

LFD480

Fundamentals of Learning Rust

Version 1.73



Version 1.73

© Copyright The Linux Foundation 2023 All rights reserved.

© Copyright The Linux Foundation 2023 All rights reserved.

The training materials provided or developed by The Linux Foundation in connection with the training services are protected by copyright and other intellectual property rights.

Open source code incorporated herein may have other copyright holders and is used pursuant to the applicable open source license.

The training materials are provided for individual use by participants in the form in which they are provided. They may not be copied, modified, distributed to non-participants or used to provide training to others without the prior written consent of The Linux Foundation.

No part of this publication may be reproduced, photocopied, stored on a retrieval system, or transmitted without express prior written consent.

Published by:

the **Linux Foundation**

<https://www.linuxfoundation.org>

No representations or warranties are made with respect to the contents or use of this material, and any express or implied warranties of merchantability or fitness for any particular purpose or specifically disclaimed.

Although third-party application software packages may be referenced herein, this is for demonstration purposes only and shall not constitute an endorsement of any of these software applications.

Linux is a registered trademark of Linus Torvalds. Other trademarks within this course material are the property of their respective owners.

If there are any questions about proper and fair use of the material herein, please go to <https://trainingsupport.linuxfoundation.org>.

Nondisclosure of Confidential Information

“Confidential Information” shall not include any of the following, even if marked confidential or proprietary: (a) information that relates to the code base of any open source or open standards project (collectively, “Open Project”), including any existing or future contribution thereto; (b) information generally relating or pertaining to the formation or operation of any Open Project; or (c) information relating to general business matters involving any Open Project.

This course does not include confidential information, nor should any confidential information be divulged in class.

Contents

1	Introduction	1
1.1	The Linux Foundation	2
1.2	The Linux Foundation Training	4
1.3	The Linux Foundation Certifications	8
1.4	The Linux Foundation Digital Badges	11
1.5	Laboratory Exercises, Solutions and Resources	12
1.6	Things Change in Linux and Open Source Projects	14
1.7	Distribution Details	15
1.8	Labs	22
2	Preliminaries	23
3	Introduction to Rust	27
4	My 1st Rust program	35
5	Program flow	39
6	Complex data types	43
7	Functions in Rust	47
8	Error handling in Rust	49
9	Testing in Rust	55
10	Debugging in Rust	59
11	Object Oriented Programming in Rust	61
12	Closures	65
13	Iterators	67
14	Lifetimes	71
15	Crates	73
16	OS functions	79
17	Benchmarking and Profiling Rust programs	85
18	Smart Pointers	89
19	Concurrency in Rust	101
20	Using Rust in containers	111
21	Cross compiling in Rust	113

22 Advanced Topics	115
23 Closing and Evaluation Survey	123
23.1 Evaluation Survey	124
Appendices	127

**Please Note**

** These sections may be considered in part or in whole as optional. They contain either background reference material, specialized topics, or advanced subjects. The instructor may choose to cover or not cover them depending on classroom experience and time constraints.

List of Figures

23.1	Course Survey	124
------	---------------	-----

List of Tables

12.1	Cities	65
------	------------------	----

Chapter 1

Introduction



1.1	The Linux Foundation	2
1.2	The Linux Foundation Training	4
1.3	The Linux Foundation Certifications	8
1.4	The Linux Foundation Digital Badges	11
1.5	Laboratory Exercises, Solutions and Resources	12
1.6	Things Change in Linux and Open Source Projects	14
1.7	Distribution Details	15
1.8	Labs	22

1.1 The Linux Foundation

What is The Linux Foundation?

- A non-profit consortium, dedicated to fostering the growth of:
 - **Linux**
 - Many other **Open Source Software (OSS)** projects and communities
- Supports the creation of sustainable **OSS** ecosystems by providing:
 - Financial and intellectual resources and services
 - Training
 - Events
- Originally founded to protect, support and improve **Linux** development and sponsors the work of **Linux** creator Linus Torvalds
- Supported by leading technology companies and developers in a neutral collaborative environment
- See <https://linuxfoundation.org>.

The Linux Foundation provides a neutral, trusted hub for developers to code, manage, and scale open technology projects. Founded in 2000, **The Linux Foundation** is supported by more than 1,000 members and is the world's leading home for collaboration on open source software, open standards, open data and open hardware. **The Linux Foundation's** methodology focuses on leveraging best practices and addressing the needs of contributors, users and solution providers to create sustainable models for open collaboration.

The Linux Foundation hosts **Linux**, the world's largest and most pervasive open source software project in history. It is also home to **Linux** creator Linus Torvalds and lead maintainer Greg Kroah-Hartman. The success of **Linux** has catalyzed growth in the open source community, demonstrating the commercial efficacy of open source and inspiring countless new projects across all industries and levels of the technology stack.

As a result, **The Linux Foundation** today hosts far more than **Linux**; it is the umbrella for many critical open source projects that power corporations today, spanning virtually all industry sectors. Some of the technologies we focus on include big data and analytics, networking, embedded systems and IoT, web tools, cloud computing, edge computing, automotive, security, blockchain, and many more.

