³²²	echo "\$((5 ^ 4)) = 1"
Assignments		
=	Ordinary assignment	echo "\$((num = 5)) = 5"
*=	Multiply and assign the result	echo "\$((num *= 2)) = 10"
/=	Divide and assign the result	echo "\$((num /= 2)) = 5"
%=	Get the remainder of the division and assign it	echo "\$((num %= 2)) = 1"
+=	Add and assign the result	echo "\$((num += 7)) = 8"
-=	Subtract and assign the result	echo "\$((num -= 3)) = 5"
<<=	Do bitwise left shift and assign the result	echo "\$((num <<= 1)) = 10"
>>=	Do bitwise right shift and assign the result	echo "\$((num >>= 2)) = 2"
&=	Do bitwise AND and assign the result	echo "\$((num &= 3)) = 2"
^=	Do bitwise XOR and assign the result	echo "\$((num ^= 7)) = 5"
=	Do bitwise OR and assign the result	echo "\$((num = 7)) = 7"

³²²https://en.wikipedia.org/wiki/Exclusive_or

Table 3-17. The operations of the arithmetic expression

Operation	Description	Example
Comparisons		
<	Less than	<code>((num < 5)) && echo "The \num\" variable is less than 5"</code>
>	Greater than	<code>((num > 5)) && echo "The \num\" variable is greater than 3"</code>
<=	Less than or equal	<code>((num <= 20)) && echo "The \num\" variable is less or equal 20"</code>
>=	Greater than or equal	<code>((num >= 15)) && echo "The \num\" variable is greater or equal 15"</code>
==	Equal	<code>((num == 3)) && echo "The \num\" variable is equal to 3"</code>
!=	Not equal	<code>((num != 3)) && echo "The \num\" variable is not equal to 3"</code>
Logical operations		
!	Logical NOT	<code>((! num)) && echo "The \num\" variable is FALSE"</code>
&&	Logical AND	<code>((3 < num && num < 5)) && echo "The \num\" variable is greater than 3 but less than 5"</code>
	Logical OR	<code>((num < 3 5 < num)) && echo "The \num\" variable is less than 3 or greater than 5"</code>
Other operations		
num++	Postfix increment	<code>echo "\$((num++))"</code>

Table 3-17. The operations of the arithmetic expression

Operation	Description	Example
num--	Postfix decrement	echo "\$((num--))"
++num	Prefix increment	echo "\$((++num))"
--num	Prefix decrement	echo "\$((--num))"
+num	Unary plus or multiplication of a number by 1	a=\$((+num))"
-num	Unary minus or multiplication of a number by -1	a=\$((-num))"
CONDITION ? ACTION_1 : ACTION_2	Ternary operator ³²³	a=\$((b < c ? b : c))
ACTION_1, ACTION_2	The list of expressions	((a = 4 + 5, b = 16 - 7))
(ACTION_1)	Grouping of expressions (subexpression)	a=\$(((4 + 5) * 2))

Bash performs all operations of the arithmetic expression in order of their priorities. The operations with a higher priority come first.

Table 3-18 shows the priority of operations.

Table 3-18. Priority of operations of the arithmetic expression

Priority	Operation	Description
1	(ACTION_1)	Grouping of expressions
2	num++, num--	Postfix increment and decrement
3	++num, --num	Prefix increment and decrement
4	+num, -num	Unary plus and minus
5	~, !	Bitwise and logical NOT
6	**	Exponentiation
7	*, /, %	Multiplication, division and the remainder of division

³²³<https://en.wikipedia.org/wiki/%3F>:

Table 3-18. Priority of operations of the arithmetic expression

Priority	Operation	Description
8	+, -	Addition and subtraction
9	<<, >>	Bitwise shifts
10	<, <=, >, >=	Comparisons
11	==, !=	Equal and not equal
12	&	Bitwise AND
13	^	Bitwise XOR
14		Bitwise OR
15	&&	Logical AND
16		Logical OR
17	CONDITION ? ACTION_1 : ACTION_2	Ternary operator
18	=, *=, /=, %=, +=, -=, <<=, >>=, &=, ^=, =	Assignments
19	ACTION_1, ACTION_2	The list of expressions

You can change the order of operations execution using parentheses “()”. Their contents are called **subexpression**. It has the highest priority for Bash. If there are several subexpressions, Bash calculates them in the left-to-right order.

Suppose your code uses a numeric constant. You can specify its value in any numeral system. Use a prefix for choosing the system. Table 3-19 shows the list of allowable prefixes.

Table 3-19. The prefixes for numeral systems

Prefix	Numeral System	Example]
0	Octal	echo "\$((071)) = 57"
0x	Hexadecimal	echo "\$((0xFF)) = 255"
0X	Hexadecimal	echo "\$((0XFF)) = 255"
<base>#	The numeral system with a base from 2 to 64	echo "\$((16#FF)) = 255" echo "\$((2#101)) = 5"

When printing a number, Bash always converts it to decimal. The `printf` built-in changes the format of the number on output. You can use it this way:

```
printf "%x\n" 250
```

This command prints number 250 in hexadecimal.

The `printf` built-in handles variables in the same way:

```
printf "%x\n" $var
```

Arithmetic Operations

Let's start with the arithmetic operations because they are the simplest. Programming languages use regular symbols to denote them:

- + addition
- - subtraction
- / division
- * multiplication

There are two more operations that are often used in programming. These are exponentiation and [division with remainder](#)³²⁴.

Suppose that you want to raise the a number to the power of b . You can write it on paper this way: a^b . Here a is the base and b is the exponent. If you want to raise two to the power of seven, you write 2^7 . The same operation in Bash looks like this:

```
2**7
```

Calculating the remainder of a division is a complex but essential operation in programming. Therefore, we should consider it in detail.

Suppose that you have divided one integer by another. You get a fractional number in the result. The division operation produced a [remainder](#)³²⁵ in this case.

Here is an example. Suppose that you want to divide the number 10 (the dividend) by 3 (the divisor). If you round the result, you will get 3.33333 (the quotient). The remainder of the division equals 1 in this case. To find it, you should multiply the divisor 3 by the integer part of the quotient 3 (the incomplete quotient). Then subtract the result from the dividend 10. It gives you the remainder, which equals 1.

Let's write our calculations in formulas. We can introduce the following notation for that:

- a is a dividend
- b is a divisor
- q is an incomplete quotient
- r is a remainder

Using the notation, you get this formula for calculating the dividend:

³²⁴https://en.wikipedia.org/wiki/Euclidean_division

³²⁵<https://en.wikipedia.org/wiki/Remainder>

$$a = b * q + r$$

Move the “ $b * q$ ” multiplication to the left side of the equal sign. Then you get the following formula for finding the remainder:

$$r = a - b * q$$

The right choice of an incomplete quotient q raises questions. Sometimes several numbers fit this role. There is a restriction that helps you to choose the right one. The q quotient should have the value that makes the r remainder’s absolute value less than the b divisor. In other words, it should fulfill the following inequality:

$$|r| < |b|$$

The percent sign denotes the operation of finding the remainder in Bash. Some languages use the same symbol for the **modulo operation**³²⁶. These two operations **are not the same**³²⁷. They give the same results only when the signs of the dividend and the divisor match.

Here is an example of calculating the division remainder and modulo when dividing 19 by 12 and -19 by -12. You will get these results:

$$19 \% 12 = 19 - 12 * 1 = 7$$

$$19 \text{ modulo } 12 = 19 - 12 * 1 = 7$$

$$-19 \% -12 = -19 - (-12) * 1 = -7$$

$$-19 \text{ modulo } -12 = -19 - (-12) * 1 = -7$$

Let’s change the signs of the dividend and divisor. Then you get the following pairs of numbers: 19, -12 and -19, 12. If you calculate the division remainder and modulo for them, you get these results:

$$19 \% -12 = 19 - (-12) * (-1) = 7$$

$$19 \text{ modulo } -12 = 19 - (-12) * (-2) = -5$$

$$-19 \% 12 = -19 - 12 * (-1) = -7$$

$$-19 \text{ modulo } 12 = -19 - 12 * (-2) = 5$$

You see that the remainder and the modulo differ for the same pairs of numbers. It looks strange because you use the same formula for calculating them. The trick happens when you choose the q incomplete quotient. You calculate it this way when finding the division remainder:

³²⁶https://en.wikipedia.org/wiki/Modulo_operation

³²⁷<https://rob.conery.io/2018/08/21/mod-and-remainder-are-not-the-same>

$$q = a / b$$

You should round the result to the lowest [absolute number](#)³²⁸. It means discarding all decimal places.

Calculating the incomplete quotient for finding the modulo depends on the signs of a and b . If the signs are the same, the formula for the quotient stays the same:

$$q = a / b$$

If the signs differ, you should use another formula:

$$q = (a / b) - 1$$

You should round the result to the lowest absolute number in both cases.

When somebody talks about the division remainder r , he usually assumes that both the dividend a and divisor b are positive. That is why programming books often mention the following condition for r :

$$0 \leq r < |b|$$

However, you can get a negative remainder when dividing numbers with different signs. Remember a simple rule: the r remainder always has the same sign as the a dividend. If the signs of r and a differ, you have found the modulo but not the division remainder.

Always keep in mind the difference between the division remainder and modulo. Some programming languages provide the `%` operator that calculates the remainder. Other languages have the same operator, but it calculates the modulo. It leads to confusion.

If you are unsure of your calculations, check them. The `%` operator of Bash always computes the division remainder. For example, you want to find the remainder of dividing 32 by -7. This command does it:

```
echo $((32 % -7))
```

This command prints the division remainder that equals four.

Now let's find the modulo for the same pair of numbers. You can use the [online calculator](#)³²⁹ for that. First, enter the dividend 32 in the "Expression" field. Then enter the divisor 7 in the "Modulus" field. Finally, click the "CALCULATE" button. You will get two results:

- The "Result" field shows 4.
- The "Symmetric representation" field shows -3.

³²⁸https://en.wikipedia.org/wiki/Absolute_value

³²⁹<https://planetcalc.com/8326>

The first number 4 is the division remainder. The second number -3 is the modulo that you are looking for.

When do you need the division remainder in programming? It is often used for checking an integer for parity. For example, there is a widespread approach that controls the integrity of transmitted data in computer networks. It is called the **parity bit**³³⁰ check. You need to calculate the division remainder there.



To check a number for parity, you should calculate the remainder of its division by 2. If the remainder is zero, then the number is even. Otherwise, it is odd.

Another task that requires calculating the division remainder is converting the time units. For example, you want to convert 128 seconds into minutes.

First, you count the number of minutes in 128 seconds. Divide 128 by 60 to do that. The result equals 2, which is the incomplete quotient. It means that 128 seconds contains 2 minutes.

The second step is calculating the remainder of dividing 128 by 60. This way, you will find the remaining seconds to add. Here is the result:

$$r = 128 - 60 * 2 = 8$$

The remainder equals 8. It means that 128 seconds equals two minutes and eight seconds.

Calculating the remainder is useful when working with loops. Suppose that you want to act on every 10th iteration of the loop. Then you need to check the remainder of dividing the loop counter by 10. If the remainder is zero, then the current iteration is a multiple of 10. Thus, you should act on this iteration.

The modulo operation is widely used in **cryptology**³³¹.

Exercise 3-8. Modulo and the division remainder

Calculate the division remainder and modulo for the following pairs of numbers:

* $1697 \% 13$

* 1697 modulo 13

* $772 \% -45$

* 772 modulo -45

* $-568 \% 12$

* -568 modulo 12

* $-5437 \% -17$

* -5437 modulo -17

³³⁰https://en.wikipedia.org/wiki/Parity_bit

³³¹[https://en.wikipedia.org/wiki/Modulo_operation#Properties_\(identities\)](https://en.wikipedia.org/wiki/Modulo_operation#Properties_(identities))

Bitwise operations

Bitwise operations³³² handle each bit of a number individually. You will use them often when programming. Let's consider how they work in detail.

Bitwise negation

First, we will consider the simplest bitwise operation that is the negation. It is also called bitwise NOT. The tilde symbol indicates this operation in Bash.

When doing bitwise negation, you swap the value of each bit of an integer. It means that you replace each 1 to 0 and vice versa.

Here is an example of doing bitwise NOT for number 5:

```
5 = 101
~5 = 010
```

The bitwise NOT is a simple operation when we are talking about mathematics. However, there are pitfalls when using it in programming. You should keep in mind two things:

- How many bytes does the number occupy?
- What is the number's representation in memory?

Suppose that the two-byte variable stores the number 5. Then it looks like this in memory:

```
00000000 00000101
```

When you apply the bitwise NOT for this variable, you get the following result:

```
11111111 11111010
```

What do these bits mean? If you store them to the variable of the unsigned integer type, you get the number 65530. If the variable is a signed integer, it equals -6. You can check it by converting the two's complement representation to decimal.

Various Bash built-ins and operators represent integers in different ways. For example, `echo` always outputs numbers as signed integers. The `printf` command allows you to choose between a signed and unsigned integers.

There are no types in the Bash language. Bash stores all scalar variables as strings. It converts strings to integers right before inserting them into arithmetic expressions. The number interpretation (signed or unsigned) depends on the context.

Bash allocates 64 bits of memory space for each integer, regardless of its sign. Table 3-20 shows maximum and minimum allowed integers in Bash.

³³²https://en.wikipedia.org/wiki/Bitwise_operation

Table 3-20. Maximum and minimum allowed integers in Bash

Integer	Hexadecimal	Decimal
Maximum positive signed	7FFFFFFFFFFFFFFF	9223372036854775807
Minimum negative signed	8000000000000000	-9223372036854775808
Maximum unsigned	FFFFFFFFFFFFFFF	18446744073709551615

The following examples show you how Bash interprets integers in the `((` operator, `echo` and `printf` built-ins:

```

1 $ echo $((16#FFFFFFFFFFFFFFF))
2 -1
3
4 $ printf "%11u\n" $((16#FFFFFFFFFFFFFFF))
5 18446744073709551615
6
7 $ if ((18446744073709551615 == 16#FFFFFFFFFFFFFFF)); then echo "ok"; fi
8 ok
9
10 $ if ((-1 == 16#FFFFFFFFFFFFFFF)); then echo "ok"; fi
11 ok
12
13 $ if ((18446744073709551615 == -1)); then echo "ok"; fi
14 ok

```

The last example of comparing the numbers 18446744073709551615 and -1 confirms that Bash stores signed and unsigned integers the same way in memory. However, it interprets them depending on the context.

Let's come back to the bitwise negation of the number 5. Bash gives you the result 0xFFFFFFFFFFFFFFFA in hexadecimal. You can print it as a positive or negative integer this way:

```

1 $ echo $((~5))
2 -6
3
4 $ printf "%11u\n" $((~5))
5 18446744073709551610

```

The numbers 18446744073709551610 and -6 are equal for Bash. It happens because all their bits in memory are the same.

Exercise 3-9. Bitwise NOT

Apply bitwise NOT for the following unsigned two-byte integers:

- * 56
- * 1018
- * 58362

Repeat the calculations for the case when these are the signed integers.

Bitwise AND, OR and XOR

The bitwise AND operation resembles the logical AND. The result of the logical AND is “true” when both operands are “true”. Any other operands lead to the “false” result.

The bitwise AND operate the numbers instead of Boolean expressions. These are steps to perform the bitwise AND manually:

1. Represent the numbers in the two’s complement.
2. If one number has fewer bits than another, add zeros to its left side.
3. Take the bits of the numbers in the same position and apply the logical AND for them.

Here is an example. You want to calculate the bitwise AND for numbers 5 and 3. First, you should represent their absolute values in binary like this:

```
5 = 101
3 = 11
```

The number 3 has fewer bits than 5. Therefore, you have to add an extra zero on its left side. This way, you get the following representation of the number 3:

```
3 = 011
```

You should convert a number in the two’s complement if it is negative.

Now you should apply the logical AND for each pair of bits of the numbers. You can write the numbers in columns for convenience:

```

101
011
---
001

```

The result equals 001. You can translate it in decimal this way:

```
001 = 1
```

It means that the bitwise AND for numbers 5 and 3 produces 1.

The ampersand sign denotes the bitwise AND operation in Bash. For example, the following command performs our calculations and prints the result:

```
echo $((5 & 3))
```

The bitwise OR operation works similarly as bitwise AND. The only difference is it applies logical OR instead of AND for each pair of bits.

Here is an example. Suppose that you need to calculate the bitwise OR for the numbers 10 and 6. First, you should write them in binary this way:

```

10 = 1010
6  = 110

```

The number 6 is one bit shorter than 10. Then you should extend it by zero like this:

```
6 = 0110
```

Now you perform the logical OR on each pair of bits of the numbers:

```

1010
0110
----
1110

```

The last step is converting the result to decimal:

```
1110 = 14
```

The number 14 is the result of the bitwise OR.

The vertical bar denotes the bitwise OR in Bash. Here is the command to check our calculations:

```
echo $((10 | 6))
```

The bitwise exclusive OR (XOR) operation is similar to the bitwise OR. You should replace the logical OR with the **exclusive OR** when handling the bits of the numbers there. The exclusive OR returns “false” only when both operands have the same values. Otherwise, it returns “true”.

Let’s calculate the exclusive OR for the numbers 12 and 5. First, represent them in binary:

```
12 = 1100
5 = 101
```

Then supplement the number 5 to four bits:

```
5 = 0101
```

Perform the exclusive OR for each pair of bits:

```
1100
0101
----
1001
```

Finally, convert the result to decimal:

```
1001 = 9
```

The caret symbol denotes the exclusive OR in Bash. For example, the following command checks our calculations:

```
echo $((12 ^ 5))
```

Exercise 3-10. Bitwise AND, OR and XOR

Perform bitwise AND, OR and XOR for the following unsigned two-byte integers:

- * 1122 and 908
 - * 49608 and 33036
-

Bit Shifts

A bit shift changes the positions of the bits in a number.

There are three types of bit shifts:

1. Logical
2. Arithmetic
3. Circular

The simplest shift type is the logical one. Let's consider it first.

Any bit shift operation takes two operands. The first one is some integer, which bits you want to shift. The second operand is the number of bits to move.

Here is an algorithm for doing the logical bit shift:

1. Represent the integer in binary.
2. Discard the required amount of bits on the RHS for the right shift and the LHS for the left shift.
3. Append zeroes on the opposite side of the number. This is LHS for the right shift and RHS for the left shift. The amount of zeroes matches the number of shifted bits.

Here is an example. You need to do a logical right shift of the unsigned integer 58 by three bits. The integer occupies one byte of memory.

First, you represent the number in binary:

```
58 = 0011 1010
```

The second step is discarding three bits on the right side of the number this way:

```
0011 1010 >> 3 = 0011 1
```

Finally, you add zeros to the left side of the result:

```
0011 1 = 0000 0111 = 7
```

The number 7 is the result of the shift.

Now let's do the left bit shift of the number 58 by three bits. You will get the following result:


```
0011 1010 << 3 = 1 1010 = 1101 0000 = 208
```

Here you follow the same algorithm as for the right shift. First, discard three leftmost bits. Then add zeros to the right side of the result.

Now let's consider the second type of bit shift that is the arithmetic shift. When you do it to the left side, you follow the logical shift algorithm. The steps are entirely the same.

The arithmetic shift to the right side differs from the logical shift. The first two steps are the same. You should convert the source integer in the two's complement and discard the bits on its right side. Then you append the same amount of bits on the left side. Their value matches the leftmost bit of the integer. If it equals one, you add ones. Otherwise, add zeros. This way, you keep the sign of the integer unchanged after the shifting.

Here is an example. Suppose that you need to do an arithmetic shift of the signed integer -105 to the right by two bits. The integer occupies one byte of memory.

First, you represent the number in the two's complement like this:

```
-105 = 1001 0111
```

Then you shift it to the right by two bits this way:

```
1001 0111 >> 2 -> 1001 01 -> 1110 0101
```

The leftmost bit of the integer equals one in this case. Therefore, you complement the result with ones on the left side.

This way, you get a negative number in the two's complement representation. You can convert it to decimal this way:

```
1110 0101 = 1001 1011 = -27
```

The number -27 is the result of the bit shift operation.

Bash has operators `<<` and `>>`. They do arithmetic bit shifts. The following commands check your calculations:

```

1 $ echo $((58 >> 3))
2 7
3
4 $ echo $((58 << 3))
5 464
6
7 $ echo $((-105 >> 2))
8 -27

```

Bash provides another result for shifting 58 to the left by three bits. It equals 208. It happens because Bash always operates eight-byte integers.

The third type of bit shift is a circular shift. It is used in programming rarely. Therefore, most programming languages do not have built-in operators for circular shifts.

When doing the cyclic bit shift, you should append the discarded bits from one side of the number to another side.

Here is an example of the circular bit shift of the number 58 to the right by three bits:

```
0011 1010 >> 3 = 010 0011 1 = 0100 0111 = 71
```

You should discard bits 010 on the right side of the number. Then add them on the left side.

Exercise 3-11. Bit shifts

Perform arithmetic bit shifts for the following signed two-byte integers:

```

* 25649 >> 3
* 25649 << 2
* -9154 >> 4
* -9154 << 3

```

Using Bitwise Operations

Bitwise operations are widely used in system programming. The specialists of this domain deal with computer networks, device drivers and OS kernels. Translating data from one format to another often happens there.

Here is an example. Suppose you are writing a driver for some peripheral device. The byte order on the device is big-endian. Your computer uses another order, which is little-endian.

The device sends an unsigned integer to the computer. It equals 0xAABB in hexadecimal. Because of the different byte orders, your computer cannot handle the integer as it is. You should convert it to 0xBBAA. Then the computer reads it correctly.

Here are the steps for converting the 0xAABB integer to the computer's byte order:

1. Read the lowest (rightmost) byte of the integer and shift it to the left by eight bits, i.e. one byte. The following Bash command does that:

```
little=$(( (0xAABB & 0x00FF) << 8 ))
```

2. Read the highest (leftmost) byte of the number and shift it to the right by eight bits. Here is the corresponding command:

```
big=$(( (0xAABB & 0xFF00) >> 8 ))
```

3. Combine the highest and lowest bytes with the bitwise OR this way:

```
result=$(( little | big ))
```

Bash wrote the conversion result to the `result` variable. It is equal to `0xBBAA`.

You can replace all three steps by the single Bash command:

```
value=0xAABB
result=$(( ((value & 0x00FF) << 8) | ((value & 0xFF00) >> 8) ))
```

Here is another example of using bitwise operations. You need them for computing bitmasks. You already know file permission masks in the Unix environment. Suppose that a file has the permissions “-rw-r-r-”. It looks like this in binary:

```
0000 0110 0100 0100
```

Suppose that you want to check if the file owner can execute it. You can do that by calculating the bitwise AND for the permission mask and the `0000 0001 0000 0000` number. Here is the calculation:

```
0000 0110 0100 0100 & 0000 0001 0000 0000 = 0000 0000 0000 0000 = 0
```

The result equals zero. It means that the owner cannot execute the file.

Using the bitwise OR, you can set bits of the bitmask. For example, you can allow the owner to execute the file this way:

```
0000 0110 0100 0100 | 0000 0001 0000 0000 = 0000 0111 0100 0100 = -rwxr--r--
```



The bits numbering is from right to left. It starts from zero.

We performed the bitwise OR for the permission mask and the 0000 0001 0000 0000 number. The eighth bit of the number equals one. It changes the eighth bit of the permission mask. The corresponding bit in the mask can have any value. It does not matter because the bitwise OR sets it to one regardless of its current value. If you do not want to change some bits in the permission mask, set the corresponding bits of the number to zero.

The bitwise AND clears bits of the bitmask. For example, let's remove the file owner's permission to write. Here is the calculation:

```
0000 0111 0100 0100 & 1111 1101 1111 1111 = 0000 0101 0100 0100 = -r-xr--r--
```

We set the ninth bit of the permission mask to zero. To do that, you should calculate the bitwise AND for the permission mask and the 1111 1101 1111 1111 number. The ninth bit of the number equals zero and all other bits are ones. Therefore, the bitwise AND changes the ninth bit of the permission mask only.

The OS operates masks whenever you access a file. This way, it checks your access rights.

Here is the last example of using bitwise operations. Until recently, software developers used bit shifts as an alternative to multiplication and division by a power of two. For example, the bit shift to the left by two bits corresponds to multiplication by 2^2 (i.e. four). You can check it with the following Bash command:

```
1 $ echo $((3 << 2))
2 12
```

The bit shift gives the same result as multiplication because “3 * 4” equals 12.

This trick reduces the number of processor clock cycles to perform multiplication and division. These optimizations are now unnecessary due to the development of compilers and processors. Compilers automatically select the fastest assembly instructions when generating code. Processors execute these instructions in parallel with [several threads](https://en.wikipedia.org/wiki/Hyper-threading)³³³. Today, software developers tend to write code that is easier to read and understand. They do not care about optimizations as they do it before. Multiplication and division operations are better for reading the code than bit shifts.

Cryptography and computer graphics algorithms use bit operations a lot.

³³³<https://en.wikipedia.org/wiki/Hyper-threading>

Logical Operations

The `[[` operator is inconvenient for comparing integers in the `if` statement. This operator uses two-letter abbreviations for expressing the relations between numbers. For example, the `-gt` abbreviation means greater. When you apply the `((` operator instead, you can use the usual comparison symbols there. These symbols are: `>`, `<` and `=`.

Here is an example. Suppose that you want to compare some variable with the number 5. The following `if` construction does that:

```
1 if ((var < 5))
2 then
3   echo "The variable is less than 5"
4 fi
```

The construction uses the `((` operator in the arithmetic evaluation form. You can replace it with the `let` built-in. Then it provides the same result:

```
1 if let "var < 5"
2 then
3   echo "The variable is less than 5"
4 fi
```

However, you should always prefer to use the `((` operator.

There is an important difference between arithmetic evaluation and expansion. According to the POSIX standard, any program or command returns the zero exit status when it succeeds. It returns the status between 1 and 255 when it fails. The shell interprets the exit status like this: zero means “true” and nonzero means “false”. If you apply this rule, the logical result of the arithmetic expansion is inverted. There is no such inversion for the arithmetic evaluation result.

Arithmetic evaluation is synonymous with the `let` built-in. Therefore, it follows the POSIX standard just like any other command. The shell executes the arithmetic expansion in the context of another command. Thus, its result depends on the interpreter’s implementation.

Suppose that you use the `((` operator in the arithmetic expansion form. Then Bash interprets its result this way: if the condition in the `((` operator equals “true”, it returns one. Otherwise, the operator returns zero. The C language deduces Boolean expressions in the same way.

An example will demonstrate the difference between the arithmetic expansion and evaluation. The following Bash command compares the `var` variable with the number 5:

```
((var < 5)) && echo "The variable is less than 5"
```

This command contains the arithmetic evaluation. Therefore, if the `var` variable is less than 5, the `((` operator succeeds. It returns the zero exit status according to the POSIX standard. Then `echo` prints the message.

When you use the operator `((` in the form of the arithmetic expansion, it gives you another result. The following command makes the same comparison:

```
echo "$((var < 5))"
```

When this condition is true, the `echo` command prints the number one. If you are familiar with the C language, you expect the same result.

You can use logical operations in the arithmetic evaluation form of the `((` operator. They work in the same way as the Bash logical operators.

Here is an example of how to apply a logical operation. The following `if` condition compares the `var` variable with the numbers 1 and 5:

```
1 if ((1 < var && var < 5))
2 then
3   echo "The variable is less than 5 but greater than 1"
4 fi
```

This condition is true when both expressions are true.

The logical OR works similarly:

```
1 if ((var < 1 || 5 < var))
2 then
3   echo "The variable is less than 1 or greater than 5"
4 fi
```

The condition is true if at least one of two expressions is true.

It rarely happens when you apply the logical NOT to some number. Instead, you use it to negate the value of some variable or expression. If you apply the logical NOT to a number, its output corresponds to the POSIX standard. In other words, zero means “true” and nonzero means “false”. Here is an example:

```
1 if ((! var))
2 then
3   echo "The variable equals true or zero"
4 fi
```

This condition is true if the `var` variable equals zero.

Increment and Decrement

The increment and decrement operations first appeared in the [programming language B³³⁴](#). Ken Thompson and Dennis Ritchie developed it in 1969 while working at Bell Labs. Dennis Ritchie moved these operations later to his new language called C. Bash copied them from C.

First, let's consider the assignment operations. It helps you to get how increment and decrement work. A regular assignment in arithmetic evaluation looks like this:

```
((var = 5))
```

This command assigns the number 5 to the `var` variable.

Bash allows you to combine an assignment with arithmetic or bitwise operation. The following command does addition and assignment at the same time:

```
((var += 5))
```

The command performs two actions:

1. It adds the number 5 to the current value of the `var` variable.
2. It writes the result back to the `var` variable.

All other assignment operations work the same way. First, they do a mathematical or bitwise operation. Second, they assign the result to the variable. Using assignments makes your code shorter and clearer.

Now we will consider the increment and decrement operations. They have two forms: postfix and prefix. You should write them in different ways. The `++` and `-` signs come after the variable name in the postfix form. They come before the variable name in the prefix form.

Here is an example of the prefix increment:

```
((++var))
```

This command provides the same result as the following assignment operation:

```
((var+=1))
```

The increment operation increases the variable's value by one. Decrement decreases it by one.

Why does it make sense to introduce special operations for adding and subtracting one? The Bash language has assignments `+=` and `-=` that you can use instead.

³³⁴[https://en.wikipedia.org/wiki/B_\(programming_language\)](https://en.wikipedia.org/wiki/B_(programming_language))

The most probable reason for introducing the increment and decrement is managing a [loop counter](#)³³⁵. This counter keeps the number of loop iterations. When you want to interrupt the loop, you check its counter in a condition. The result defines if you should interrupt the loop or not.

Increment and decrement make it easier to serve the loop counter. Besides that, modern processors perform these operations on the hardware level. Therefore, they work faster than addition and subtraction combined with the assignment.

What is the difference between prefix and postfix forms of increment? If the expression consists only of an increment operation, you get the same result for both forms.

For example, the following two commands increase the variable's value by one:

```
1 ((++var))
2 ((var++))
```

The difference between the increment forms appears when you assign the result to some variable. Here is an example:

```
1 var=1
2 ((result = ++var))
```

After executing these two commands, both variables `result` and `var` store the number 2. It happens because the prefix increment first adds one and then returns the result.

If you break the prefix increment into steps, you get the following commands:

```
1 var=1
2 ((var = var + 1))
3 ((result = var))
```

The postfix increment behaves differently. Let's change the increment's form in our example:

```
1 var=1
2 ((result = var++))
```

These commands write the number 1 to the `result` variable and the number 2 to the `var` variable. It happens because the postfix increment returns the current value of the variable first. Then it adds one to this value and writes the result back to the variable.

If you break the postfix increment into steps, you get the following commands:

³³⁵https://en.wikipedia.org/wiki/For_loop#Loop_counters


```
1 var=1
2 ((tmp = var))
3 ((var = var + 1))
4 ((result = tmp))
```

Note the order of steps in the postfix increment. First, it increments the `var` variable by one. Then it returns the previous value of `var`. Therefore, the increment needs the temporary variable `tmp` to store this previous value.

The postfix and prefix forms of decrement work similarly to increment. They decrease the variable by one.

Always use the prefix increment and decrement instead of the postfix form. First, the CPU performs them faster. The reason is it does not need to save the previous value of the variable in the registers. Second, it is easier to make an error using the postfix form. It happens because of the non-obvious order of assignments.

Ternary Operator

The ternary operator is also known as the conditional operator and **ternary if**. The first time, it appeared in the programming language [ALGOL](#)³³⁶. The operator turned out to be convenient and many programmers liked it. The languages of the next generation ([BCPL](#)³³⁷ and C) inherited the ternary if. This way, it comes to almost all modern languages: C++, C#, Java, Python, PHP, etc.

The ternary operator is a compact form of the `if` statement.

Here is an example. Suppose that your script has the following `if` statement:

```
1 if ((var < 10))
2 then
3   ((result = 0))
4 else
5   ((result = var))
6 fi
```

Here the `result` variable gets the zero value if `var` is less than 10. Otherwise, `result` gets the value of `var`.

You can get the same behavior using the ternary operator. It looks like this:

```
((result = var < 10 ? 0 : var))
```

³³⁶<https://en.wikipedia.org/wiki/ALGOL>

³³⁷<https://en.wikipedia.org/wiki/BCPL>

The ternary operator replaced six lines of the `if` statement with one line. This way, you got simpler and clearer code.

The ternary operator consists of a conditional expression and two actions. Its general form looks like this:

```
(( CONDITION ? ACTION_1 : ACTION_2 ))
```

If the `CONDITION` is true, Bash executes the `ACTION_1`. Otherwise, it executes the `ACTION_2`. This behavior matches the following `if` statement:

```
1 if CONDITION
2 then
3     ACTION_1
4 else
5     ACTION_2
6 fi
```

Unfortunately, Bash allows the ternary operator in arithmetic evaluation and expansion only. It means that the operator accepts arithmetic expressions as actions. You cannot call commands and utilities there, as you do it in the code blocks of the `if` statement. There is no such restriction in other programming languages.

Use the ternary operator as often as possible. It is considered a good practice. The operator makes your code compact and easy to read. The less code has less room for potential errors.

Loop Constructs

You have learned the conditional statements. They manage the **control flow**³³⁸ of a program. The control flow is the execution order of the program's commands.

The conditional statement chooses a branch of execution depending on the result of a Boolean expression. However, this statement is not enough in some cases. You need extra features to manage the control flow. The **loop construct**³³⁹ helps you to handle these cases.

The loop construct repeats the same block of commands multiple times. The single execution of this block is called the **loop iteration**. The loop checks its condition after each iteration. The check result defines if the next iteration should be executed.

³³⁸https://en.wikipedia.org/wiki/Control_flow

³³⁹https://en.wikipedia.org/wiki/Control_flow#Loops

Repetition of Commands

Why does somebody need to repeat the same block of commands in his program? Several examples will help us to answer this question.

You are already familiar with the `find` utility. It looks for files and directories on the disk drive. If you add the `-exec` option to the `find` call, you can specify some action. The utility performs this action for each found object.

For example, the following command deletes all PDF documents in the `~/Documents` directory:

```
find ~/Documents -name "*.pdf" -exec rm {} \;
```

Here `find` calls the `rm` utility several times. It passes the next found file on each call. It means that the `find` utility executes the loop implicitly. The loop ends when `find` finishes the processing of all found files.

The `du` utility is another example of the repetition of commands. The utility estimates the amount of occupied disk space. It has one optional parameter. The parameter sets the path to start the estimation.

Here is an example of the `du` call:

```
du ~/Documents
```

Here the utility traverses all `~/Documents` subdirectories recursively. It adds the size of each found file to the final result. This way, incrementing the result repeats several times.

The `du` utility performs a loop implicitly. The loop traverses over all files and subdirectories. It does the same actions on each iteration. The only difference between iterations is a file system object to check.

You can meet the repetition of operations in regular mathematical calculations. A canonical example here is the calculation of **factorial**³⁴⁰. The factorial of the number `N` is a multiplication of natural numbers from 1 to `N` inclusive.

Here is an example of calculating the factorial of number 4:

```
4! = 1 * 2 * 3 * 4 = 24
```

You can calculate the factorial easily when using the loop. The loop should pass through the integers from 1 to `N` in sequence. You should multiply the final result by each passed integer. This way, you repeat the multiplication operation several times.

Here is the last example of action repetition in a computer system. Repetition is an effective approach to manage some events.

³⁴⁰<https://en.wikipedia.org/wiki/Factorial>

Suppose that you write a program. It downloads files to your computer from the Internet. First, the program establishes a connection to a server. If the server doesn't respond, the program has two options to do. The first one is to terminate with a non-zero exit status. The second option is to wait for the server response. This behavior of the program is preferable. There are many reasons why the packets from the server can delay. It can be an overload of the network, for example. Waiting for a couple of seconds is enough to get the packet. Then your program can continue to work.

Now the question arises: how can you wait for the event to occur in the program? The easiest way to get it is using a loop operator. The operator's condition should check if the event occurs. When it happens, the operator stops.

Let's come back to our example. The loop should stop when the program receives a response from the server. While it does not happen, the loop continues. You do not need any actions on each iteration. Instead, you leave the loop body empty. This technique is called **busy waiting**³⁴¹.

Busy waiting does nothing but can consume a lot of CPU time. This is a reason why you should optimize it when possible. Add the command, which stops the program for a short time, to the loop body. It gives OS a chance to execute another task while your program is waiting.

We have considered examples when the program repeats the same action several times. Let's write down the tasks that such repetition solves:

1. Process multiple entities monotonously. The `find` utility processes the search results this way.
2. Accumulate intermediate data for calculating the final result. The `du` utility does it for collecting statistics.
3. Mathematical calculations. You can calculate factorial using the loop.
4. Wait for some event to happen. You can wait for the server response in the busy waiting loop.

This list is far from being complete. It demonstrates just the most common programming tasks that require the loop operator.

While Statement

Bash provides two loop operators: `while` and `for`. We will start with the `while` statement because it is more straightforward than `for`.

The `while` syntax resembles the `if` statement. If you write `while` in the general form, it looks this way:

³⁴¹https://en.wikipedia.org/wiki/Busy_waiting

```
1 while CONDITION
2 do
3     ACTION
4 done
```

You can write the `while` statement in one line:

```
while CONDITION; do ACTION; done
```

Both `CONDITION` and `ACTION` can be a single command or block of commands. The `ACTION` is called the **loop body**.

When Bash executes the `while` loop, it checks the `CONDITION` first. If a command of the `CONDITION` returns the zero exit status, it means “true”. Bash executes the `ACTION` in the loop body in this case. Then it rechecks the `CONDITION`. If it still equals “true”, the `ACTION` is performed again. The loop execution stops when the `CONDITION` becomes “false”.

Use the `while` loop when you do not know the number of iterations beforehand. A good example of this case is busy waiting for some event.

Suppose that you write a script that checks if some web server is available. The simplest check looks this way:

1. Send a request to the server.
2. Receive the response.
3. If there is no response, the server is unavailable.

When the script receives the response from the server, it should print a message and stop.

You can call the `ping`³⁴² utility to send a request to the server. The utility uses the `ICMP`³⁴³ protocol.

The protocol is an agreement for the format of messages between the computers of the network. The `ICMP` protocol describes the error messages and packets with operational information. For example, you need them to check if some computer is available.

When calling the `ping` utility, you should specify an `IP address`³⁴⁴ or `URL`³⁴⁵ of the target **host**. A host is a computer or device connected to the network.

Here is an example of the `ping` call:

```
ping google.com
```

We have specified the Google server as the target host. The utility sends `ICMP` messages there. The server replies to them. The utility output looks like this:

³⁴²[https://en.wikipedia.org/wiki/Ping_\(networking_utility\)](https://en.wikipedia.org/wiki/Ping_(networking_utility))

³⁴³https://en.wikipedia.org/wiki/Internet_Control_Message_Protocol

³⁴⁴https://en.wikipedia.org/wiki/IP_address

³⁴⁵<https://en.wikipedia.org/wiki/URL>

```
1 PING google.com (172.217.21.238) 56(84) bytes of data.
2 64 bytes from fra16s13-in-f14.1e100.net (172.217.21.238): icmp_seq=1 ttl=51 time=17.\
3 8 ms
4 64 bytes from fra16s13-in-f14.1e100.net (172.217.21.238): icmp_seq=2 ttl=51 time=18.\
5 5 ms
```

You see information about each sent and received ICMP message. The “time” field means the delay between sending the request and receiving the server response.

The utility runs in the infinite loop by default. You can stop it by pressing Ctrl+C.

You do not need to send several requests to check if some server is available. It is sufficient to send a single ICMP message instead. The `-c` option of the `ping` utility specifies the number of messages to send. Here is an example of how to use it:

```
ping -c 1 google.com
```

If the `google.com` server is available, the utility returns the zero exit status. Otherwise, it returns a non-zero value.

The `ping` utility expects the server response until you do not interrupt it. The `-W` option limits this waiting time. You can specify one second to wait this way:

```
ping -c 1 -W 1 google.com
```

Now you have the condition for the `while` statement. There is time to write this statement:

```
1 while ! ping -c 1 -W 1 google.com &> /dev/null
2 do
3     sleep 1
4 done
```

The output of the `ping` utility does not matter in our case. Therefore, you can redirect it to the `/dev/null` file.

The exit status of the `ping` utility is inverted in our `while` condition. Therefore, Bash executes the loop body as long as the utility returns a non-zero exit status. It means that the loop continues as long as the server stays unavailable.

The loop body contains the `sleep` utility call only. It stops the script execution for the specified number of seconds. The stop lasts for one second in our case.



You can specify a suffix for a parameter when calling the `sleep` utility. The suffix defines the units of time. The suffix “s” matches seconds. It is “m” for minutes, “h” for hours and “d” for days.

Listing 3-18 shows the complete script for checking server availability.

Listing 3-18. Script for checking server availability

```
1 #!/bin/bash
2
3 while ! ping -c 1 -W 1 google.com &> /dev/null
4 do
5     sleep 1
6 done
7
8 echo "The google.com server is available"
```

The `while` statement has an alternative form called `until`. It executes the `ACTION` until the `CONDITION` stays “false”. It means that the loop continues as long as the `CONDITION` returns a non-zero exit status. Use the `until` statement when you need to invert the condition of the `while` loop.

The general form of the `until` statement looks this way:

```
1 until CONDITION
2 do
3     ACTION
4 done
```

You can write it in one line, the same way as you do it for `while`:

```
until CONDITION; do ACTION; done
```

Let’s replace the `while` statement with `until` in Listing 3-18. You should remove the negation of the ping result for that. Listing 3-19 shows the changed script.

Listing 3-19. Script for checking server availability

```
1 #!/bin/bash
2
3 until ping -c 1 -W 1 google.com &> /dev/null
4 do
5     sleep 1
6 done
7
8 echo "The google.com server is available"
```

The scripts in Listing 3-18 and Listing 3-19 behave the same.

Choose the `while` or `until` statement, depending on the loop condition. Your goal is to avoid negations there. Negations make the code harder to read.

Infinite Loop

The `while` statement fits well when you need to implement an **infinite loop**³⁴⁶. This kind of loop continues as long as the program is running.

You can meet infinite loops in system software. They run the whole time while a computer stays powered on. An example is the microcontroller firmware that checks some sensors cyclically. It happens in the infinite loop. Also, such loops are used in computer games, antiviruses, monitors of computer resources, etc.

The `while` loop becomes infinite if its condition always stays true. The easiest way to make such a condition is to call the `true` Bash built-in. Here is an example for doing that:

```
1 while true
2 do
3   sleep 1
4 done
```

The `true` built-in always returns the “true” value. It means that it returns zero exit status. There is the symmetric command called `false`. It always returns exit status one that matches the “false” value.



Words “true” and “false” are **literals**³⁴⁷ in most programming languages. They represent the corresponding Boolean values. Literals are reserved words for representing fixed values.

You can replace the `true` built-in in the `while` condition with a colon. Then you will get the following statement:

```
1 while :
2 do
3   sleep 1
4 done
```

The colon is synonymous with the `true` command. This synonymous solves the **compatibility task**³⁴⁸ with the Bourne shell. This shell does not have `true` and `false` built-ins. Bourne shell scripts use a colon instead, and Bash should support it.

The POSIX standard includes all three keywords: `colon`, `true`, and `false`. However, you should avoid using a colon in your scripts. It is a deprecated syntax that makes your code harder to understand.

Here is an example of an infinite loop. Suppose that you need a script that displays statistics of disk space usage. The `df` utility can help you in this case. It provides the following output when called without parameters:

³⁴⁶https://en.wikipedia.org/wiki/Infinite_loop

³⁴⁷[https://en.wikipedia.org/wiki/Literal_\(computer_programming\)](https://en.wikipedia.org/wiki/Literal_(computer_programming))

³⁴⁸<https://stackoverflow.com/questions/3224878/what-is-the-purpose-of-the-colon-gnu-bash-built-in>


```

1 $ df
2 Filesystem      1K-blocks      Used Available Use% Mounted on
3 C:/msys64      41940988  24666880  17274108  59% /
4 Z:              195059116 110151748  84907368  57% /z

```

The utility shows “Used” and “Available” disk space in bytes. You can improve this output by adding the `-h` option to the utility call. Then `df` shows kilobytes, megabytes, gigabytes and terabytes instead of bytes. Another option that you can apply is `-T`. It shows the file system type for each disk. You will get the following output after all improvements:

```

1 $ df -hT
2 Filesystem      Type  Size  Used Avail Use% Mounted on
3 C:/msys64      ntfs  40G   24G   17G  59% /
4 Z:              hgfs  187G  106G   81G  57% /z

```

If you need to get information about all mount points, add the `-a` option.

Now you should write the infinite loop. It calls the `df` utility on each iteration. This way, you will get a simple script to monitor the file system. Listing 3-20 shows how it looks like.

Listing 3-20. The script to monitor the file system

```

1 #!/bin/bash
2
3 while true
4 do
5     clear
6     df -hT
7     sleep 2
8 done

```

The first action of the loop body is the `clear` utility call. It removes all text in the terminal window. Thanks to this step, the terminal shows the output of your script only.

When working with Bash, you often face the task of executing a command in a cycle. The `watch` utility does it. The utility is a part of the `procs` package. If you need it, the following command installs it to the MSYS2 environment:

```
pacman -S procs
```

Now you can replace the script from Listing 3-20 with a single command this way:

```
watch -n 2 "df -hT"
```

The `-n` option of the `watch` utility specifies the interval between command calls. The command to execute follows all options of `watch`.

The `-d` utility option highlights the difference of the command output at the current and past iterations. This way, it is easier to keep track of occurred changes.

Reading a Standard Input Stream

The `while` loop fits well for handling an input stream. Here is an example of such a task. Suppose that you need a script that reads a text file. It should make an associative array from the file content.

Listing 3-10 shows the script for managing the list of contacts. The script stores contacts in the format of the Bash array declaration. It makes adding a new person to the list inconvenient. The user must know the Bash syntax. Otherwise, he can make a mistake when initializing an array element. It will break the script.

There is a solution to the problem of editing the contacts list. You can move the list into a separate text file. Then, the script would read it at startup. This way, you separate data and code. It is a well-known good practice in software development.

Listing 3-21 shows a possible format of the file with contacts.

Listing 3-21. The file with contacts `contacts.txt`

```
1 Alice=alice@gmail.com
2 Bob=(697) 955-5984
3 Eve=(245) 317-0117
4 Mallory=mallory@hotmail.com
```

Let's write the script that reads this file. It is convenient to read the list of contacts directly into the associative array. This way, you will keep the searching mechanism over the list as effective as before.

When reading the file, you should handle its lines in the same manner. It means that you will repeat the same action several times. Therefore, you need a loop statement.

At the beginning of the loop, you don't know the file size. Thus, you do not know the number of iterations to do. The `while` statement fits this case perfectly.

Why is the number of iterations unknown in advance? It happens because the script reads the file line by line. It cannot count the lines before it reads them all. There is an option to make two loops. Then the first one counts the lines. The second loop processes them. However, this solution works slower and is less efficient.

You can call the `read` Bash built-in for reading lines of the file. The command receives a string from the standard input stream. Then it writes the string into the specified variable. You can pass the variable name as the parameter. Here is an example of calling `read`:

```
read var
```

Run this command. Then type the string and press Enter. The `read` command writes your string into the `var` variable. You can call `read` without parameters. It writes the string into the reserved variable `REPLY` in this case.

When `read` receives the string, it removes backslashes `\` there. They escape special characters. Therefore, the `read` command considers the backslashes unnecessary. The `-r` option disables this feature. Use it always to prevent losing characters of the input string.

You can pass several variable names to the `read` command. Then it divides the input text into parts. The command uses the delimiters from the reserved variable `IFS` in this case. Default delimiters are spaces, tabs and line breaks.

Here is an example of how the `read` built-in deals with several variables. Suppose that you want to store the input string into two variables. They are called `path` and `file`. The following command reads them:

```
read -r path file
```

Suppose that you have typed the following string for this command:

```
~/Documents report.txt
```

Then the `read` command writes the `~/Documents` path into the `path` variable. The filename `report.txt` comes into the `file` variable.

If the path or filename contains spaces, the error occurs. For example, you can type the following string:

```
~/My Documents report.txt
```

Then the `read` built-in writes the `~/My` string into the `path` variable. The `file` variable gets the rest part of the input: `Documents report.txt`. This is the wrong result. Keep in mind this behavior when using the `read` command.

There is a solution to the problem of splitting the input string. You can solve it by redefining the `IFS` variable. For example, the following declaration specifies a comma as the only possible delimiter:

```
IFS=$',' read -r path file
```

This declaration uses the Bash-specific [type of quotes](#)³⁴⁹ `$'...'`. Bash does not perform any expansions inside them. At the same time, you can place the following control sequences there: `\n` (new line), `\\` (escaped backslash), `\t` (tabulation) and `\xnn` (bytes in hexadecimal).

The `IFS` redeclaration allows you to process the following input string properly:

³⁴⁹<http://mywiki.woledge.org/Quotes>

```
1 ~/My Documents,report.txt
```

Here the comma separates the path and filename. Therefore, the `read` command writes the `~/My Documents` string into the `path` variable. The `report.txt` string comes into the `file` variable.

The `read` built-in receives data from the standard input stream. It means that you can redirect the file contents there.

Here is an example to read the first line of the `contacts.txt` file from Listing 3-21. The following command does it:

```
read -r contact < contacts.txt
```

This command writes the “Alice=alice@gmail.com” string to the `contact` variable.

You can write the name and contact information to two different variables. You need to define the equal sign as a delimiter to do that. Then you will get the following `read` call:

```
IFS=$'= ' read -r name contact < contacts.txt
```

Now the `name` variable gets the “Alice” name. The e-mail address comes to the `contact` variable.

Let’s try the following `while` loop for reading the entire `contacts.txt` file:

```
1 while IFS=$'= ' read -r name contact < "contacts.txt"
2 do
3   echo "$name = $contact"
4 done
```

Unfortunately, this approach does not work. You get the infinite loop accidentally. It happens because the `read` command always reads only the first line of the file. Then it returns the zero exit status. The zero status leads to another execution of the loop body. It happens over and over again.

You should force the `while` loop to pass through all lines of the file. The following form of the loop does it:

```
1 while CONDITION
2 do
3   ACTION
4 done < FILE
```

This form of the loop can handle keyboard input too. You need to specify the `/dev/tty` input file for doing that. Then the loop will read keystrokes until you press Ctrl+D.

Here is the corrected `while` loop that reads the entire `contacts.txt` file:

```

1 while IFS='=' read -r name contact
2 do
3     echo "$name = $contact"
4 done < "contacts.txt"

```

This loop prints all lines of the contacts file.

There is the last step left to finish your script. You should write the name and contact variables to the associative array on each iteration. The name variable is the key and contact is the value.

Listing 3-22 shows the final version of the script for reading the contacts from the file.

Listing 3-22. The script for managing the contacts

```

1 #!/bin/bash
2
3 declare -A array
4
5 while IFS='=' read -r name contact
6 do
7     array[$name]=$contact
8 done < "contacts.txt"
9
10 echo "${array["$1"]}"

```

This script behaves the same way as the one in Listing 3-10.

For Statement

There is another loop statement in Bash called `for`. You should use it when you know the number of iterations in advance.

The `for` statement has two forms. The first one processes words in a string sequentially. The second form applies an arithmetic expression in the loop condition.

The First Form of For

Let's start with the first form of the `for` statement. It looks this way in the general form:

```

1 for VARIABLE in STRING
2 do
3     ACTION
4 done

```

You can write the same construction in a single line like this:

```
for VARIABLE in STRING; do ACTION; done
```

The ACTION of the `for` statement is a single command or a block of commands. It is the same thing as the one in the `while` statement.

Bash performs all expansions in the `for` condition before starting the first iteration of the loop. What does it mean? Suppose you specified the command instead of the STRING. Then Bash executes this command and replaces it with its output. Also, you can specify a pattern instead of the STRING. Then Bash expands it before starting the loop.

Bash splits the STRING into words when there are no commands or patterns left in the `for` condition. It takes the separators for splitting from the IFS variable.

Then Bash executes the first iteration of the loop. The first word of the STRING is available via the VARIABLE inside the loop body on the first iteration. Then Bash writes the second word of the STRING to the VARIABLE and starts the second iteration. It happens again and again until you handle all words of the STRING.

Here is an example of the `for` loop. Suppose that you need a script that prints words of a string one by one. The script receives the string via the first parameter. Listing 3-23 shows how its code looks like.

Listing 3-23. The script for printing words of a string

```
1 #!/bin/bash
2
3 for word in $1
4 do
5     echo "$word"
6 done
```

Here you should not enclose the position parameter `$1` in quotes. Quotes prevent word splitting. Without word splitting, Bash passes the whole string to the first iteration of the `for` loop. Then the loop finishes. You do not want this behavior. The script should process each word of the string separately.

When you call the script, you should enclose the input string in the double quotes. Then the whole string comes into the `$1` parameter. Here is an example of calling the script:

```
./for-string.sh "this is a string"
```

There is a way to get rid of the double quotes when calling the script. Replace the `$1` parameter in the `for` condition with `$@`. Then the loop statement becomes like this:

```
1 for word in $@
2 do
3     echo "$word"
4 done
```

Now both following script calls work properly:

```
1 ./for-string.sh this is a string
2 ./for-string.sh "this is a string"
```

The for loop condition has a short form. Use it when you need to handle all input parameters of the script. This short form looks this way:

```
1 for word
2 do
3     echo "$word"
4 done
```

It does the same as our previous script for processing an unquoted string. The only difference is dropping the “in \$@” part in the for condition. It did not change the loop behavior.

Let’s make the task a bit more complicated. Suppose the script receives a list of paths in input parameters. They are separated by commas. The paths may contain spaces. Then you should redefine the IFS variable to process such input correctly.

Listing 3-24 shows the for loop that prints the list of paths.

Listing 3-24. The script for printing the list of paths

```
1 #!/bin/bash
2
3 IFS=$', '
4 for path in $1
5 do
6     echo "$path"
7 done
```

You have specified only one allowable delimiter in the IFS variable. This delimiter is the comma. Therefore, the for loop ignores spaces when splitting the input string.

You can call the script this way:

```
./for-path.sh "~/My Documents/file1.pdf,~/My Documents/report2.txt"
```

There are the mandatory double quotes for the input string here. You cannot replace the \$1 parameter with @\$ in the for condition and omit quotes. This will lead to an error. The error happens because Bash does word splitting when calling the script. This word splitting applies spaces as delimiters. It occurs before the redeclaration of the IFS variable. Thus, Bash ignores your change of the variable in this case.

If some path contains a comma, it leads to an error.

The for loop can pass through the elements of an indexed array. It works the same way as processing words in a string. Listing 3-25 shows an example of doing that.

Listing 3-25. The script for printing all elements of the array

```
1 #!/bin/bash
2
3 array=(Alice Bob Eve Mallory)
4
5 for element in "${array[@]}"
6 do
7     echo "$element"
8 done
```

Suppose that you need the first three elements of an array. In this case, you should expand only the elements you need in the loop condition. Listing 3-26 shows how to do that.

Listing 3-26. The script for printing the first three elements of the array

```
1 #!/bin/bash
2
3 array=(Alice Bob Eve Mallory)
4
5 for element in "${array[@]:0:2}"
6 do
7     echo "$element"
8 done
```

There is another option to handle the array. You can iterate over the indexes instead of the elements. These are the steps for doing that:

1. Write the string with indexes of the elements you need. They should be separated by spaces.
2. Put the string into the for condition.
3. The loop gives you an index on each iteration.

Here is an example of the loop condition:


```
1 array=(Alice Bob Eve Mallory)
2
3 for i in 0 1 2
4 do
5     echo "${array[i]}"
6 done
```

This loop passes only through elements with indexes 0, 1 and 2.

You can apply the brace expansion to specify the indexes list. Here is an example:

```
1 array=(Alice Bob Eve Mallory)
2
3 for i in {0..2}
4 do
5     echo "${array[i]}"
6 done
```

This loop prints the first three elements of the array too.

Do not iterate over the element indexes when processing arrays with gaps. You should expand the array elements in the loop condition instead. Listing 3-25 and Listing 3-26 show how to do that.

Files Processing

The `for` loop works well when you need to process a list of files. The only point here is to compose the loop condition correctly. There are several common mistakes when writing this condition. Let's consider them by examples.

The first example is a script that reads the current directory and prints the types of all files there. You can call the `file` utility to get this information for each file.

When composing the `for` loop condition, the most common mistake is the neglect of patterns (globbing). Users often call the `ls` or `find` utility to get the `STRING`. It happens this way:

```
1 for filename in $(ls)
2 for filename in $(find . -type f)
```

Both these `for` conditions are wrong. They lead to the following problems:

1. Word splitting breaks the names of files and directories with spaces.
2. If the filename contains an asterisk, Bash performs globbing before starting the loop. Then it writes the globbing result to the `filename` variable. This way, you lose the actual filename.

3. The output of the `ls` utility depends on the regional settings. Therefore, you can get question marks instead of the national alphabet characters in filenames. Then the `for` loop cannot process these files.

Always use patterns in the `for` condition when you need to enumerate filenames. It is the only correct solution for this task.

You should write the following `for` loop condition for our script:

```
for filename in *
```

Listing 3-27 shows the complete script.

Listing 3-27. The script for printing the file types

```
1 #!/bin/bash
2
3 for filename in *
4 do
5     file "$filename"
6 done
```

Do not forget to use the double quotes when accessing the `filename` variable. They prevent word splitting of filenames with spaces.

You can use a pattern in the `for` loop condition if you want to process files from a specific directory. Here is an example for doing that:

```
for filename in /usr/share/doc/bash/*
```

A pattern can filter out files with a specific extension or name. It looks this way:

```
for filename in ~/Documents/*.pdf
```

There is a new feature of patterns in Bash version 4. You can pass through directories recursively. Here is an example:

```
1 shopt -s globstar
2
3 for filename in **
```

This feature is disabled by default. You can activate it by enabling the `globstar` Bash option with the `shopt` command.

When Bash meets the `**` pattern, it inserts a list of all subdirectories and their files starting from the current directory. You can combine this mechanism with a regular pattern.

For example, you want to process all files with the PDF extension from the home directory. The following `for` loop condition does that:

```
1 shopt -s globstar
2
3 for filename in ~/**/*.pdf
```

There is another common mistake when using the `for` loop. Sometimes you just do not need it. For example, you can replace the script in Listing 3-27 with the following `find` call:

```
find . -maxdepth 1 -exec file {} \;
```

This command is more efficient than the `for` loop. It is compact and works faster because of fewer operations to do.

When should you use the `for` loop instead of the `find` utility? Use `find` when one short command is enough to process found files. If you need a conditional statement or block of commands for that, use the `for` loop.

There are cases when patterns are not enough for the `for` loop condition. For example, you need a complex search with checking file types. Use the `while` loop in this case.

Let's replace the `for` loop with `while` in Listing 3-27. Then you can replace the pattern with the `find` call. When doing that, you should apply the `-print0` option of `find`. This way, you avoid issues caused by word splitting. Listing 3-28 shows how to combine the `find` utility with the `while` loop properly.

Listing 3-28. The script for printing the file types

```
1 #!/bin/bash
2
3 while IFS= read -r -d '' filename
4 do
5     file "$filename"
6 done <<(find . -maxdepth 1 -print0)
```

There are several tricky solutions in this script. Let's take a closer look at them. The first question is, why does the `IFS` variable get an empty value? If you keep it unchanged, Bash splits the `find` output by default delimiters (spaces, tabs and line breaks). It can break filenames with these characters.

The second solution is to apply the `-d` option of the `read` command. This option defines a delimiter character for splitting the input text. When using it, the `filename` variable gets the part of the string that comes before the next delimiter.

The `-d` option specifies the empty delimiter in our case. It means the NULL character. You can also specify it explicitly this way:

```
while IFS= read -r -d $'\0' filename
```

Thanks to the `-d` option, the `read` command handles the `find` output correctly. There is the `-print0` option in the utility call. It means that `find` separates found files by a NULL character. This way, you reconcile the `read` input format and the `find` output.

Note that you cannot specify the NULL character as a delimiter using the `IFS` variable. In other words, the following solution does not work:

```
while IFS=$'\0' read -r filename
```

The problem comes from the peculiarity when [interpreting the IFS variable](#)³⁵⁰. If the variable is empty, Bash does not do word splitting at all. When you assign the NULL character to the variable, it means an empty value for Bash.

There is the last tricky solution in Listing 3-28. The process substitution helps us to pass the `find` output to the `while` loop. Why did we not use the command substitution instead? It can look this way:

```
1 while IFS= read -r -d '' filename
2 do
3   file "$filename"
4 done < $(find . -maxdepth 1 -print0)
```

Unfortunately, this redirection does not work. The `<` operator couples the input stream and the specified file descriptor. When you apply the command substitution, there is no file descriptor. In this case, Bash calls the `find` utility and inserts its output to the command instead of `$(...)`. When you use the process substitution, Bash writes the `find` output to a temporary file. This file has a descriptor. Therefore, the stream redirection works fine.

The process substitution has only one issue. It is not part of the POSIX standard. If you should follow the standard, use a pipeline instead. Listing 3-29 demonstrates how to do that.

Listing 3-29. The script for printing the file types

```
1 #!/bin/bash
2
3 find . -maxdepth 1 -print0 |
4 while IFS= read -r -d '' filename
5 do
6   file "$filename"
7 done
```

Combine the `while` loop and `find` utility only when you meet both following cases:

³⁵⁰<https://mywiki.woledge.org/IFS>

1. You need a conditional statement or code block to process files.
2. You need a complex condition for searching files.

When combining `while` and `find`, always use the NULL character as a delimiter. This way, you avoid the word splitting problems.

The Second Form of For

The second form of the `for` statement allows you to apply an arithmetic expression as a condition. Let's consider cases when you need it.

Suppose that you write a script for calculating a factorial. The solution for this task depends on the way you enter the data. The first option is you have a predefined integer. Then you can apply the first form of the `for` loop. Listing 3-30 shows this solution.

Listing 3-30. The script for calculating the factorial for integer 5

```
1  #!/bin/bash
2
3  result=1
4
5  for i in {1..5}
6  do
7      ((result *= $i))
8  done
9
10 echo "The factorial of 5 is $result"
```

The second option is the script gets an integer via the parameter. You can try to keep the first form of `for` and handle the `$1` parameter this way:

```
for i in {1..$1}
```

You can expect that Bash does brace expansion here. However, it does not happen.

According to Table 3-2, the brace expansion happens before the parameter expansion. Thus, the loop condition gets the `"{1...$1}"` string instead of `"1 2 3 4 5"`. Bash does not recognize the brace expansion here because the upper bound of the range is not an integer. Then Bash writes the `"{1...$1}"` string to the `i` variable. Therefore, the `((` operator in the loop body fails.

The `seq` utility can solve our problem. It generates a sequence of integers or fractions.

Table 3-21 shows options to call the `seq` utility.

Table 3-21. The options to call the seq utility

Number of parameters	Description	Example	Result
1	The parameter defines the last number in the generated sequence. The sequence starts with one.	seq 5	1 2 3 4 5
2	The parameters are the first and last numbers of the generated sequence.	seq -3 3	-2 -1 0 1 2
3	The parameters are the first number, step and last numbers of the generated sequence.	seq 1 2 5	1 3 5

The seq utility splits the generated integers by line breaks. You can specify another delimiter using the -s option. If you skip this option, you can process the seq output anyway. It happens because the IFS variable contains a line break by default. Therefore, Bash performs word splitting for the seq output properly.

The “Result” column of Table 3-21 should have line breaks instead of spaces between generated integers. The spaces are used there for convenience.

Let’s apply the seq utility and adapt the script for calculating a factorial for any integer. Listing 3-31 shows the result.

Listing 3-31. The script for calculating a factorial

```

1  #!/bin/bash
2
3  result=1
4
5  for i in $(seq $1)
6  do
7      ((result *= $i))
8  done
9
10 echo "The factorial of $1 is $result"

```

This solution works properly. However, it is ineffective. The performance overhead comes because of calling the seq utility. It costs the same time as launching an application (for example, Windows Calculator). The OS kernel performs several complicated operations whenever Bash creates a new process. They take significant time from the processor’s point of view. Therefore, you should apply Bash built-ins whenever possible.

You need the second form of the for loop to calculate a factorial effectively. This form looks like this in general:

```
1 for (( EXPRESSION_1; EXPRESSION_2; EXPRESSION_3 ))
2 do
3     ACTION
4 done
```

You can write this loop in one line this way:

```
for (( EXPRESSION_1; EXPRESSION_2; EXPRESSION_3 )); do ACTION; done
```

Here is an algorithm that Bash follows when executing the `for` loop with an arithmetic condition:

1. Calculate the `EXPRESSION_1` once before the first loop iteration.
2. Execute the loop body while the `EXPRESSION_2` remains true. When it becomes false, the loop stops.
3. Calculate the `EXPRESSION_3` at the end of each iteration.

Let's change the `for` condition of Listing 3-31 with the arithmetic expression. Listing 3-32 shows the result.

Listing 3-32. The script for calculating a factorial

```
1 #!/bin/bash
2
3 result=1
4
5 for (( i = 1; i <= $1; ++i ))
6 do
7     ((result *= i))
8 done
9
10 echo "The factorial of $1 is $result"
```

The new script works faster. It uses Bash built-ins only. There is no need to create new processes anymore.

The `for` statement in the new script follows this algorithm:

1. Declare the `i` variable and assign it the number 1 before the first iteration of the loop. This variable is a loop counter.
2. Compare the loop counter with the input parameter `$1`.
3. If the counter is smaller than the `$1` parameter, do the loop iteration.
4. If the counter is greater than the parameter, stop the loop.

5. Calculate the arithmetic expression “`result *= i`” in the loop body. It multiplies the `result` variable by `i`.
6. When the loop iteration is done, calculate the “`++i`” expression of the `for` condition. It increments the `i` variable by one.
7. Go to the 2nd step of the algorithm.



In the general case, you do not need the dollar sign for variable names in the `((` operator and `let` command. However, it is necessary for code in Listing 3-32. Without the dollar sign, Bash confuses the `$1` parameter and the literal `1`.

We use the prefix increment form in the loop. The reason is it works faster than the postfix form.

Use the second form of the `for` whenever you should calculate the loop counter. There are no other effective solutions in this case.

Controlling the Loop Execution

The loop condition dictates when it should run and stop. Two Bash built-ins can change this behavior. Using them, you can interrupt the loop or skip its iteration. Let’s consider these built-ins in detail.

break

The `break` built-in stops the loop immediately. It is useful for handling an error and finishing an infinite loop.

Here is an example of using `break`. Suppose you write the script that searches the specific array element by its value. You apply the loop to traverse the array. When you find the element, there is no reason to continue the loop. You can finish it immediately with the `break` command. Listing 3-33 shows how to do it.

Listing 3-33. The script for searching an array element

```
1  #!/bin/bash
2
3  array=(Alice Bob Eve Mallory)
4  is_found="0"
5
6  for element in "${array[@]}"
7  do
8      if [[ "$element" == "$1" ]]
9      then
10         is_found="1"
11         break
```



```
12  fi
13  done
14
15  if [[ "$is_found" -ne "0" ]]
16  then
17    echo "The array contains the $1 element"
18  else
19    echo "The array does not contain the $1 element"
20  fi
```

The script receives one parameter on input. It contains the element's value that you are looking for.

The `is_found` variable stores the search result. The `if` statement in the loop body checks the array elements. If some element matches the `$1` parameter, the `is_found` variable gets the value 1. Then the `break` command interrupts the loop.

There is the `if` statement after the loop. It checks the `is_found` variable. Then the `echo` command prints the message with the search result.

Using the `break` built-in, you can extract some commands from the loop body and place them after it. This is a good practice to keep the loop body as short as possible. It makes your code easier to read and understand.

Please have a look at Listing 3-33 again. You can print the search result right in the loop body. Then you do not need the `is_found` variable at all. On the other hand, the processing of the found element can be complex. If it happens, it is better to take the code out of the loop body.

Sometimes it does not make sense to continue the script when interrupting the loop. Call the `exit` Bash built-in instead of `break` in this case.

For example, your script detects an error when processing the input data in the loop body. Then printing a message and calling the `exit` command is a good decision to handle this case.

The `exit` command makes your code cleaner when you handle the loop result in its body. Just call `exit` when you are done.

Let's replace the `break` command with `exit` in the code of Listing 3-33. Listing 3-34 shows the result.

Listing 3-34. The script for searching an array element

```
1  #!/bin/bash
2
3  array=(Alice Bob Eve Mallory)
4
5  for element in "${array[@]}"
6  do
7    if [[ "$element" == "$1" ]]
8    then
```

```
9     echo "The array contains the $1 element"
10    exit 0
11    fi
12 done
13
14 echo "The array does not contain the $1 element"
```

Using the `exit` command, you handle the search result in the loop body. This solution made the code shorter and simpler. However, you can get the opposite effect if the result processing requires a block of commands.

The scripts of Listing 3-33 and Listing 3-34 give the same result.

continue

The `continue` Bash built-in skips the current loop iteration. The loop does not stop in this case. It starts the next iteration instead.

Here is an example of using `continue`. Suppose you calculate the sum of positive integers of some array. You should distinguish the signs of the integers for doing that. The `if` statement can solve this task. If the sign is positive, you add the integer to the result. Listing 3-35 shows the script that does it.

Listing 3-35. The script for calculating the sum of positive integers of the array

```
1  #!/bin/bash
2
3  array=(1 25 -5 4 -9 3)
4  sum=0
5
6  for element in "${array[@]}"
7  do
8      if (( 0 < element ))
9      then
10         ((sum += element))
11     fi
12 done
13
14 echo "The sum of the positive numbers is $sum"
```

If the `element` variable is greater than zero, you add it to the result `sum`.

Let's apply the `continue` command to get the same behavior. Listing 3-36 shows the new version of the script.

Listing 3-36. The script for calculating the sum of positive integers of the array

```
1  #!/bin/bash
2
3  array=(1 25 -5 4 -9 3)
4  sum=0
5
6  for element in "${array[@]}"
7  do
8      if (( element < 0 ))
9      then
10         continue
11      fi
12
13      ((sum += element))
14  done
15
16  echo "The sum of the positive numbers is $sum"
```

You need to invert the condition of the `if` statement. Now it is “true” for negative numbers. Bash calls the `continue` command in this case. The command interrupts the current loop iteration. It means that all further operations of the loop body are ignored. Then the next iteration starts and handles the next array element.

Using the `continue` built-in, you apply the early return pattern in the context of the loop.

The `continue` command is convenient for handling errors. It is also helpful for cases when it does not make sense to execute the loop body to the end. Skipping a part of the loop body, you can avoid the nested `if` statements. It will make your code cleaner.

Exercise 3-12. Loop Constructs

Write a game called “More or Fewer”.

The first participant chooses any number from 1 to 100.

The second participant tries to guess it in seven tries.

Your script chooses a number. The user enters his guess.

The script answers if the guess is more or less than the chosen number.

The user then tries to guess the number six more times.

Functions

Bash is the **procedural programming language**³⁵¹. Procedural languages allow you to divide a program into logical parts called **subroutines**³⁵². A subroutine is an independent block of code that solves a specific task. A program calls subroutines when it is necessary.

A subroutine is a deprecated term. It is called **function** in modern programming languages. We have already met functions when considering the `declare` Bash built-in. Now it is time to study them in detail.

Programming Paradigms

We should start with the terminology. It will explain to you why functions were introduced and which tasks they solve.

What is procedural programming? It is one of the **paradigms**³⁵³ of software development. A paradigm is a set of ideas, methods and principles that define how to write programs.

There are two dominant paradigms today. Most modern programming languages follow them. That paradigms are the following:

1. **Imperative programming**³⁵⁴. The developer explicitly specifies to the computer how to change the state of the program. In other words, he writes a complete algorithm for calculating the result.
2. **Declarative programming**³⁵⁵. The developer specifies the properties of the desired result, but not the algorithm to calculate it.

Bash follows the first paradigm. It is an imperative language.

The imperative and declarative paradigms define general principles for writing programs. There are different methodologies (i.e. approaches) within the same paradigm. Each methodology offers specific programming techniques.

The imperative paradigm has two dominant methodologies:

1. Procedural programming.
2. **Object-oriented programming**³⁵⁶.

³⁵¹https://en.wikipedia.org/wiki/Procedural_programming

³⁵²<https://en.wikipedia.org/wiki/Subroutine>

³⁵³https://en.wikipedia.org/wiki/Programming_paradigm

³⁵⁴https://en.wikipedia.org/wiki/Imperative_programming

³⁵⁵https://en.wikipedia.org/wiki/Declarative_programming

³⁵⁶https://en.wikipedia.org/wiki/Object-oriented_programming

Each of these methodologies suggests a specific way for structuring the source code of programs. Bash follows the first methodology.

Let's take a closer look at procedural programming. This methodology suggests features for combining the program instructions into independent code blocks. These blocks are called subroutines or functions.

You can call a function from any place of a program. The function can receive input parameters. This mechanism works similarly to passing command-line parameters to a script. This is a reason why a function is called "a program inside a program" sometimes.

The main task of the functions is to manage the complexity of the source code. The larger size it has, the harder it is to maintain. Repeating code fragments make things worse. They are scattered throughout the program and may contain errors. After fixing a mistake in one fragment, you have to find and fix all the rest. If you put the fragment into a function, it is enough to fix the error only there.

Here is an example of a repeating code fragment. Suppose that you are writing a large program. Whenever some error happens, the program prints the corresponding text message to the error stream. This approach leads to duplicating `echo` calls in the source code. The typical call looks this way:

```
>&2 echo "The N error has happened"
```

At some point, you decide that it is better to write all errors to the log file. It will help you to debug possible issues. Users of your program may redirect the error stream to the log file themselves. This is a good idea, but some users do not know how to use redirection. Thus, your program must write messages into the log file by itself.

You decided to change the way how the program prints error messages. It means that you need to check every place where it happens. You should change the `echo` calls there this way:

```
echo "The N error has happened" >> debug.log
```

If you miss one `echo` call accidentally, its output does not come to the log file. This specific output can be critical for debugging. Without it, you would not understand why the program fails on the user side.

We have considered one of several problems of maintaining programs. The maintenance forces you to change the existing source code. If you have violated the **don't repeat yourself**³⁵⁷ or DRY development principle, you get a lot of troubles. Remember a simple rule: do not copy the same code block of your program.

Functions solve the problem of code duplication. They resemble loops in some sense. The difference is, a loop executes a code block in one place of the program cyclically. In contrast to a loop, a function executes the code block at different program places.

³⁵⁷https://en.wikipedia.org/wiki/Don't_repeat_yourself

Using functions improves the readability of the source code. A function combines a set of commands into a single block. If you give a speaking name to this block, its purpose becomes obvious. Then you can use this name to call the function. It makes your program easier to read. You replace a dozen lines of the function body with its name wherever you call it.

Using Functions in Shell

The functions are available in both Bash modes: shell and script execution. First, let's consider how they work in the shell.

Here is the general form of the function declaration:

```
1 FUNCTION_NAME()  
2 {  
3     ACTION  
4 }
```

You can also declare the function in one line this way:

```
FUNCTION_NAME() { ACTION ; }
```

The semicolon before the closing curly bracket is mandatory here.

The ACTION is a single command or block of commands. It is called the **function body**.

Function names follow the same restrictions as variable names in Bash. You are allowed to use Latin letters, numbers and the underscore character there. The name must not begin with a number.

Let's have a look at how to declare and use functions in the shell. Suppose you need statistics about memory usage. These statistics are available via the special file system **proc**³⁵⁸ or **procfs**. This file system provides the following information:

- The list of running processes.
- The state of the OS.
- The state of the computer hardware.

The default mount point of the **procfs** is the `/proc` path. You can find special files there. They provide you an interface to the kernel data.

You can read the RAM usage statistics in the `/proc/meminfo` file. The `cat` utility prints the file contents to the screen:

³⁵⁸<https://en.wikipedia.org/wiki/Procfs>

```
cat /proc/meminfo
```

The output of this command depends on your OS. The `/proc/meminfo` file provides less information for the MSYS2 environment and more for the Linux system.

Here is an example of the `meminfo` file contents for the MSYS2 environment:

```
1 MemTotal:      6811124 kB
2 MemFree:      3550692 kB
3 HighTotal:    0 kB
4 HighFree:     0 kB
5 LowTotal:    6811124 kB
6 LowFree:     3550692 kB
7 SwapTotal:   1769472 kB
8 SwapFree:    1636168 kB
```

Table 3-22 explains the meaning of these abbreviations.

Table 3-22. Fields of the `meminfo` file

Field	Description
MemTotal	The total amount of usable RAM in the system.
MemFree	The amount of unused RAM at the moment. It is equal to sum of fields LowFree + HighFree.
HighTotal	The total amount of usable RAM in the high region (above 860 MB).
HighFree	The amount of unused RAM in the high region (above 860 MB).
LowTotal	The total amount of usable RAM in the non-high region.
LowFree	The amount of unused RAM in the non-high region.
SwapTotal	The total amount of physical swap ³⁵⁹ memory.
SwapFree	The amount of unused swap memory.

This [article](#)³⁶⁰ provides more details about fields of the `meminfo` file.

You can always call the `cat` utility and get the `meminfo` file contents. However, typing this call takes time. You can shorten it by declaring the function this way:

³⁵⁹https://en.wikipedia.org/wiki/Memory_paging#Unix_and_Unix-like_systems

³⁶⁰<https://www.thegeekdiary.com/understanding-proc-meminfo-file-analyzing-memory-utilization-in-linux/>

```
mem() { cat /proc/meminfo; }
```

This is the one-line declaration of the `mem` function. Now you can call it the same way as any regular Bash built-in. Do it like this:

```
mem
```

This command calls the `mem` function that prints statistics on memory usage.

The `unset` Bash built-in removes the declared function. For example, the following call removes our `mem` function:

```
unset mem
```

Suppose that you have declared a variable and function with the same names. Call `unset` with the `-f` option to remove the function and keep the variable. Here is an example:

```
unset -f mem
```

You can add the function declaration to the `~/.bashrc` file. Then the function will be available whenever you start the shell.

We have declared the `mem` function in the single-line format. It is convenient when you type it in the shell. However, clarity is more important when you declare the function in the `~/.bashrc` file. Therefore, it is better to apply the following format there:

```
1 mem()  
2 {  
3   cat /proc/meminfo  
4 }
```

Difference Between Functions and Aliases

We have declared the `mem` function. It prints statistics on memory usage. The following alias does the same thing:

```
alias mem="cat /proc/meminfo"
```

It looks like functions and aliases work the same way. What should you choose then?

Functions and aliases have one similar aspect only. They are built-in Bash mechanisms. From the user's point of view, they shorten long commands. However, these mechanisms work in completely different ways.

An alias replaces one text with another in a typed command. In other words, Bash finds a part of the command that matches the alias name. Then the shell replaces it with the alias value. Finally, Bash executes the resulting command.

Here is an example of an alias. Suppose you have declared the alias for the `cat` utility. It adds the `-n` option to the utility call. This option adds line numbers to the `cat` output. The alias declaration looks this way:

```
alias cat="cat -n"
```

Whenever you type a command that starts with the word “`cat`”, Bash replaces it with the “`cat -n`”. For example, you type this command:

```
cat ~/.bashrc
```

Bash inserts the alias value here, and the command becomes like this:

```
cat -n ~/.bashrc
```

Bash has replaced the word “`cat`” with “`cat -n`”. It did not change the parameter, i.e. the `~/.bashrc` path.



You can force Bash to insert an alias without executing the resulting command. Type the command and press the `Ctrl+Alt+E` keystroke for doing that.

Now let’s have a look at how functions work. Suppose that Bash meets the function name in the typed command. The shell does not replace the function name with its body, as it does for the alias. Instead, Bash executes the function body.

An example will explain to you how it works. Suppose that you want to write the function that behaves the same way as the `cat` alias. If Bash functions work as aliases, the following declaration should solve your task:

```
cat() { cat -n; }
```

You expect that Bash will add the `-n` option to the following command:

```
cat ~/.bashrc
```

However, it does not happen. Bash does not insert the function body into the command. The shell executes the body and inserts the result into the command.

In our example, Bash calls the `cat` function. The function calls the `cat` utility with the `-n` option, but it ignores the `~/ .bashrc` parameter. You do not want such behavior.

You can solve the problem of ignoring the `~/ .bashrc` parameter. Pass this path to the function as a parameter. This mechanism works similarly to passing a parameter to some command or script. You can call the function and specify its parameters, separated by spaces.

Calling a function and passing parameters to it looks this way in the general form:

```
FUNCTION_NAME PARAMETER_1 PARAMETER_2 PARAMETER_3`
```

You can read parameters in the function body via their names `$1`, `$2`, `$3`, etc. The `$@` array stores all these parameters.

Let's correct the declaration of the `cat` function. You should pass all parameters of the function to the input of the `cat` utility. Then the declaration becomes like this:

```
cat() { cat -n $@; }
```

This function does not work either. The problem happens because of unintentional **recursion**. When some function calls itself, it is called recursion.

Why did we get the recursion? Bash checks the list of declared functions before executing the command `cat -n $@`. There is the `cat` function in the list. Bash executes it at the moment, but it does not change anything. Thus, the shell calls the `cat` function again instead of calling the `cat` utility. This call repeats over and over again. It leads to the infinite recursion, which is similar to an infinite loop.

Recursion is not a mistake of Bash behavior. It is a powerful mechanism that simplifies complex algorithms. An example of such algorithms is the traversing of a **graph**³⁶¹ or **tree**³⁶².

The mistake occurred in our declaration of the `cat` function. The recursive call happens by accident and leads to a loop. There are two ways to solve this problem:

1. Use the command Bash built-in.
2. Give another name to the function that does not conflict with the utility name.

Let's try the first solution. The `command` built-in receives some command on input. If there are aliases or function names there, Bash ignores them. It does not insert the alias value instead of its name. It does not call a function. Instead, Bash executes the command as it is.

If you add the `command` built-in to the `cat` function, you get the following result:

³⁶¹[https://en.wikipedia.org/wiki/Graph_\(abstract_data_type\)](https://en.wikipedia.org/wiki/Graph_(abstract_data_type))

³⁶²[https://en.wikipedia.org/wiki/Tree_\(data_structure\)](https://en.wikipedia.org/wiki/Tree_(data_structure))

```
cat() { command cat -n "$@"; }
```

Now Bash calls the `cat` utility instead of the `cat` function here.

Another solution is renaming the function. For example, this declaration works well:

```
cat_func() { cat -n "$@"; }
```

Always be aware of the problem of unintentional recursion. Give unique names to your functions. They should not match the names of Bash built-ins and GNU utilities.

Here is a summary of our comparison of functions and aliases. If you want to shorten a long command, use an alias.

When using the shell, you need a function in two cases only:

1. You need a conditional statement, loop, or code block to perform your command.
2. The input parameters are not at the end of the command.

The second case needs an example. Let's shorten the `find` utility call. It should search for files in the specified directory. When you search them in the home directory, the `find` call looks this way:

```
find ~ -type f
```

You cannot declare an alias that takes the target path as a parameter. In other words, the following solution does not work:

```
alias="find -type f"
```

The target path should come before the `-type` option. This requirement is a serious problem for the alias.

However, you can declare the function in this case. The function allows you to specify the position to insert the parameter to the `find` call. The declaration looks like this:

```
find_func() { find $1 -type f; }
```

Now you can call the function that searches files in the home directory this way:

```
find_func ~
```

Using Functions in Scripts

You can declare a function in a script the same way as you do it in the shell. Bash allows both full or one-line form there.

For example, let's come back to the task of handling errors in the large program. You can declare the following function for printing error messages:

```
1 print_error()  
2 {  
3     &&2 echo "The error has happened: $@"  
4 }
```

This function expects input parameters. They should explain to a user the root cause of the error. Suppose that your program reads a file on the disk. The file becomes unavailable for some reason. Then the following `print_error` call reports this problem:

```
print_error "the readme.txt file was not found"
```

Suppose that the requirements for the program have changed. Now the program should print error messages to a log file. It is enough to change only the declaration of the `print_error` function to meet the new requirement. The function body looks this way after the change:

```
1 print_error()  
2 {  
3     echo "The error has happened: $@" >> debug.log  
4 }
```

This function prints all error messages to the `debug.log` file. There is no need to change anything at the points where the function is called.

Sometimes you want to call one function from another. This technique is named a **nested function call**. Bash allows it. In general, you can call a function from any point of the program.

Here is an example of a nested function call. Suppose you want to translate the program interface to another language. This task is called **localization**³⁶³. It is better to print error messages in a language the user understands, right? You need to duplicate all messages in all languages supported by your program to reach this requirement.

The straightforward solution for localization is to assign a unique code to each error. Using such codes is a common practice in system programming. Let's apply this approach to your program. Then the `print_error` function will receive an error code via the input parameter.

You can write error codes to the log file as it is. However, it will be inconvenient for a user to interpret these codes. He would need a table to map them into error messages. Therefore, the better solution is to print the text messages to the log file. It means that it is your responsibility to convert error codes to text in a specific language. You would need a separate function for doing this conversion. Here is an example of such a function:

³⁶³https://en.wikipedia.org/wiki/Internationalization_and_localization

```
1 code_to_error()  
2 {  
3     case $1 in  
4         1)  
5             echo "File not found:"  
6             ;;  
7         2)  
8             echo "Permission to read the file denied:"  
9             ;;  
10    esac  
11 }
```

Now you can apply the `code_to_error` function when printing an error in the `print_error` body. You will get the following result:

```
1 print_error()  
2 {  
3     echo "$(code_to_error $1) $2" >> debug.log  
4 }
```

Here is an example of the `print_error` function call from some point of your program:

```
print_error 1 "readme.txt"
```

It prints the following message to the log file:

```
File not found: readme.txt
```

The first parameter of the `print_error` function is the error code. The second parameter is the name of the file that caused the error.

Using functions made the error handling in your program easier to maintain. Changing the requirements demonstrates it. Suppose that your customer asked you to support the German language. You can introduce this feature by declaring two extra functions:

- `code_to_error_en` for messages in English.
- `code_to_error_de` for messages in German.

How can you choose the proper function to convert error codes? The `LANGUAGE` Bash variable helps you in this case. It stores the language that the user has chosen for his system. You should check this variable in the `print_error` function and convert all error codes accordingly.



If the `LANGUAGE` variable is not available on the target system, use the `LANG` variable instead.

Our functions for handling the error codes are just an example for demonstration. Never apply them in your real project. Bash has a special mechanism to localize scripts. It uses PO files with texts in different languages. Read more about this mechanism in the [BashFAQ article](#)³⁶⁴.

Exercise 3-13. Functions

Write the following functions for printing error messages in English and German:

```
* print_error
* code_to_error_en
* code_to_error_de
```

Write two versions of the "code_to_error" function:

```
* Using the case statement.
* Using an associative array.
```

Returning a Function Result

Most procedural languages have a reserved word for returning the function result. It is called `return`. Bash also has a built-in with the same name. However, it has another purpose. The `return` command of Bash does not return a value. Instead, it provides a function exit status to the caller. This status is an integer between 0 and 255.

The complete algorithm of calling and executing the function looks this way:

1. Bash meets the function name in the command.
2. The interpreter goes to the function body and executes it starting from the first command.
3. If Bash meets the `return` command in the function body, it stops executing it. The interpreter jumps to the place where the function was called. The special parameter `?` keeps an exit status of the function.
4. If there is no `return` command in the function body, Bash executes it until the last command. Then, the interpreter jumps to the place where the function was called.

In a typical procedural language, the `return` command returns a variable of any type from a function. It can be a number, string or array. You need other mechanisms for doing that in Bash. There are three options:

1. The command substitution.

³⁶⁴<https://mywiki.woledge.org/BashFAQ/098>

2. A global variable.
3. The caller specifies a global variable.

Let's consider these approaches with examples.

We wrote the `code_to_error` and `print_error` functions to print error messages. Here are their declarations:

```
1 code_to_error()
2 {
3     case $1 in
4         1)
5             echo "File not found:"
6             ;;
7         2)
8             echo "Permission to read the file denied:"
9             ;;
10    esac
11 }
12
13 print_error()
14 {
15     echo "$(code_to_error $1) $2" >> debug.log
16 }
```

Here we have used the first approach for returning the function result. We call the `code_to_error` function using the command substitution. Thus, Bash inserts whatever the function prints to the console instead of its call.

The `code_to_error` function prints the error message using the `echo` command. Then Bash inserts this output to the `print_error` function body. There is only one `echo` call there. It consists of two parts:

1. Output of the `code_to_error` function. It contains an error message.
2. The input parameter `$2` of the `print_error` function. This is the name of the file that caused the error.

The `echo` command of the `print_error` function accumulates all data and prints the final error message to the log file.

The second way to return a value from a function is to write it to some global variable. This kind of variable is available anywhere in the script. Thus, you can access it in the function body and the place where it is called.



All variables declared in the script are global by default. There is one exception to this rule. We will discuss it later.

Let's apply the global variable approach to our case. You should rewrite the `code_to_error` and `print_error` functions for doing that. The first function will write its result to a global variable. Then `print_error` reads it. The resulting code looks this way:

```
1 code_to_error()
2 {
3     case $1 in
4         1)
5             error_text="File not found:"
6             ;;
7         2)
8             error_text="Permission to read the file denied:"
9             ;;
10    esac
11 }
12
13 print_error()
14 {
15     code_to_error $1
16     echo "$error_text $2" >> debug.log
17 }
```

The `code_to_error` function writes its result to the `error_text` global variable. Then the `print_error` function combines this variable with the `$2` parameter to make the final error message and print it to the log file.

Returning a function result via a global variable is the error-prone solution. It may cause a naming conflict. Here is an example of such an error. Suppose that there is another variable called `error_text` in your script. It has nothing to do with the output to the log file. Then any `code_to_error` call will overwrite the value of that variable. This will cause errors in all places where `error_text` is used outside the `code_to_error` and `print_error` functions.

Variable naming convention can solve the problem of naming conflict. The convention is an agreement on how to name the variables in all parts of the project. This agreement is one of the clauses of the [code style](https://en.wikipedia.org/wiki/Programming_style)³⁶⁵ guide. Any large program project must have such a guide.

Here is an example of a variable naming convention:

All global variables, which functions use to return their results, should have an underscore sign prefix in their names.

³⁶⁵https://en.wikipedia.org/wiki/Programming_style

Let's follow this convention in our example. Then you should rename the `error_text` variable to `_error_text`. This way, you solve one specific problem. However, there are cases when a naming conflict can happen. Suppose one function calls another, i.e. there is a nested call. What happens if both functions use the same variable to return their results? You will get the naming conflict again.

The third way to return a function result solves the name conflict problem. The idea is to let the caller the possibility to specify the global variable name. Then the called function writes its result to that variable.

How to pass a variable name to the called function? You can do it using an input parameter. Then the function calls the `eval` built-in. This command converts the specified text into a Bash command. You need this conversion because you passed the variable name as text. Bash does not allow you to refer to the variable using text. So, `eval` resolves this obstacle.

Let's adapt the `code_to_error` function for receiving a global variable name. The function should accept two input parameters:

1. The error code in the `$1` parameter.
2. The name of the global variable to store the result. Use the `$2` parameter for that.

This way, you will get the following code:

```
1 code_to_error()
2 {
3     local _result_variable=$2
4
5     case $1 in
6         1)
7             eval $_result_variable="'File not found:'"
8             ;;
9         2)
10            eval $_result_variable="'Permission to read the file denied:'"
11            ;;
12    esac
13 }
14
15 print_error()
16 {
17     code_to_error $1 "error_text"
18     echo "$error_text $2" >> debug.log
19 }
```

At first glance, the code looks almost the same as it was before. However, it behaves more flexibly now. The `print_error` function chooses the global variable to get the `code_to_error` result. The caller explicitly specifies the variable name. Therefore, it is easier to find and resolve naming conflicts.

Variable Scope

Naming conflict is a serious problem. It occurs when functions declare their variables in the global scope. As a result, the names of two or more variables can match. If functions access these variables at different moments, they overwrite data of each other.

Procedural languages provide the feature that resolves naming conflicts. The idea of this mechanism is to restrict the scope of declared variables.

Bash provides the `local` keyword. Suppose that you declare the variable in some function using this keyword. Then you can access this variable in the function body only. It means that the function body limits the variable scope.

Here is the latest version of the `code_to_error` function:

```
1 code_to_error()
2 {
3     local _result_variable=$2
4
5     case $1 in
6         1)
7             eval $_result_variable="'File not found:'"
8             ;;
9         2)
10            eval $_result_variable="'Permission to read the file denied:'"
11            ;;
12    esac
13 }
```

We have declared the `_result_variable` variable using the `local` keyword. Therefore, it becomes the local variable. You can read and write its value inside `code_to_error` and any other function that it calls.

Bash limits a local variable scope by the execution time of the function where it is declared. Such a scope is called **dynamic**³⁶⁶. Modern languages tend to use **lexical** scope. There the variable is available in the function body only. If you have nested calls, the variable is not available in the called functions.

A local variable does not come to the global scope. It guarantees that no function will overwrite it by accident.

³⁶⁶[https://en.wikipedia.org/wiki/Scope_\(computer_science\)#Lexical_scope_vs._dynamic_scope_2](https://en.wikipedia.org/wiki/Scope_(computer_science)#Lexical_scope_vs._dynamic_scope_2)

Exercise 3-14. Variable scope

What text does the script in Listing 3-37 print to the console when it executes?

Listing 3-37. The script for testing the variable scope

```
1  #!/bin/bash
2
3  bar()
4  {
5      echo "bar1: var = $var"
6      var="bar_value"
7      echo "bar2: var = $var"
8  }
9
10 foo()
11 {
12     local var="foo_value"
13
14     echo "foo1: var = $var"
15     bar
16     echo "foo2: var = $var"
17 }
18
19 echo "main1: var = $var"
20 foo
21 echo "main2: var = $var"
```

Careless handling of local variables leads to errors. They happen because a local variable hides a global variable with the same name.

An example will demonstrate the problem. Suppose that you write a function for processing a file. It calls the `grep` utility to look for a pattern in the file contents. The function looks this way:

```
1  check_license()
2  {
3      local filename="$1"
4      grep "General Public License" "$filename"
5  }
```

Now suppose that you have declared the global variable named `filename` at the beginning of the script. Here is its declaration:

```
1  #!/bin/bash
2
3  filename="$1"
```

Will the `check_license` function work correctly? Yes. It happens thanks to **hiding a global variable**. This mechanism works in the following way. When Bash meets the `filename` variable in the function body, it accesses the local variable instead of the global one. It happens because the local variable is declared later than the global one. When Bash translates the variable name to the memory address, it takes the latest variable declaration. The hiding mechanism has a side effect. It does not allow you to access the `filename` global variable inside the `check_license` function.

When you hide global variables accidentally, you get troubles. The best solution is to avoid any possibility of getting such a situation. Add a prefix or postfix for local variable names for doing that. For example, it can be an underscore at the end of each name.

A global variable becomes unavailable in the function body only after declaring the local variable with the same name there.

Let's consider the following variant of the `check_license` function:

```
1  #!/bin/bash
2
3  filename="$1"
4
5  check_license()
6  {
7      local filename="$filename"
8      grep "General Public License" "$filename"
9  }
```

Here we initialize the local variable `filename` by the value of the global variable with the same name. This assignment works as expected. It happens because Bash does parameter expansion before executing the assignment.

Suppose that you pass the `README` filename to the script. Then you will get this assignment after the parameter expansion:

```
local filename="README"
```

Bash developers changed the default scope of arrays in the 4.2 shell version. If you declare an indexed or associative array in a function body, it comes to the local scope. You should use the `-g` option of the `declare` command to make the array global.

For example, here is the declaration of the `files` local array:

```
1 check_license()  
2 {  
3     declare files=(Documents/*.txt)  
4     grep "General Public License" "$files"  
5 }
```

You should change this declaration if you need the global array. It becomes like this:

```
1 check_license()  
2 {  
3     declare -g files=(Documents/*.txt)  
4     grep "General Public License" "$files"  
5 }
```

We have considered the functions in Bash. Here are general recommendations on how to use them:

1. Choose names for your functions carefully. They should explain the purpose of the functions.
2. Declare only local variables inside functions. Use some naming convention for them. This solves potential conflicts of local and global variable names.
3. Do not use global variables in functions. Instead, pass their values to the functions using input parameters.
4. Do not use the `function` keyword when declaring a function. It presents in Bash, but the POSIX standard does not have it.

Let's take a closer look at the last tip. Do not declare functions this way:

```
1 function check_license()  
2 {  
3     declare files=(Documents/*.txt)  
4     grep "General Public License" "$files"  
5 }
```

There is only one case when the `function` keyword is useful. It resolves the conflict between the names of some function and alias.

For example, the following function declaration does not work without the `function` keyword:

```
1 alias check_license="grep 'General Public License'"
2
3 function check_license()
4 {
5     declare files=(Documents/*.txt)
6     grep "General Public License" "$files"
7 }
```

If you have such a declaration, you can call the function by adding a slash before its name. Here is an example of this call:

```
\check_license
```

If you skip the slash, Bash inserts the alias value instead of calling the function. It means that the following command runs the alias:

```
check_license
```

There is a low probability that you get the conflict of function and alias names in the script. Each script runs in a separate Bash process. This process does not load aliases from the `.bashrc` file. Therefore, name conflicts can happen by mistake in the shell mode only.

Package Manager

You have learned basic Bash built-ins and GNU utilities. These tools are installed on the Unix environment by default. It can happen that they cannot solve your task. In this case, you should find an appropriate program or utility and install it on your own.

Installing software to the Unix environment is not the same as it happens on Windows. Let's have a look at how to install and update the software to any Unix environment or Linux distribution properly.

Repository

Whenever you install the software to the Unix environment, you should use a **repository**³⁶⁷. The repository is a server that stores all available programs. These programs are prepared by **maintainers**³⁶⁸. Maintainers are persons who take open source software and compile it for some repository. Most of these persons are volunteers and free software enthusiasts.

The repository stores each program as a separate file. All these files have the same format. The format depends on the Linux distribution. Thus, each Linux distribution has its own repository. Examples of the formats are DEB, RPM, ZST, etc. A single file with some application is called a **package**. A package is a unit for installing software on your system.

The repository stores packages with applications, libraries and resource files. Besides that, the repository has meta-information about all packages. One or more files store this meta-information. They are called the **package index**.

You can install packages to your Unix environment from several repositories at once. It can be useful in some cases. For example, one repository provides new versions of packages, and another offers special builds of them. Depending on your requirements, you can choose the repository for installing each package.

Package Operating

Each Unix environment provides a special program for accessing the repository. It is called a **package manager**³⁶⁹.

Why does the Unix environment need a package manager? Windows does not have such a program. Users of this OS download all software from the Internet and install it manually.

³⁶⁷<https://help.ubuntu.com/community/Repositories>

³⁶⁸https://en.wikipedia.org/wiki/Software_maintainer

³⁶⁹https://en.wikipedia.org/wiki/Package_manager



There are several third-party package managers for Windows. The most popular one is [Chocolatey](https://chocolatey.org)³⁷⁰. Microsoft plans to develop the [official package manager](https://devblogs.microsoft.com/commandline/windows-package-manager-1-0)³⁷¹ in the nearest future.

The package manager installs and removes packages from the Unix environment. Its main task is to keep track of **package dependencies**. Suppose that some program from one package uses features of the library from another package. Then the first package depends on the second one. It means that you should install the second package whenever you install the first one.

Package dependency allows you to have a single copy of every program and library in your file system. All dependent programs know the installation path of the software they need. This way, they can share it.

You should install all software to your Unix environment or Linux system using the package manager. This rule has one exception. If you need a proprietary program, you have to install it manually. Usually, such a program is distributed in a single package. It includes all dependencies, which are necessary programs and applications. There is no need to track dependencies in this case. Therefore, you can install the program without the package manager.

Here is the algorithm to install a package from the repository:

1. Download a package index from the repository.
2. Find the required program or library in the package index.
3. Download the package with the program or library from the repository.
4. Install the downloaded package.

The package manager does all these steps. It has parameters and options that choose an action to do. You need to know them for installing packages properly.

The MSYS2 environment uses the package manager called [pacman](https://wiki.archlinux.org/index.php/Pacman)³⁷². It was developed for the Arch Linux distribution. The pacman manager operates packages of the ZST format. You do not need any extra knowledge about this format now.

Let's take the pacman manager as an example and consider the commands for accessing the repository.

The following command downloads the package index of the repository:

```
pacman -Syy
```

When you get the package index on your computer, you can find the required package there. This command finds it by the KEYWORD:

³⁷⁰<https://chocolatey.org>

³⁷¹<https://devblogs.microsoft.com/commandline/windows-package-manager-1-0>

³⁷²<https://wiki.archlinux.org/index.php/Pacman>


```
pacman -Ss KEYWORD
```

Suppose that you are looking for a utility for processing MS Word documents. The following command finds the right package for that:

```
pacman -Ss word
```

This command gives you two options:

- `mingw-w64-i686-antiword`
- `mingw-w64-x86_64-antiword`

These are builds of the `antiword` utility for 32-bit and 64-bit systems. The utility converts MS Word documents to text format.

Now you can run the following command that installs the `PACKAGE`:

```
pacman -S PACKAGE
```

This command installs the package with the `antiword` utility:

```
pacman -S mingw-w64-x86_64-antiword
```

When this command finishes, you get the `antiword` utility and all packages that it needs for running.

Now you can launch the `antiword` utility for the `my_report.doc` file this way:

```
antiword my_report.doc
```

This command prints the document contents in text format.

You have installed the new package on your system. If it becomes unnecessary, you can uninstall it. When you do it, the package manager uninstalls both the package and all its dependencies. It happens only when there are no other programs that require them.

Here is the command to uninstall some `PACKAGE`:

```
pacman -Rs PACKAGE
```

The following command uninstalls the package of the `antiword` utility:

```
pacman -Rs mingw-w64-x86_64-antiword
```

Suppose that you have installed several new packages on your system. After a while, maintainers compile new versions of these packages and push them to the repository. You want to get these new versions because of their features. The following command does that:

```
pacman -Syu
```

This command updates all installed packages on your system to their actual versions in the repository.

We have considered the basic `pacman` commands. Other package managers work the same way. They follow the same algorithm as `pacman` when installing and removing packages. However, they have other command-line parameters.

Table 4-1 shows how to use package managers of several well-known Linux distributions.

Table 4-1. The commands of package managers

Command	MSYS2 and Arch Linux	Ubuntu	CentOS	Fedora
Download a package index.	<code>pacman -Syy</code>	<code>apt-get update</code>	<code>yum check-update</code>	<code>dnf check-update</code>
Search for a package by some keyword.	<code>pacman -Ss</code> KEYWORD	<code>apt-cache search</code> KEYWORD	<code>yum search</code> KEYWORD	<code>dnf search</code> KEYWORD
Install the package from the repository.	<code>pacman -S</code> PACKAGE_NAME	<code>apt-get install</code> PACKAGE_NAME	<code>yum install</code> PACKAGE_NAME	<code>dnf install</code> PACKAGE_NAME
Install the package from the local file.	<code>pacman -U</code> FILENAME	<code>dpkg -i FILENAME</code>	<code>yum install</code> FILENAME	<code>dnf install</code> FILENAME
Remove the installed package.	<code>pacman -Rs</code> PACKAGE_NAME	<code>apt-get remove</code> PACKAGE_NAME	<code>yum remove</code> PACKAGE_NAME	<code>dnf erase</code> PACKAGE_NAME
Update all installed packages.	<code>pacman -Syu</code>	<code>apt-get upgrade</code>	<code>yum update</code>	<code>dnf upgrade</code>

Final words

Here we finish our introduction to Bash. We have covered the language basics only. This book skips several important Bash topics. You will need them if you plan to use Bash for professional software development.

Here are the topics that would be a good start for advanced learning of Bash:

- [Strings manipulation](#)³⁷³.
- [Regular expressions](#)³⁷⁴.
- [The sed stream editor](#)³⁷⁵.
- [The awk text processing language](#)³⁷⁶.



GNU utilities `sed` and `awk` are not part of Bash. However, they give you advanced features for processing strings.

These topics are material for advanced study. You can skip them if you are using Bash for simple tasks and basic automation only.

Perhaps you liked writing programs. Now you want to know more about this topic. What is the next step after reading this book?

First, you should accept the idea that Bash is not the **general-purpose programming language**. This term means a language for developing applications in various **domains**³⁷⁷. Such a language does not contain features that are suitable for the specific domain and useless in another one.

Bash is the **domain-specific language**³⁷⁸. It does not mean that Bash is useless for you. It is a powerful auxiliary tool for each software developer. Today it is used for integrating large projects, testing, building software and automating routine tasks. However, you would not find any commercial project written in Bash only. This language copes well with the tasks for which it was created. On the other side, there are many domains where Bash cannot compete with modern general-purpose languages.

Nobody creates some programming language just for fun. Its author faced some applied task. Then he found that all existing languages are inconvenient for solving it. At some point, this developer writes

³⁷³<https://tldp.org/LDP/abs/html/string-manipulation.html>

³⁷⁴<https://tldp.org/LDP/abs/html/x17129.html>

³⁷⁵<https://tldp.org/LDP/abs/html/x23170.html>

³⁷⁶<https://tldp.org/LDP/abs/html/awk.html>

³⁷⁷[https://en.wikipedia.org/wiki/Domain_\(software_engineering\)](https://en.wikipedia.org/wiki/Domain_(software_engineering))

³⁷⁸https://en.wikipedia.org/wiki/Domain-specific_language

the new language that is focused on his specific task. Then there are two options for developing this language.

The first option is the language can provide universal and well-known constructs that fit several domains. It becomes general-purpose in this case. The author adds new features and libraries that adapt the language for various domains.

The second option is the new language has many domain-specific features that are useful only for a limited range of tasks. It becomes domain-specific in this case. The author does not pay efforts for adding any features outside one specific domain.

There is one important conclusion of the typical language development process. All modern general-purpose languages have strong advantages in a few domains only. No language fits any tasks perfectly. It means that the domain dictates you the language to use.

Let's come back to our question regarding your next step for learning programming. You have read this book. Now it is time to choose the applied domain that is interested to you. How can you do it?

You already know something about software from your user experience. Read articles on the Internet about the programs that you use every day. Think about them this way: do you want to develop similar programs? It can happen that your user experience will help you to find the right applied domain.

Suppose that you find the applied domain that you want to learn. The next step is choosing the appropriate programming language that fits this domain well.

Table 5-1 will help you to find the right programming language for your applied domain.

Table 5-1. Applied domains of software development

Domain	Programming language
Mobile applications ³⁷⁹	Java, C, C++, HTML5, JavaScript
Web applications ³⁸⁰ (front end ³⁸¹)	JavaScript, PHP, HTML5, CSS, SQL
Web applications ³⁸² (back end ³⁸³)	JavaScript, Python, Ryby, Perl, C#, Java, Go
High Load Systems ³⁸⁴	C++, Rust, Python, Ruby, SQL
System administration ³⁸⁵	Bash, Python, Perl, Ruby
Embedded systems ³⁸⁶	C, C++, Rust, Assembler

³⁷⁹https://en.wikipedia.org/wiki/Mobile_app

³⁸⁰https://en.wikipedia.org/wiki/Web_application

³⁸¹https://en.wikipedia.org/wiki/Front_end_and_back_end

³⁸²https://en.wikipedia.org/wiki/Web_application

³⁸³https://en.wikipedia.org/wiki/Front_end_and_back_end

³⁸⁴[https://en.wikipedia.org/wiki/Server_\(computing\)](https://en.wikipedia.org/wiki/Server_(computing))

³⁸⁵https://en.wikipedia.org/wiki/System_administrator

³⁸⁶https://en.wikipedia.org/wiki/Embedded_system

Table 5-1. Applied domains of software development

Domain	Programming language
Machine learning ³⁸⁷ and data analysis ³⁸⁸	Python, Java, C++
Information security ³⁸⁹	C, C++, Python, Bash
Enterprise software ³⁹⁰	Java, C#, C++, SQL
Video games ³⁹¹	C++

Unfortunately, it is not enough to know one specific programming language to become a lead developer. You need to know the technologies that are used in the specific applied domain. For example, an information security expert must understand the architecture of computer networks and operating systems. As you grow professionally, you will get the list of technologies that you should learn.

Suppose that you have chosen the applied domain and programming language. Now it is time to enroll in the corresponding online course. This book has introduced you to the basics of programming. Thus, learning the new language will go faster. You will find that some Python and C++ statements look the same as in Bash. However, these languages have concepts that you have to learn from scratch. Do not lose your motivation. Apply the new knowledge into practice and learn from your mistakes. This is the only way to get results.

I hope that you learned something new from this book and had an enjoyable time while reading it. If you like the book, please share it with your friends. I would also appreciate it if you would take a few minutes to score it on [Goodreads](#)³⁹².

If you have any questions or comments about this book, please write to me here: petrsum@gmail.com³⁹³. Also, you can ask questions in the “Issues” section of the [GitHub repository of the book](#)³⁹⁴.

Thank you for reading “Bash programming from scratch”³⁹⁵!

³⁸⁷https://en.wikipedia.org/wiki/Machine_learning

³⁸⁸https://en.wikipedia.org/wiki/Data_analysis

³⁸⁹https://en.wikipedia.org/wiki/Information_security

³⁹⁰https://en.wikipedia.org/wiki/Enterprise_software

³⁹¹https://en.wikipedia.org/wiki/Video_game

³⁹²<https://www.goodreads.com/book/show/57301128-bash-programming-from-scratch>

³⁹³<mailto:petrsum@gmail.com>

³⁹⁴<https://github.com/ellysh/bash-programming-from-scratch/issues>

³⁹⁵<https://leanpub.com/bash-programming-from-scratch/>

Acknowledgements

Nobody writes his book alone. Several people helped me a lot with this work. Some of them suggested just general ideas. Others gave me comments and recommendations. These are the people I want to thank here.

Thanks to Sophia Kayunova for the idea of the programming book for beginners. It gave me a strong motivation to write a book for my friends who asked me about programming.

Thanks to Vitaly Lipatov for introducing me to Linux and Bash. He taught me programming basics and showed me the world of open-source software.

Thanks to Ruslan Piasetskyi for consulting on some Bash topics. He explained to me the idioms and pitfalls of the language.

Thanks also to everyone who supported me and motivated me to finish this work.

Glossary

A

Abstraction³⁹⁶ is a software module, application, or library that replicates the basic properties of some object. Abstractions help to manage the complexity of software systems. They hide irrelevant details. Abstractions allow the same algorithm to handle different objects.

Algorithm³⁹⁷ is a finite sequence of instructions that are understandable for the executor. The goal of an algorithm is to calculate something or solve a specific task.

Alias³⁹⁸ is a built-in Bash command for shortening long strings in the user input. You can apply this command when Bash works in the shell mode.

Application programming interface³⁹⁹ (API) is a set of agreements on interaction components of the information system. The agreements answer the following questions:

- What function does the called component performs?
- What data does the function need on the input?
- What data does the function return on the output?

Argument⁴⁰⁰ is a word or string that the program receives via the command-line interface. Here is an example of two arguments for the `grep` utility:

```
grep "GNU" README.txt
```

Arithmetic expansion⁴⁰¹ calculates an arithmetic expression and substitutes its result into the Bash command or statement. Here is an example of the expansion:

```
echo $((4+3))
```

Array⁴⁰² is a data structure that consists of a set of elements. Their positions are defined by the sequence numbers or indexes. The array elements are placed one after another in computer memory.

ASCII⁴⁰³ is an eight-bit character encoding standard. It includes the following characters:

³⁹⁶[https://en.wikipedia.org/wiki/Abstraction_\(computer_science\)](https://en.wikipedia.org/wiki/Abstraction_(computer_science))

³⁹⁷<https://en.wikipedia.org/wiki/Algorithm>

³⁹⁸https://www.gnu.org/software/bash/manual/html_node/Aliases.html#Aliases

³⁹⁹https://en.wikipedia.org/wiki/Application_programming_interface

⁴⁰⁰http://linuxcommand.org/lc3_wss0120.php

⁴⁰¹https://www.gnu.org/software/bash/manual/html_node/Arithmetic-Expansion.html#Arithmetic-Expansion

⁴⁰²https://en.wikipedia.org/wiki/Array_data_structure

⁴⁰³<https://en.wikipedia.org/wiki/ASCII>

- Decimal digits
- Latin alphabet
- National alphabet
- Punctuation marks
- Control characters

Asynchrony⁴⁰⁴ means events that occur independently of the main program flow. Asynchrony also means the methods for processing such events.

B

Background⁴⁰⁵ is a mode of Bash for process execution. When you apply this mode, the process identifier does not belong to the **identifier group**⁴⁰⁶ of the terminal. Also, the executed process does not handle keyboard interrupts in the terminal window.

Bash⁴⁰⁷ (Bourne again shell) is a command-line interpreter developed by Brian Fox. Bash has replaced the Bourne shell in Linux distributions and some proprietary Unix systems. Bash is compatible with the POSIX standard. However, the standard does not include some features of the interpreter.

Bash script⁴⁰⁸ is a text file that contains commands for the interpreter. Bash executes scripts in the non-interactive mode.

Best practices⁴⁰⁹ are recommended approaches for using a programming language or technology. An example of best practice for Bash is enclosing strings in double quotes to avoid word splitting.

Bottleneck⁴¹⁰ is a component or resource of a computer system that limits its performance.

Boolean expression⁴¹¹ is a programming language construct. When calculated, It results in either the “true” or “false” value.

Bourne shell⁴¹² is a command-line interpreter developed by Stephen Bourne. It replaced the original Ken Thompson’s **interpreter**⁴¹³ in **Unix version 7**⁴¹⁴. The **POSIX standard**⁴¹⁵ contains all features of the Bourne shell. However, the shell misses some features from the standard.

Brace expansion⁴¹⁶ is a Bash mechanism for generating words from given parts. The POSIX standard misses this mechanism.

Here is an example of the brace expansion:

⁴⁰⁴[https://en.wikipedia.org/wiki/Asynchrony_\(computer_programming\)](https://en.wikipedia.org/wiki/Asynchrony_(computer_programming))

⁴⁰⁵https://www.gnu.org/software/bash/manual/html_node/Job-Control-Basics.html#Job-Control-Basics

⁴⁰⁶https://en.wikipedia.org/wiki/Process_group

⁴⁰⁷[https://en.wikipedia.org/wiki/Bash_\(Unix_shell\)](https://en.wikipedia.org/wiki/Bash_(Unix_shell))

⁴⁰⁸https://www.gnu.org/software/bash/manual/html_node/Shell-Scripts.html#Shell-Scripts

⁴⁰⁹<http://mywiki.woledge.org/BashGuide/Practices>

⁴¹⁰[https://en.wikipedia.org/wiki/Bottleneck_\(software\)](https://en.wikipedia.org/wiki/Bottleneck_(software))

⁴¹¹https://en.wikipedia.org/wiki/Boolean_expression

⁴¹²https://en.wikipedia.org/wiki/Bourne_shell

⁴¹³https://en.wikipedia.org/wiki/Thompson_shell

⁴¹⁴https://en.wikipedia.org/wiki/Version_7_Unix

⁴¹⁵https://pubs.opengroup.org/onlinepubs/9699919799/utilities/V3_chap02.html

⁴¹⁶<http://mywiki.woledge.org/BraceExpansion>


```
cp test.{txt,md,log} Documents
```

Bash generates the following command from it:

```
cp test.txt test.md test.log Documents
```

Builtin commands⁴¹⁷ are the built-in commands that the interpreter executes by itself. It does not call utilities or programs for that. An example is `pwd`.

C

Child process⁴¹⁸ is a process spawned by another process called the parent.

Code base⁴¹⁹ is a collection of source code used to build a software system or application. It does not contain generated code by tools.

Code style⁴²⁰ is a set of rules and conventions for writing the program source code. These rules help several programmers write, read, and manage their code base.

Command is the text entered after the command prompt. This text defines the action that the interpreter should perform. If the interpreter cannot do the specified action, it can call another application for that.

Command-line parameter⁴²¹ is a type of command argument that passes information to the program. The parameter can also be a part of some option. For example, it specifies the program working mode.

Here is the `find` utility call with two parameters:

```
find ~/Documents -name README
```

The first parameter `~/Documents` specifies the path where the search starts. The second parameter `README` refers to the option `-name`. It specifies the file or directory name for searching.

Command prompt⁴²² is a sequence of characters. The shell prints a prompt when it is ready to process a user command.

Command substitution⁴²³ is the Bash mechanism that replaces a command call to its output. The subshell executes the command call.

Here is an example of the command substitution:

⁴¹⁷https://www.gnu.org/software/bash/manual/html_node/Shell-Builtin-Commands.html

⁴¹⁸https://en.wikipedia.org/wiki/Child_process

⁴¹⁹<https://en.wikipedia.org/wiki/Codebase>

⁴²⁰https://en.wikipedia.org/wiki/Programming_style

⁴²¹<https://stackoverflow.com/a/36495940/6562278>

⁴²²https://www.gnu.org/software/bash/manual/html_node/Controlling-the-Prompt.html#Controlling-the-Prompt

⁴²³https://www.gnu.org/software/bash/manual/html_node/Command-Substitution.html#Command-Substitution

```
echo "$(date)"
```

Compiler⁴²⁴ is a program for translating source code from some programming language to machine code.

Computer program⁴²⁵ is a set of instructions that a computer can execute. Each program solves a specific applied task.

Conditional statement⁴²⁶ is a construct of a programming language. This statement chooses a block of commands to execute depending on the result of the Boolean expression.

Control character⁴²⁷ is a separate character in the escape sequence.

Control flow⁴²⁸ is an order in which the program executes its instructions and functions.

E

Endianness⁴²⁹ is the byte order that the computer uses to store numbers in memory. The CPU defines the supported endianness. There are two commonly used byte orders: big-endian and little-endian. Some CPUs support both (bi-endian).

Here is an example of representing the four-byte number 0x0A0B0C0D in different byte orders:

```
0A 0B 0C 0D    big-endian
0D 0C 0B 0A    little-endian
```

Environment variables⁴³⁰ is an unordered set of variables that the child process copies from the parent one. The `env` utility can set environment variables when a program starts. If you call the utility without parameters, it prints all variables declared in the current shell.

Error-prone⁴³¹ (error-prone) means failed programming techniques and solutions. These solutions work correctly in particular cases. However, they cause errors with specific input data or conditions. An example of an error-prone solution is handling the `ls` utility output in the pipeline like this:

```
ls | grep "test"
```

Escape sequence⁴³² is a set of characters that have no meaning of their own. Instead, they control the output device. For example, the line break character `\n` commands the output device to start a new line.

⁴²⁴<https://en.wikipedia.org/wiki/Compiler>

⁴²⁵https://en.wikipedia.org/wiki/Computer_program

⁴²⁶[https://en.wikipedia.org/wiki/Conditional_\(computer_programming\)](https://en.wikipedia.org/wiki/Conditional_(computer_programming))

⁴²⁷https://en.wikipedia.org/wiki/Control_character

⁴²⁸https://en.wikipedia.org/wiki/Control_flow

⁴²⁹<https://en.wikipedia.org/wiki/Endianness>

⁴³⁰<http://mywiki.woledge.org/Environment>

⁴³¹<https://en.wiktionary.org/wiki/error-prone>

⁴³²https://en.wikipedia.org/wiki/Escape_sequence

Exit code⁴³³ (or exit status) is an integer value from 0 to 255 that the shell command returns when finishing. The zero status means successful execution of the command. All other codes indicate an error.

F

File descriptor⁴³⁴ is an abstract pointer to some file or communication channel (stream, pipeline or network socket). Descriptors are part of the POSIX interface. Non-negative integers represent them.

Filename expansion⁴³⁵ is a Bash mechanism that replaces patterns with filenames. A pattern can contain the following wildcards: ?, *, [.

Here is an example of using the filename expansion:

```
rm -rf *
```

Filename extension⁴³⁶ is a suffix of the filename. The extension defines a file type.

File system⁴³⁷ is a set of rules to store and read data from a storage device. Also, a file system means a component of OS to manage storage devices.

Foreground⁴³⁸ is a default process execution mode in Bash. When you apply it, the process identifier belongs to the **identifier group**⁴³⁹ of the terminal. The executed process handles keyboard interrupts in the terminal window.

Function is a modern name for a subroutine.

G

General-purpose programming language⁴⁴⁰ is a language that you can use to develop applications for various applied domains. It does not contain constructs that are useful in one domain and useless in others.

Globbering⁴⁴¹ or glob is another name for the filename expansion mechanism of Bash.

Glob pattern⁴⁴² is a search query. It includes regular and **wildcard characters**⁴⁴³ (* and ?). The wildcards correspond to any characters.

For example, the “R*M?” pattern matches strings that begin with R and whose penultimate letter is M.

⁴³³https://www.gnu.org/software/bash/manual/html_node/Exit-Status.html#Exit-Status

⁴³⁴https://en.wikipedia.org/wiki/File_descriptor

⁴³⁵https://www.gnu.org/software/bash/manual/html_node/Filename-Expansion.html#Filename-Expansion

⁴³⁶https://en.wikipedia.org/wiki/Filename_extension

⁴³⁷https://en.wikipedia.org/wiki/File_system

⁴³⁸https://www.gnu.org/software/bash/manual/html_node/Job-Control-Basics.html#Job-Control-Basics

⁴³⁹https://en.wikipedia.org/wiki/Process_group

⁴⁴⁰https://en.wikipedia.org/wiki/General-purpose_programming_language

⁴⁴¹<https://mywiki.woledge.org/glob?action=show&redirect=globbering>

⁴⁴²[https://en.wikipedia.org/wiki/Glob_\(programming\)](https://en.wikipedia.org/wiki/Glob_(programming))

⁴⁴³https://en.wikipedia.org/wiki/Wildcard_character

H

Hash function⁴⁴⁴ generates a unique sequence of bytes from the input data.

[calculates the keys for added elements.

I

Idiom⁴⁴⁵ is a way to express some statement using a specific programming language. An idiom can be a template for implementing an algorithm or data structure.

Here is the Bash idiom for using patterns when processing files in the `for` loop:

```
1 for file in ./*.txt
2 do
3   cp "$file" ~/Documents
4 done
```

Input field separator⁴⁴⁶ (IFS) is a list of characters. Bash uses them as separators when processing input strings. For example, it uses them for word splitting. The default separators are space, tab and a line break.

Interpreter⁴⁴⁷ is a program that executes instructions. They are written in a programming language. The interpreter allows you to execute them without compilation.

Iteration⁴⁴⁸ is a single execution of a command block in the loop body.

L

Library⁴⁴⁹ is a collection of subroutines and data structures assembled into a standalone module. Applications use libraries as building blocks.

Linked list⁴⁵⁰ is a data structure that consists of elements called nodes. Their positions in the list do not match their placement in memory. Therefore, each node has a pointer to the next one. Such list organization makes insertion and deletion operations effective.

Linux distribution⁴⁵¹ is an operating system based on the Linux kernel and the **GNU packages**⁴⁵². Such an OS is assembled from open-source programs and libraries.

⁴⁴⁴https://en.wikipedia.org/wiki/Hash_function

⁴⁴⁵https://en.wikipedia.org/wiki/Programming_idiom

⁴⁴⁶<https://mywiki.woledge.org/IFS>

⁴⁴⁷[https://en.wikipedia.org/wiki/Interpreter_\(computing\)](https://en.wikipedia.org/wiki/Interpreter_(computing))

⁴⁴⁸<https://en.wikipedia.org/wiki/Iteration#Computing>

⁴⁴⁹[https://en.wikipedia.org/wiki/Library_\(computing\)](https://en.wikipedia.org/wiki/Library_(computing))

⁴⁵⁰https://en.wikipedia.org/wiki/Linked_list

⁴⁵¹https://en.wikipedia.org/wiki/Linux_distribution

⁴⁵²https://en.wikipedia.org/wiki/List_of_GNU_packages

Linux environment is another name for a POSIX environment.

Literal⁴⁵³ is a notation for some fixed value in the source code of a program. There are different ways to write literals depending on their data types. Most programming languages support literals for integers, **floating point numbers**⁴⁵⁴ and strings.

Here is an example of the string literal `~/Documents` in Bash:

```
1 var="~/Documents"
```

Logical operator⁴⁵⁵ is an operation on one or more Boolean expressions. The operation can combine them into a single expression.

M

Multiprogramming⁴⁵⁶ is a technique for distributing computer resources among several programs. For example, a program runs until it needs some busy resource. Then OS switches to another program. It comes back to the first program when the required resource becomes free.

Multitasking⁴⁵⁷ is the execution of several tasks (processes) in parallel. The OS does it by **switching**⁴⁵⁸ between tasks frequently and executing them in parts.

N

Network protocol⁴⁵⁹ agrees on the format of messages between nodes of a computer network.

O

Operand⁴⁶⁰ is an argument of some operation or command. It represents the data to be processed. Here is an example of operands 1 and 4 for the addition operation:

```
1 + 4
```

Option⁴⁶¹ is an argument in a standardized form, which the program receives as input. Options of GNU utilities begin with a dash - or a double dash --. Each option sets a specific mode of the program. You can combine successive options into one group.

Here is an example of grouping the -l, -a and -h options of the `ls` utility:

⁴⁵³[https://en.wikipedia.org/wiki/Literal_\(computer_programming\)](https://en.wikipedia.org/wiki/Literal_(computer_programming))

⁴⁵⁴https://en.wikipedia.org/wiki/Floating-point_arithmetic

⁴⁵⁵https://en.wikipedia.org/wiki/Logical_connective

⁴⁵⁶<https://www.geeksforgeeks.org/difference-between-multitasking-multithreading-and-multiprocessing>

⁴⁵⁷https://en.wikipedia.org/wiki/Computer_multitasking

⁴⁵⁸https://en.wikipedia.org/wiki/Context_switch

⁴⁵⁹https://en.wikipedia.org/wiki/Communication_protocol

⁴⁶⁰<https://en.wikipedia.org/wiki/Operand>

⁴⁶¹http://linuxcommand.org/lc3_wss0120.php

ls -lah

P

Parameter⁴⁶² is an entity that transfers some value from one part of the system to another. A parameter may have no name, unlike a variable.

Parameter expansion⁴⁶³ is the Bash mechanism that replaces a variable name with its value. Here are two examples of the parameter expansion:

```
echo "$PATH"
echo "${var:-empty}"
```

Pipeline⁴⁶⁴ is a process communication mechanism of Unix-like operating systems. The mechanism is based on passing messages. Also, the pipeline means two or more processes with connected input and output streams. OS sends the output stream of one process directly to the input stream of another.

Portable operating system interface⁴⁶⁵ (POSIX) is a set of standards. They describe the API of a portable OS, its shell and the utility interfaces. POSIX guarantees compatibility of OSes from the Unix family. This compatibility allows you to move programs between systems with minimal efforts.

Positional parameters⁴⁶⁶ contain all command-line arguments that the Bash script receives as input. Parameter names match the order of arguments.

Here is an example of using the first positional parameter in a script:

```
cp "$1" ~
```

POSIX environment is a software environment that is compatible with the POSIX standard. The full compatibility is available only when the OS kernel, shell and file system follow POSIX. Environments like **Cygwin**⁴⁶⁷ provide partial compatibility.

POSIX shell⁴⁶⁸ is a standard for POSIX systems that describes a minimum set of shell features. If the shell provides these features, it is considered POSIX compatible. The standard does not restrict additional features and extensions. The standard is based on the ksh88 implementation of the **Korn shell**⁴⁶⁹. This interpreter appeared later than the Bourne shell. Therefore, the Bourne shell misses some features of the POSIX standard.

Process⁴⁷⁰ is an instance of a computer program that the CPU executes.

⁴⁶²<http://www.gnu.org/savannah-checkouts/gnu/bash/manual/bash.html#Shell-Parameters>

⁴⁶³https://www.gnu.org/software/bash/manual/html_node/Shell-Parameter-Expansion.html#Shell-Parameter-Expansion

⁴⁶⁴[https://en.wikipedia.org/wiki/Pipeline_\(Unix\)](https://en.wikipedia.org/wiki/Pipeline_(Unix))

⁴⁶⁵<https://en.wikipedia.org/wiki/POSIX>

⁴⁶⁶https://www.gnu.org/software/bash/manual/html_node/Shell-Parameters.html#Shell-Parameters

⁴⁶⁷<https://en.wikipedia.org/wiki/Cygwin>

⁴⁶⁸<https://www.grymoire.com/Unix/Sh.html>

⁴⁶⁹<https://en.wikipedia.org/wiki/KornShell>

⁴⁷⁰[https://en.wikipedia.org/wiki/Process_\(computing\)](https://en.wikipedia.org/wiki/Process_(computing))

Process substitution⁴⁷¹ is the Bash mechanism that resembles command substitution. It executes the command and provides its output to the Bash process. The mechanism transfers data via temporary files.

Here is an example of using process substitution:

```
diff <(sort file1.txt) <(sort file2.txt)
```

Programming paradigm⁴⁷² is a set of ideas, methods and principles that dictate how to write programs.

Q

Quote removal⁴⁷³ is the Bash mechanism that removes the following three unescaped characters from the strings: \ ' “.

R

Recursion⁴⁷⁴ is a case when some function calls itself. This case is called the direct recursion. The indirect recursion happens when the function calls itself through other functions.

Redirections⁴⁷⁵ are the special shell constructs that redirect I/O streams of executed commands. You should specify file descriptors as the source and target for a redirection. A descriptor can point to a file or standard stream.

Here is an example of redirecting the `find` utility output to the `result.txt` file:

```
find / -path */doc/* -name README 1> result.txt
```

Reserved variables mean the same as shell variables.

S

Scope⁴⁷⁶ is a part of a program or system where the variable name and value remain associated with each other. In other words, the program can convert the variable name into the memory address inside the scope. This conversion does not work outside the scope.

Shebang⁴⁷⁷ is a sequence of a number sign and exclamation mark (`#!`) at the beginning of the script. The program loader treats the string after shebang as the interpreter name. The loader launches the interpreter and passes the script to it for execution.

Here is an example of shebang for the Bash script:

⁴⁷¹https://www.gnu.org/software/bash/manual/html_node/Process-Substitution.html#Process-Substitution

⁴⁷²https://en.wikipedia.org/wiki/Programming_paradigm

⁴⁷³https://www.gnu.org/software/bash/manual/html_node/Quote-Removal.html#Quote-Removal

⁴⁷⁴[https://en.wikipedia.org/wiki/Recursion_\(computer_science\)](https://en.wikipedia.org/wiki/Recursion_(computer_science))

⁴⁷⁵https://www.gnu.org/software/bash/manual/html_node/Redirections.html#Redirections

⁴⁷⁶[https://en.wikipedia.org/wiki/Scope_\(computer_science\)](https://en.wikipedia.org/wiki/Scope_(computer_science))

⁴⁷⁷[https://en.wikipedia.org/wiki/Shebang_\(Unix\)](https://en.wikipedia.org/wiki/Shebang_(Unix))

1 `#!/bin/bash`

Shell options⁴⁷⁸ change the interpreter behavior. They affect both shell and script modes. The built-in `set` command sets these options.

For example, here is the command to enable the debug output from the interpreter:

```
set -x
```

Shell parameter⁴⁷⁹ is a named area of the shell memory for storing data.

Shell variables⁴⁸⁰ are set by the shell (for example, `PATH`). They store temporary data, settings and states of the OS or Unix environment. A user can read these variables. Only some of them are writable. The `set` command prints all shell variables.

Short-circuit evaluation⁴⁸¹ (short-circuit) is the approach to limit calculations when deducing Boolean expression. The idea is to calculate only those operands that are sufficient to deduce the whole expression.

Special parameters⁴⁸² are set by the shell. They perform the following tasks:

1. Store the shell state.
2. Pass parameters to the called program.
3. Store the exit status of the finished program.

Special parameters are read-only. An example of such a parameter is `$?`.

Standard streams⁴⁸³ are software channels of communication between a program and the environment where it operates. Streams are abstractions of physical channels of input from the keyboard and output to the screen. You can operate the channels via their descriptors. OS assigns these descriptors.

Subroutine⁴⁸⁴ is a fragment of a program that performs a single task. This fragment is an independent code block. It can be called from any place of the program.

Subshell⁴⁸⁵ is a way of grouping shell commands. A child process executes the grouped commands. Variables defined in the child process are not available in the parent process.

Here is an example of executing commands in a subshell:

```
(ps aux | grep "bash")
```

⁴⁷⁸<https://www.tldp.org/LDP/abs/html/options.html>

⁴⁷⁹<http://mywiki.woledge.org/BashGuide/Parameters>

⁴⁸⁰http://tldp.org/LDP/Bash-Beginners-Guide/html/sect_03_02.html

⁴⁸¹https://en.wikipedia.org/wiki/Short-circuit_evaluation

⁴⁸²<http://mywiki.woledge.org/BashGuide/Parameters>

⁴⁸³https://en.wikipedia.org/wiki/Standard_streams

⁴⁸⁴<https://en.wikipedia.org/wiki/Subroutine>

⁴⁸⁵<http://mywiki.woledge.org/SubShell>

Symbolic link⁴⁸⁶ is a special type of file. Instead of data, it contains a pointer to another file or directory.

Synchronous⁴⁸⁷ means events or actions that occur in the main program flow.

T

Tilde expansion⁴⁸⁸ (tilde expansion) is the Bash mechanism that replaces the ~ character with the user home directory. The shell takes the path to this directory from the HOME variable.

Time-sharing⁴⁸⁹ is a computer mode when several users utilize its resources simultaneously. This mode is based on multitasking and multiprogramming.

U

Unix environment⁴⁹⁰ is another name for the POSIX environment.

Utility software⁴⁹¹ is a special program for managing the OS or hardware.

V

Variable⁴⁹² is an area of memory that can be accessed by its name.

There is another meaning of this term for Bash. A variable is a parameter that is accessible by its name. The user or the interpreter defines variables.

Here is an example of the variable declaration:

```
filename="README.txt"
```

Vulnerability⁴⁹³ is a bug or flaw of the computing system. An attacker can perform unauthorized actions using the vulnerability.

W

Word splitting⁴⁹⁴ is the Bash mechanism that splits command-line arguments into words. Then Bash passes these words to the command as separate parameters. The mechanism uses the characters from the IFS variable as delimiters. It skips arguments that are enclosed in quotes.

Here is an example of a command where word splitting takes place:

⁴⁸⁶https://en.wikipedia.org/wiki/Symbolic_link

⁴⁸⁷<https://en.wiktionary.org/wiki/synchronous>

⁴⁸⁸https://www.gnu.org/software/bash/manual/html_node/Tilde-Expansion.html#Tilde-Expansion

⁴⁸⁹<https://en.wikipedia.org/wiki/Time-sharing>

⁴⁹⁰<https://ccrma.stanford.edu/guides/planetccrma/Unix.html>

⁴⁹¹https://en.wikipedia.org/wiki/Utility_software

⁴⁹²https://www.gnu.org/software/bash/manual/html_node/Shell-Parameters.html#Shell-Parameters

⁴⁹³[https://en.wikipedia.org/wiki/Vulnerability_\(computing\)](https://en.wikipedia.org/wiki/Vulnerability_(computing))

⁴⁹⁴https://www.gnu.org/software/bash/manual/html_node/Word-Splitting.html#Word-Splitting

```
cp file1.txt file2.txt "my file.txt" ~
```

Solutions for Exercises

General Information

Exercise 1-1. Numbers conversion from binary to hexadecimal

* 10100110100110 = 0010 1001 1010 0110 = 2 9 A 6 = 29A6

* 1011000111010100010011 = 0010 1100 0111 0101 0001 0011 = 2 C 7 5 1 3 = 2C7513

* 1111101110001001010100110000000110101101 =
1111 1011 1000 1001 0101 0011 0000 0001 1010 1101 = F B 8 9 5 3 0 1 A D =
FB895301AD

Exercise 1-2. Numbers conversion from hexadecimal to binary

* FF00AB02 = F F 0 0 A B 0 2 = 1111 1111 0000 0000 1010 1011 0000 0010 =
11111111000000001010101100000010

* 7854AC1 = 7 8 5 4 A C 1 = 0111 1000 0101 0100 1010 1100 0001 =
1111000010101001010110000001

* 1E5340ACB38 = 1 E 5 3 4 0 A C B 3 8 =
001 1110 0101 0011 0100 0000 1010 1100 1011 0011 1000 =
11110010100110100000010101100101100111000

Bash Shell

Exercise 2-1. Glob patterns

The correct answer is “README.md”.

The “00_README.txt” string does not fit. It happens because the “*ME.??” pattern requires two characters after the dot. However, the string has three characters.

The “README” string does not fit because it does not have a dot.

Exercise 2-2. Glob patterns

The following three lines match the “`/doc?openssl`” pattern:

- `/usr/share/doc/openssl/IPAddressChoice_new.html`
- `/usr/share/doc_openssl/IPAddressChoice_new.html`
- `/doc/openssl`

The “`doc/openssl`” string does not fit. It does not have the slash symbol before the “`doc`” word.

Exercise 2-3. Searching for files with the `find` utility

The following command searches text files in the system paths:

```
find /usr -name "*.txt"
```

The `/usr` path stores text files. So, there is no reason to check other system paths.

Now let’s count the number of lines in the found files. The `wc` utility can do this task. We should call the utility using the `-exec` action. Then the resulting command looks like this:

```
find /usr -name "*.txt" -exec wc -l {} +
```

You can find all text files on the disk if you start searching from the root directory. Here is the example command:

```
find / -name "*.txt"
```

If you add the `wc` call to the command, it fails when running in the `MSYS2` environment. In other words, the following command does not work:

```
find / -name "*.txt" -exec wc -l {} +
```

The problem happens because of the error message that Figure 2-17 shows. The `find` utility passes its error message to the `wc` input. The `wc` utility treats each word it receives as a file path. The error message is not a path. Therefore, `wc` fails.

Exercise 2-4. Searching for files with the `grep` utility

Look for information about application licenses in the `/usr/share/doc` system path. It contains documentation for all installed software.

If the program has the [GNU General Public License](https://en.wikipedia.org/wiki/GNU_General_Public_License)⁴⁹⁵, its documentation contains the “General Public License” phrase. The following command searches this phrase in all documents:

⁴⁹⁵https://en.wikipedia.org/wiki/GNU_General_Public_License

```
grep -Rl "General Public License" /usr/share/doc
```

The `/usr/share/licenses` path is the place where you can find license information for all installed software. You can search the “General Public License” phrase there with the following command:

```
grep -Rl "General Public License" /usr/share/licenses
```

The MSYS2 environment has two extra paths for installing programs: `/mingw32` and `/mingw64`. They do not match the POSIX standard. The following `grep` calls check these paths:

```
1 grep -Rl "General Public License" /mingw32/share/doc
2 grep -Rl "General Public License" /mingw64/share
```

You can find applications with other licenses than GNU General Public License. Here is the list of licenses and corresponding phrases for searching:

- MIT - “MIT license”
- Apache - “Apache license”
- BSD - “BSD license”

Exercise 2-6. Operations with files and directories

First, you should create directories for saving photos. Each directory should match a specific month of the year. The following commands create them:

```
1 mkdir -p ~/photo/2019/11
2 mkdir -p ~/photo/2019/12
3 mkdir -p ~/photo/2020/01
```

Suppose that the `D:\Photo` directory contains all your photos. You can use the `find` utility to search photos created in November 2019. The `-newermt` option of the utility checks the creation date. Here is an example command of how to use it:

```
find /d/Photo -type f -newermt 2019-11-01 ! -newermt 2019-12-01
```

This command looks for files in the `/d/Photo` directory. It matches the `D:\Photo` path in the Windows environment.

The first expression `-newermt 2019-11-01` points to files changed since November 1, 2019. The second expression `! -newermt 2019-12-01` excludes files modified since December 1, 2019. The exclamation point before the expression is a negation. There is no condition between expressions. The `find` utility inserts the logical AND by default. The resulting expression looks like: “files created after November 1, 2019, but no later than November 30, 2019”. It means “the files created in November 2019”.

The file search command is ready. Now we should add the copy action there. The result looks like this:

```
find /d/Photo -type f -newermt 2019-11-01 ! -newermt 2019-12-01 -exec cp {} ~/photo/\
2019/11 \;
```

This command copies the files created in November 2019 into the ~/photo/2019/11 directory.

Here are the similar commands for copying the December and January files:

```
1 find /d/Photo -type f -newermt 2019-12-01 ! -newermt 2020-01-01 -exec cp {} ~/photo/\
2 2019/12 \;
3 find /d/Photo -type f -newermt 2020-01-01 ! -newermt 2020-02-01 -exec cp {} ~/photo/\
4 2020/01 \;
```

Let's assume that you do not need old files in the D:\Photo directory. Then you should replace the copying action with renaming. This way, you get the following commands:

```
1 find /d/Photo -type f -newermt 2019-11-01 ! -newermt 2019-12-01 -exec mv {} ~/photo/\
2 2019/11 \;
3 find /d/Photo -type f -newermt 2019-12-01 ! -newermt 2020-01-01 -exec mv {} ~/photo/\
4 2019/12 \;
5 find /d/Photo -type f -newermt 2020-01-01 ! -newermt 2020-02-01 -exec mv {} ~/photo/\
6 2020/01 \;
```

Note the scalability of our solution. The number of files in the D:\Photo directory does not matter. You need only three commands to process them.

Exercise 2-7. Pipelines and I/O streams redirection

First, let's figure out how the `bsdtar` utility works. Call it with the `--help` option. It shows you a brief help. The help tells you that the utility archives a target directory if you apply the `-c` and `-f` options. You should specify the archive name after these options. Here is an example call of the `bsdtar` utility:

```
bsdtar -c -f test.tar test
```

This command creates the `test.tar` archive. It includes the contents of the `test` directory. Note that the command does not [compress data](https://en.wikipedia.org/wiki/Data_compression)⁴⁹⁶. It means that the archive occupies the same disk space as the files it contains.

The purposes of archiving and compression operations differ. You need archiving for storing and copying a large number of files. Compression reduces the amount of the occupied disk memory. These operations are combined into one often, but they are not the same.

Suppose that you want to create an archive and compress it. Then you need to add the `-j` option to the `bsdtar` call. Here is an example:

⁴⁹⁶https://en.wikipedia.org/wiki/Data_compression

```
bsdtar -c -j -f test.tar.bz2 test
```

You can combine the `-c`, `-j` and `-f` options into one group. Then you get the following command:

```
bsdtar -cjf test.tar.bz2 test
```

Let's write a command for processing all photos. The command should archive each directory with the photos of the specific month.

First, you need to find the directories for archiving. The following command does it:

```
find ~/photo -type d -path */2019/* -o -path */2020/*
```

Next, you redirect the output of the `find` call to the `xargs` utility. It will generate the `bsdtar` call. This way, you get the following command:

```
find ~/photo -type d -path */2019/* -o -path */2020/* | xargs -I% bsdtar -cf %.tar %
```

You can add the `-j` option to force `bsdtar` to compress archived data. The command became like this:

```
find ~/photo -type d -path */2019/* -o -path */2020/* | xargs -I% bsdtar -cjf %.tar.\bz2 %
```



The data compression usually takes longer than archiving.

We pass the `-I` parameter to the `xargs` utility. It specifies the place to insert the arguments into the generated command. There are two such places in the `bsdtar` utility call: the archive's name and the directory's path to be processed.

Do not forget about filenames with line breaks. To process them correctly, add the `-print0` option to the `find` call. This way, you get the following command:

```
find ~/photo -type d -path */2019/* -o -path */2020/* -print0 | xargs -0 -I% bsdtar \-cjf %.tar.bz2 %
```

Suppose that you want to keep the files in the archives without relative paths (e.g. `2019/11`). The `--strip-components` option of `bsdtar` removes them. The following command uses this option:

```
find ~/photo -type d -path */2019/* -o -path */2020/* -print0 | xargs -0 -I% bsdtar \
--strip-components=3 -cjf %.tar.bz2 %
```

Exercise 2-8. Logical operators

Let's implement the algorithm step by step. The first action is to copy the README file to the user's home directory. The following command does it:

```
cp /usr/share/doc/bash/README ~
```

Apply the `&&` operator and the `echo` built-in to print the `cp` result into the log file. Then you get the following command:

```
cp /usr/share/doc/bash/README ~ && echo "cp - OK" > result.log
```

The second step is archiving the file. You can call the `bsdtar` or `tar` utility for that. Here is an example:

```
bsdtar -cjf ~/README.tar.bz2 ~/README
```

Add `echo` built-in to print the result of the archiving utility. The command becomes like this:

```
bsdtar -cjf ~/README.tar.bz2 ~/README && echo "bsdtar - OK" >> result.log
```

Here the `echo` command appends the string to the end of the existing log file.

Let's combine the `cp` and `bsdtar` utilities into one command. You should call `bsdtar` only if the README file has been copied successfully. To achieve this dependency, put the `&&` operator between the calls. This way, you get the following command:

```
cp /usr/share/doc/bash/README ~ && echo "cp - OK" > result.log && bsdtar -cjf ~/READ\
ME.tar.bz2 ~/README && echo "bsdtar - OK" >> result.log
```

The last step is deleting the README file. Do it by the `rm` call this way:

```
cp /usr/share/doc/bash/README ~ && echo "cp - OK" > ~/result.log && bsdtar -cjf ~/RE\
ADME.tar.bz2 ~/README && echo "bsdtar - OK" >> ~/result.log && rm ~/README && echo "\
rm - OK" >> ~/result.log
```

Run this command in your terminal. If it succeeds, you get the following log file:

- 1 cp - OK
- 2 bsdtar - OK
- 3 rm - OK

The current version of our command calls three utilities in a row. It looks cumbersome and inconvenient for reading. Let's break the command into lines. There are several ways to do that.

The first way is to break lines after logical operators. If you apply this approach, you get the following result:

- ```
1 cp /usr/share/doc/bash/README ~ && echo "cp - OK" > ~/result.log &&
2 bsdtar -cjf ~/README.tar.bz2 ~/README && echo "bsdtar - OK" >> ~/result.log &&
3 rm ~/README && echo "rm - OK" >> ~/result.log
```

Copy this command to your terminal and execute it. It works properly.

The second way to add line breaks is to use the backslash symbol. Put a line break right after it. This method fits well when there are no logical operators in the command.

For example, you can put backslashes before the && operators in our command. Then you get this result:

- ```
1 cp /usr/share/doc/bash/README ~ && echo "cp - OK" > ~/result.log \
2 && bsdtar -cjf ~/README.tar.bz2 ~/README && echo "bsdtar - OK" >> ~/result.log \
3 && rm ~/README && echo "rm - OK" >> ~/result.log
```

Run this command in your terminal. It works properly too.

Bash Scripts

Exercise 3-2. The full form of parameter expansion

The `find` utility searches for files recursively. It starts from the specified path and passes through all subdirectories. If you want to exclude subdirectories from the search, apply the `-maxdepth` option.

Here is the command for searching TXT files in the current directory:

```
find . -maxdepth 1 -type f -name "*.txt"
```

Let's add an action to copy the found files to the user's home directory. The command becomes like this:

```
find . -maxdepth 1 -type f -name "*.txt" -exec cp -t ~ {} \;
```

Now create a script and name it `txt-copy.sh`. Copy the `find` call to this file.

The script should choose an action to apply for each found file. It can be copying or moving. The straightforward way is to pass the parameter to the script. This parameter defines the required action.

You can choose any values for parameters. However, the most obvious values are the names of GNU utilities to perform actions on files. The names are `cp` and `mv`. If you pass them, the script can extract the utility name from the parameter and call it.

The script copies found files when you call it like this:

```
./txt-copy.sh cp
```

If you need to move files, you call the script this way:

```
./txt-copy.sh mv
```

The first parameter of the script is available via the `$1` variable. Expand it in the `-exec` action of the `find` call. Then you get the following command:

```
find . -maxdepth 1 -type f -name "*.txt" -exec "$1" -t ~ {} \;
```

If you do not specify an action, the script should copy the files. It means that the following call should work well:

```
./txt-copy.sh
```

Add a default value to the parameter expansion to handle the case of the missing parameter.

Listing 5-1 shows the final script that you get this way.

Listing 5-1. The script for copying TXT files

```
1 #!/bin/bash
2
3 find . -maxdepth 1 -type f -name "*.txt" -exec "${1:-cp}" -t ~ {} \;
```

Exercise 3-4. The `if` statement

The original command looks this way:

```
( grep -RLZ "123" target | xargs -0 cp -t . && echo "cp - OK" || ! echo "cp - FAILS" \
) && ( grep -RLZ "123" target | xargs -0 rm && echo "rm - OK" || echo "rm - FAILS" \
)
```

Note the negation of the echo call with the “cp - FAILS” output. The negation prevents the second grep call if the first one fails.

First, replace the && operator between grep calls with the if statement. Then you get the following code:

```
1 if grep -RLZ "123" target | xargs -0 cp -t .
2 then
3   echo "cp - OK"
4   grep -RLZ "123" target | xargs -0 rm && echo "rm - OK" || echo "rm - FAILS"
5 else
6   echo "cp - FAILS"
7 fi
```

Next, replace the || operator in the second grep call with the if statement. The result looks like this:

```
1 if grep -RLZ "123" target | xargs -0 cp -t .
2 then
3   echo "cp - OK"
4   if grep -RLZ "123" target | xargs -0 rm
5   then
6     echo "rm - OK"
7   else
8     echo "rm - FAILS"
9   fi
10 else
11   echo "cp - FAILS"
12 fi
```

You get the nested if statement. Apply the early return pattern to get rid of it.

The last step is adding the shebang at the beginning of the script. Listing 5-2 shows the final result.

Listing 5-2. The script for searching a string in files

```
1  #!/bin/bash
2
3  if ! grep -RLZ "123" target | xargs -0 cp -t .
4  then
5      echo "cp - FAILS"
6      exit 1
7  fi
8
9  echo "cp - OK"
10
11 if grep -RLZ "123" target | xargs -0 rm
12 then
13     echo "rm - OK"
14 else
15     echo "rm - FAILS"
16 fi
```

Exercise 3-5. The [[operator

Let's compare the contents of the two directories. The result of the comparison is a list of files that differ.

First, you have to pass through all files in each directory. The `find` utility does this task. Here is a command to search files in the `dir1` directory:

```
find dir1 -type f
```

Here is an example output of the command:

```
dir1/test3.txt
dir1/test1.txt
dir1/test2.txt
```

You got a list of files in the `dir1` directory. Next, you should check that each of them presents in the `dir2` directory. Add the following `-exec` action for this check:

```
1  cd dir1
2  find . -type f -exec test -e ../dir2/{} \;
```

Here you should use the `test` command instead of the `[[` operator. The reason is the built-in interpreter of the `find` utility can not handle this Bash operator. This is one of the few exceptions when you need to apply the `test` command. In general, the `[[` operator should be used instead.

If the `dir2` directory does not contain some file, let's print its name. You need two things to do that. The first one is inverting the `test` command result. Second, you should add an extra `-exec` action with the `echo` call. Place the logical AND these two `-exec` actions. This way, you get the following command:

```
1 cd dir1
2 find . -type f -exec test ! -e ../dir2/{} \; -a -exec echo {} \;
```

You found files of the `dir1` directory that miss in `dir2`. Now you should repeat the search and check the vice versa case. The similar `find` call can print `dir2` files that miss in `dir1`.

Listing 5-3 shows the complete `dir-diff.sh` script for directory comparison.

Listing 5-3. The script for directory comparison

```
1 #!/bin/bash
2
3 cd dir1
4 find . -type f -exec test ! -e ../dir2/{} \; -a -exec echo {} \;
5
6 cd ../dir2
7 find . -type f -exec test ! -e ../dir1/{} \; -a -exec echo {} \;
```



You wrote the directory comparison script for training purposes. Please do not use it for real tasks. The `diff` GNU utility should be used instead.

Exercise 3-6. The `case` statement

Let's write the script for switching between two Bash configurations. It will create a symbolic link for the `~/ .bashrc` file.



You cannot create a symbolic link in a Unix environment running on Windows.

The `ln` utility creates a link. It does so when you launch it on Linux or macOS. The utility copies the file or directory instead of creating a link on Windows.

Symbolic links are useful when you want to access a file or directory from the specific location of the file system. Suppose that you have a link to the file. You open the link and make some changes. Then, OS applies your changes to the file that the link points to. A link to a directory works the same way. OS applies your changes to the target directory.

The algorithm for switching between Bash configurations looks like this:

1. Remove the existing symbolic link or file `~/ .bashrc`.
2. Check the command-line option passed to the script.
3. Depending on the option, create the `~/ .bashrc` link to the `~/ .bashrc-home` or `~/ .bashrc-work` file.

Let's implement this algorithm using the case statement. Listing 5-4 shows the result.

Listing 5-4. The script for switching Bash configurations

```

1  #!/bin/bash
2
3  file="$1"
4
5  rm ~/.bashrc
6
7  case "$file" in
8      "h")
9          ln -s ~/.bashrc-home ~/.bashrc
10         ;;
11
12     "w")
13         ln -s ~/.bashrc-work ~/.bashrc
14         ;;
15
16     *)
17         echo "Invalid option"
18         ;;
19 esac

```

There are two calls of the `ln` utility in this script. They differ by the target filename. This similarity gives us a hint that you can replace the case statement with an associative array. Then you get the script from Listing 5-5.

Listing 5-5. The script for switching Bash configurations

```

1  #!/bin/bash
2
3  option="$1"
4
5  declare -A files=(
6      ["h"]="~/.bashrc-home"
7      ["w"]="~/.bashrc-work")
8
9  if [[ -z "$option" || ! -v files["$option"] ]]

```

```

10 then
11     echo "Invalid option"
12     exit 1
13 fi
14
15 rm ~/.bashrc
16
17 ln -s "${files["$option"]}" ~/.bashrc

```

Consider the last line of the script. In our case, double quotes are not necessary when you insert the array element. However, they prevent a potential error of processing filenames with spaces.

Exercise 3-7. Arithmetic operations with numbers in the two's complement representation

Here are the results of adding single-byte integers:

$$* 79 + (-46) = 0100\ 1111 + 1101\ 0010 = 1\ 0010\ 0001 \rightarrow 0010\ 0000 = 33$$

$$* -97 + 96 = 1001\ 1111 + 0110\ 0000 = 1111\ 1111 \rightarrow 1111\ 1110 \rightarrow 1000\ 0001 = -1$$

Here are the result of adding two-byte integers:

$$* 12868 + (-1219) = 0011\ 0010\ 0100\ 0100 + 1111\ 1011\ 0011\ 1101 = \\ 1\ 0010\ 1101\ 1000\ 0001 \rightarrow 0010\ 1101\ 1000\ 0001 = 11649$$

Use the [online calculator](https://planetcalc.com/747/)⁴⁹⁷ to check your conversion of integers to the two's complement.

Exercise 3-8. Modulo and the remainder of a division

$$* 1697 \% 13$$

$$q = 1697 / 13 \sim 130.5385 \sim 130$$

$$r = 1697 - 13 * 130 = 7$$

$$* 1697 \text{ modulo } 13$$

$$q = 1697 / 13 \sim 130.5385 \sim 130$$

$$r = 1697 - 13 * 130 = 7$$

$$* 772 \% -45$$

$$q = 772 / -45 \sim -17.15556 \sim -17$$

$$r = 772 - (-45) * (-17) = 7$$

⁴⁹⁷<https://planetcalc.com/747/>

```
* 772 modulo -45
q = (772 / -45) - 1 ~ -18.15556 ~ -18
r = 772 - (-45) * (-18) = -38
```

```
* -568 % 12
q = -568 / 12 ~ -47.33333 ~ -47
r = -568 - 12 * (-47) = -4
```

```
* -568 modulo 12
q = (-568 / 12) - 1 ~ -48.33333 ~ -48
r = -568 - 12 * (-48) = 8
```

```
* -5437 % -17
q = -5437 / -17 ~ 319.8235 ~ 319
r = -5437 - (-17) * 319 = -14
```

```
* -5437 modulo -17
q = -5437 / -17 ~ 319.8235 ~ 319
r = -5437 - (-17) * 319 = -14
```

You can use this [Python script](#)⁴⁹⁸ to check your calculations. Call the `getRemainder` and `getModulo` functions for your pair of numbers. Then print the results using the `print` function. Take the existing function calls in this script as examples.

Exercise 3-9. Bitwise NOT

First, let's calculate bitwise NOT for unsigned two-byte integers. You get the following results:

```
56 = 0000 0000 0011 1000
~56 = 1111 1111 1100 0111 = 65479

1018 = 0000 0011 1111 1010
~1018 = 1111 1100 0000 0101 = 64517

58362 = 1110 0011 1111 1010
~58362 = 0001 1100 0000 0101 = 7173
```

If you apply bitwise NOT for signed two-byte integers, you get other results. They look this way:

⁴⁹⁸<https://github.com/ellysh/bash-programming-from-scratch/blob/master/manuscript/resources/code/Answers/remainder-modulo.py>

56 = 0000 0000 0011 1000
 $\sim 56 = 1111 1111 1100 0111 \rightarrow 1000 0000 0011 1001 = -57$

1018 = 0000 0011 1111 1010
 $\sim 1018 = 1111 1100 0000 0101 \rightarrow 1000 0011 1111 1011 = -1019$

You cannot represent the 58362 number as a signed two-byte integer. The reason is an overflow. If you write bits of the number in a variable of this type, you get -7174. The following conversion of the 58362 number to two's complement explains it:

58362 = 1110 0011 1111 1010 \rightarrow 1001 1100 0000 0110 = -7174

Now you can apply the bitwise NOT and get the following result:

-7174 = 1110 0011 1111 1010
 $\sim(-7174) = 0001 1100 0000 0101 = 7173$

You can check the results of your calculations for the signed integers using Bash. Here are the commands for that:

```
1 $ echo $((~56))
2 -57
3 $ echo $((~1018))
4 -1019
5 $ echo $((~(-7174)))
6 7173
```

Bash does not allow you to calculate the bitwise NOT for the two-byte unsigned integer 58362. It happens because the interpreter stores this number as a signed eight-byte integer. Then the NOT operation gives you the following result:

```
1 $ echo $((~58362))
2 -58363
```

Exercise 3-10. Bitwise AND, OR and XOR

Let's calculate bitwise operations for unsigned two-byte integers. You will get the following results:

$1122 \ \& \ 908 = 0000 \ 0100 \ 0110 \ 0010 \ \& \ 0000 \ 0011 \ 1000 \ 1100 = 0000 \ 0000 \ 000 \ 0000 = 0$

$1122 \ | \ 908 = 0000 \ 0100 \ 0110 \ 0010 \ | \ 0000 \ 0011 \ 1000 \ 1100 = 0000 \ 0111 \ 1110 \ 1110 = 2030$

$1122 \ ^ \ 908 = 0000 \ 0100 \ 0110 \ 0010 \ ^ \ 0000 \ 0011 \ 1000 \ 1100 = 0000 \ 0111 \ 1110 \ 1110 = 2030$

$49608 \ \& \ 33036 = 1100 \ 0001 \ 1100 \ 1000 \ \& \ 1000 \ 0001 \ 0000 \ 1100 = 1000 \ 0001 \ 0000 \ 1000 = 33032$

$49608 \ | \ 33036 = 1100 \ 0001 \ 1100 \ 1000 \ | \ 1000 \ 0001 \ 0000 \ 1100 = 1100 \ 0001 \ 1100 \ 1100 = 49612$

$49608 \ ^ \ 33036 = 1100 \ 0001 \ 1100 \ 1000 \ ^ \ 1000 \ 0001 \ 0000 \ 1100 = 0100 \ 0000 \ 1100 \ 0100 = 16580$

If the integers are signed, the bitwise operations give the same results for the first pair of numbers 1122 and 908.

Let's calculate the bitwise operations for signed two-byte integers 49608 and 33036. First, you should represent these numbers in the two's complement this way:

$49608 = 1100 \ 0001 \ 1100 \ 1000 \ \rightarrow \ 1011 \ 1110 \ 0011 \ 1000 = -15928$

$33036 = 1000 \ 0001 \ 0000 \ 1100 \ \rightarrow \ 1111 \ 1110 \ 1111 \ 0100 = -32500$

The integer overflow happened here. So, you get negative numbers instead of positive ones.

Then you do bitwise operations:

$-15928 \ \& \ -32500 = 1100 \ 0001 \ 1100 \ 1000 \ \& \ 1000 \ 0001 \ 0000 \ 1100 = 1000 \ 0001 \ 0000 \ 1000 \ \rightarrow \ 1111 \ 1110 \ 1111 \ 1000 = -32504$

$-15928 \ | \ -32500 = 1100 \ 0001 \ 1100 \ 1000 \ | \ 1000 \ 0001 \ 0000 \ 1100 = 1100 \ 0001 \ 1100 \ 1100 \ \rightarrow \ 1011 \ 1110 \ 0011 \ 0100 = -15924$

$-15928 \ ^ \ -32500 = 1100 \ 0001 \ 1100 \ 1000 \ ^ \ 1000 \ 0001 \ 0000 \ 1100 = 0100 \ 0000 \ 1100 \ 0100 = 16580$

If you need to check your results, Bash can help you. Here are the commands:

```

1 $ echo $((1122 & 908))
2 0
3 $ echo $((1122 | 908))
4 2030
5 $ echo $((1122 ^ 908))
6 2030
7
8 $ echo $((49608 & 33036))
9 33032
10 $ echo $((49608 | 33036))
11 49612
12 $ echo $((49608 ^ 33036))
13 16580
14
15 $ echo $((-15928 & -32500))
16 -32504
17 $ echo $((-15928 | -32500))
18 -15924
19 $ echo $((-15928 ^ -32500))
20 16580

```

Exercise 3-11. Bit shifts

Let's perform bit-shifts for the signed two-byte integers. Then you will get the following results:

```

* 25649 >> 3 = 0110 0100 0011 0001 >> 3 =
0110 0100 0011 0 = 0000 1100 1000 0110 = 3206

* 25649 << 2 = 0110 0100 0011 0001 << 2 =
10 0100 0011 0001 -> 1001 0000 1100 0100 = -28476

* -9154 >> 4 = 1101 1100 0011 1110 >> 4 =
1101 1100 0011 -> 1111 1101 1100 0011 = -573

* -9154 << 3 = 1101 1100 0011 1110 << 3 =
1 1100 0011 1110 -> 1110 0001 1111 0000 = -7696

```

Here are the Bash commands for checking the calculations:

```
1 $ echo $((25649 >> 3))
2 3206
3 $ echo $((25649 << 2))
4 102596
5 $ echo $((-9154 >> 4))
6 -573
7 $ echo $((-9154 << 3))
8 -73232
```

Bash results for the second and fourth cases differ from our calculations. It happens because the interpreter stores all integers in eight bytes.

The [online calculator](#)⁴⁹⁹ allows you to specify the integer type. Therefore, it is a more reliable tool for checking bit shifts than Bash.

Exercise 3-12. Loop Constructs

The player has seven tries to guess a number. The same algorithm handles each try. Therefore, the loop with seven iterations can handle the user input.

Here is the algorithm for processing one player action:

1. Read the input with the `read` command.
2. Compare the entered number with the number chosen by the script.
3. If the player makes a mistake, print a hint and go to step 1.
4. If the player guessed the number, finish the script.

The script can pick a random number using the `RANDOM` Bash variable. When you read it, you get a random value from 0 to 32767. The next read gives you another value.

You can convert the `RANDOM` value to the range from 1 to 100. Here is the algorithm for doing that:

1. Divide the value from the `RANDOM` variable by 100 this way. It gives you a random number in the 0 to 99 range.
2. Add one to the result. It gives you a number in the 1 to 100 range.

The following command writes a random number of the 1 to 100 range to the `number` variable:

```
number=$((RANDOM % 100 + 1))
```

Listing 5-6 shows the script that implements the game's algorithm.

⁴⁹⁹<https://onlinetoolz.net/bitshift>

Listing 5-6. The script for playing More or Fewer

```
1  #!/bin/bash
2
3  number=$((RANDOM % 100 + 1))
4
5  for i in {1..7}
6  do
7      echo "Enter the number:"
8
9      read input
10
11     if (( input < number))
12     then
13         echo "The number $input is less"
14     elif (( number < input))
15     then
16         echo "The number $input is greater"
17     else
18         echo "You guessed the number"
19         exit 0
20     fi
21 done
22
23 echo "You didn't guess the number"
```

You need to apply the **binary search**⁵⁰⁰ for winning this game. The idea behind this algorithm is to divide an array of numbers into halves. Let's look at how to use the binary search in the "More or Fewer" game.

Once the game starts, you guess a number in the 1 to 100 range. The middle of this range is 50. You should enter this value on the first step.

The script gives you the first hint. Suppose the script prints that 50 is less than the required number. This means that you should search for it in the 50-100 range. Now you enter the middle of this range, i.e. the number 75.

The script says that 75 is less than the required number. Then you truncate the searching range again and get 75-100. The middle of this range equals 87.5. You can round it up or down. It doesn't matter. Round it down and type number 87.

If the number is still wrong, keep dividing the searching range in half. Then seven steps are enough to find the required number.

⁵⁰⁰https://en.wikipedia.org/wiki/Binary_search_algorithm

Exercise 3-13. Functions

We have considered the `code_to_error` function in the section “Using Functions in Scripts”. Your task is to extend this function for supporting two languages. The straightforward solution is to split it. Then each variant of the function corresponds to a specific language.

In the first step, let’s combine the code of the `print_error` and `code_to_error` functions into one file. You will get the script from Listing 5-7 this way.

Listing 5-7. The script for printing error messages

```
1  #!/bin/bash
2
3  code_to_error()
4  {
5      case $1 in
6          1)
7              echo "File not found:"
8              ;;
9          2)
10             echo "Permission to read the file denied:"
11             ;;
12     esac
13 }
14
15 print_error()
16 {
17     echo "$(code_to_error $1) $2" >> debug.log
18 }
19
20 print_error 1 "readme.txt"
```

The function `code_to_error` prints messages in English. You can rename it to `code_to_error_en`. Then the language of the messages becomes clear from the function name.

The next step is adding the `code_to_error_de` function to your script. It prints the message in German according to the received error code. The function looks like this:

```
1 code_to_error_de()
2 {
3     case $1 in
4         1)
5             echo "Der Datei wurde nicht gefunden:"
6             ;;
7         2)
8             echo "Berechtigung zum Lesen der Datei verweigert:"
9             ;;
10    esac
11 }
```

Now you need to modify the `print_error` function. It should choose `code_to_error_en` or `code_to_error_de` to call. The regional settings of a user can help you with this choice. The environment variable `LANG` stores these settings. If its value matches the “`de_DE*`” pattern, you should call the `code_to_error_de` function. Otherwise, it should be `code_to_error_en` call.

Listing 5-8 shows the complete code of the script.

Listing 5-8. The script for printing error messages

```
1 #!/bin/bash
2
3 code_to_error_de()
4 {
5     case $1 in
6         1)
7             echo "Der Datei wurde nicht gefunden:"
8             ;;
9         2)
10            echo "Berechtigung zum Lesen der Datei verweigert:"
11            ;;
12    esac
13 }
14
15 code_to_error_en()
16 {
17     case $1 in
18         1)
19             echo "File not found:"
20             ;;
21         2)
22             echo "Permission to read the file denied:"
23             ;;
```

```
24  esac
25  }
26
27  print_error()
28  {
29    if [[ "$LANG" == de_DE* ]]
30    then
31      echo "$(code_to_error_de $1) $2" >> debug.log
32    else
33      echo "$(code_to_error_en $1) $2" >> debug.log
34    fi
35  }
36
37  print_error 1 "readme.txt"
```

You can replace the `if` statement in the `print_error` function with `case`. Then you get the following code:

```
1  print_error()
2  {
3    case $LANG in
4      de_DE*)
5        echo "$(code_to_error_de $1) $2" >> debug.log
6        ;;
7      en_US*)
8        echo "$(code_to_error_en $1) $2" >> debug.log
9        ;;
10     *)
11       echo "$(code_to_error_en $1) $2" >> debug.log
12       ;;
13    esac
14  }
```

The `case` statement is convenient if you need to support more than two languages.

There is code duplication in the `print_error` function. You call the `echo` command in each block of the `case` statement. The only difference between the blocks is the function for converting an error code into text. You can introduce the `func` variable to get rid of the code duplication. This variable stores the function name to call. Here is an example of how to use the variable:


```
1 print_error()
2 {
3     case $LANG in
4         de_DE)
5             local func="code_to_error_de"
6             ;;
7         en_US)
8             local func="code_to_error_en"
9             ;;
10        *)
11            local func="code_to_error_en"
12            ;;
13    esac
14
15    echo "$($func $1) $2" >> debug.log
16 }
```

There is another option to solve the code duplication problem. The first step is to replace the case statements in the functions `code_to_error_de` and `code_to_error_en` with indexed arrays. You get the following code this way:

```
1 code_to_error_de()
2 {
3     declare -a messages
4
5     messages[1]="Der Datei wurde nicht gefunden:"
6     messages[2]="Berechtigung zum Lesen der Datei verweigert:"
7
8     echo "${messages[$1]}"
9 }
10
11 code_to_error_en()
12 {
13     declare -a messages
14
15     messages[1]="File not found:"
16     messages[2]="Permission to read the file denied:"
17
18     echo "${messages[$1]}"
19 }
```

The second step is moving the code of `code_to_error_de` and `code_to_error_en` functions into `print_error`. For doing that, you need to combine messages in all languages into one associative

array. The array keys are combinations of the LANGUAGE value and the error code. Listing 5-9 shows the modified `print_error` function.

Listing 5-9. The script for printing error messages

```
1  #!/bin/bash
2
3  print_error()
4  {
5      declare -A messages
6
7      messages["de_DE",1]="Der Datei wurde nicht gefunden:"
8      messages["de_DE",2]="Berechtigung zum Lesen der Datei verweigert:"
9
10     messages["en_US",1]="File not found:"
11     messages["en_US",2]="Permission to read the file denied:"
12
13     echo "${messages[$LANGUAGE,$1]} $2" >> debug.log
14 }
15
16 print_error 1 "readme.txt"
```

Exercise 3-14. Variable scope

When you launch the script from Listing 3-37, it prints the following text:

```
1  main1: var =
2  foo1: var = foo_value
3  bar1: var = foo_value
4  bar2: var = bar_value
5  foo2: var = bar_value
6  main2: var =
```

Let's start with the output of "main1" and "main2" strings. The `var` variable is declared in the `foo` function. It has the `local` attribute. The attribute makes the variable available in the `foo` and `bar` functions only. Hence, Bash deduces that `var` is undefined before and after calling the `foo` function. The undefined variable has an empty value in Bash.

When the script prints the `var` variable at the beginning of the `foo` function, it equals "foo_value". This output happens right after the `var` declaration.

The next output happens in the `bar` function. There, the first `echo` call prints the "foo_value" value. It happens because the body of the `bar` function is also the scope of `var` declared in `foo`.

The script assigns the new "bar_value" value to the `var` variable in the `bar` function. Note that this is not a declaration of the new global `var` variable. This is the overwriting of the existing local variable. Therefore, you get the "bar_value" strings in both "bar2" and "foo2" outputs.

Helpful Links

General Information

- [A history of the electronic computers](#)⁵⁰¹.
- [A history of Apple's operating systems](#)⁵⁰².
- [The article about type systems in programming languages](#)⁵⁰³.

Bash

- [“Bash Guide for Beginners”](#)⁵⁰⁴ by Machtelt Garrels.
- [“Advanced Bash-Scripting Guide”](#)⁵⁰⁵ by Mendel Cooper.
- [Bash Shell Cheatsheet](#)⁵⁰⁶.
- [Bash Reference Manual](#)⁵⁰⁷.
- [The Bash Hackers Wiki](#)⁵⁰⁸.
- [Bash Pitfalls](#)⁵⁰⁹.
- [Useless Use of Cat Award](#)⁵¹⁰.
- [Regular expressions in grep with examples](#)⁵¹¹.
- [Bash Best Practices](#)⁵¹².
- [Bash Style Guide](#)⁵¹³ based on [mywiki.woledge.org](#)⁵¹⁴.
- [Shell Style Guide](#)⁵¹⁵ by Google.
- [Online analyzer for Bash commands](#)⁵¹⁶.
- [The guide for Bash scripts localization](#)⁵¹⁷.

⁵⁰¹<https://technichistory.com/2017/08/29/the-electronic-computers-part-1-prologue>

⁵⁰²<http://web.archive.org/web/20180702193510/http://kernelthread.com/publications/appleoshistory/>

⁵⁰³https://www.destroyallsoftware.com/compendium/types?share_key=baf6b67369843fa2

⁵⁰⁴<https://tldp.org/LDP/Bash-Beginners-Guide/html/>

⁵⁰⁵<http://tldp.org/LDP/abs/html>

⁵⁰⁶<https://github.com/NisreenFarhoud/Bash-Cheatsheet>

⁵⁰⁷https://www.gnu.org/software/bash/manual/html_node/index.html#SEC_Contents

⁵⁰⁸<https://wiki.bash-hackers.org>

⁵⁰⁹<http://mywiki.woledge.org/BashPitfalls>

⁵¹⁰<http://www.smallo.ruhr.de/award.html>

⁵¹¹<https://www.cyberciti.biz/faq/grep-regular-expressions/>

⁵¹²<http://mywiki.woledge.org/BashGuide/Practices>

⁵¹³<https://github.com/bahamas10/bash-style-guide>

⁵¹⁴<http://mywiki.woledge.org>

⁵¹⁵<https://google.github.io/styleguide/shellguide.html>

⁵¹⁶<https://explainshell.com/#>

⁵¹⁷<https://mywiki.woledge.org/BashFAQ/098>

Unix Environment

- [“The Art of Unix Programming”⁵¹⁸](#) book by Eric S. Raymond.

⁵¹⁸<http://www.catb.org/~esr/writings/taoup>