The Linux Foundation Events

This is only a very partial list of **The Linux Foundation** events. Some are held in multiple locations yearly, such as North America, Europe and Asia.

- Open Source Summit
- Embedded Linux Conference North
- Open Networking & Edge Summit
- KubeCon + CloudNativeCon
- Automotive Linux Summit
- KVM Forum
- Linux Storage Filesystem and Memory Management Summit
- Linux Security Summit
- Linux Kernel Maintainer Summit
- The Linux Foundation Member Summit
- Open Compliance Summit
- And many more.

Over 85,000 open source technologists and leaders worldwide gather at **The Linux Foundation** events annually to share ideas, learn and collaborate. **The Linux Foundation** events are the meeting place of choice for open source maintainers, developers, architects, infrastructure managers, and sysadmins and technologists leading open source program offices, and other critical leadership functions.

These events are the best place to gain visibility within the open source community quickly and advance open source development work by forming connections with the people evaluating and creating the next generation of technology. They provide a forum to share and gain knowledge, help organizations identify software trends early to inform future technology investments, connect employers with talent, and showcase technologies and services to influential open source professionals, media, and analysts around the globe.

1.2 The Linux Foundation Training

Training Venues

The Linux Foundation offers several types of training:

- Physical Classroom (often On-Site)
- Online Virtual Classroom
- Individual Self-Paced E-learning over the Internet
- Events-Based

The Linux Foundation's training is for the community, by the community, and features instructors and content straight from the leaders of the developer community.

Attendees receive training that is operating system and/or **Linux** distribution-flexible, technically advanced and created with the actual leaders of the development community themselves. **The Linux Foundation** courses give attendees the broad, foundational knowledge and networking needed to thrive in their careers today. With either online or in person training, **The Linux Foundation** classes can keep you or your developers ahead of the curve on the essentials of open source administration and development.

Training Offerings

Our current course offerings include:

- Linux Programming & Development Training
- Enterprise IT & Linux System Administration Courses
- Open Source Compliance Courses

For more information see <https://training.linuxfoundation.org>

The Linux Foundation also offers a wide range of free **MOOCs** (**M**assively **O**pen **O**nline **C**ourses) offered through **edX** at <https://edx.org>. These cover basic as well as rather advanced topics associated with open source.

To find them at the **edX** website, search on "The Linux Foundation"

Copyright

- The contents of this course and all its related materials, including hand-outs, are © Copyright The Linux Foundation 2023 All rights reserved.



Do not copy or distribute

No part of this publication may be stored in a retrieval system, transmitted or reproduced in any way, including, but not limited to, photocopy, photograph, magnetic, electronic or other record, without the prior written permission of The Linux Foundation.

This training, including all material provided herein, is supplied without any guarantees from **The Linux Foundation**. **The Linux Foundation** assumes no liability for damages or legal action arising from the use or misuse of contents or details contained herein.

If you believe **The Linux Foundation** materials are being used, copied, or otherwise improperly distributed please go to <https://trainingsupport.linuxfoundation.org>.

About Confidential Information



Nondisclosure of Confidential Information

“Confidential Information” shall not include any of the following, even if marked confidential or proprietary: (a) information that relates to the code base of any open source or open standards project (collectively, “Open Project”), including any existing or future contribution thereto; (b) information generally relating or pertaining to the formation or operation of any Open Project; or (c) information relating to general business matters involving any Open Project.

This course does not include confidential information, nor should any confidential information be divulged in class.

1.3 The Linux Foundation Certifications

The Linux Foundation Certification Exams

- **The Linux Foundation** offers comprehensive **Certification Programs**.
- Full details about this program can be found at <https://training.linuxfoundation.org/certification>. This information includes a thorough description of the **Domains** and **Competencies** covered by each exam.
- Besides the **LFCS** (**L**inux **F**oundation **C**ertified **S**ysadmin) exam, **The Linux Foundation** provides certification exams for many other open source projects. The list is constantly expanding so please see <https://training.linuxfoundation.org/certification> for the current list.
- For additional information, including course descriptions, technical requirements and other logistics, see <https://training.linuxfoundation.org>.

Certification/Training Firewall

- **The Linux Foundation** has two separate training divisions:
 - Certification
 - Course Delivery
- These are separated by a **firewall**:
 - Enables third party organizations to develop and deliver **LF** certification preparation classes
 - Prevents using ***secret sauce*** in **LF** courses
 - Prevents ***teaching the test*** in **LF** courses
- Instructors (including today) are guided entirely by publicly available information

The curriculum development and maintenance division of **The Linux Foundation** training department has no direct role in developing, administering, or grading certification exams.

Enforcing this self-imposed **firewall** ensures that independent organizations and companies can develop third party training material, geared to helping test takers pass their certification exams.

Furthermore, it ensures that there are no secret “tips” (or secrets in general) that one needs to be familiar with to succeed.

It also permits **The Linux Foundation** to develop a very robust set of courses that do far more than ***teach the test***, but rather equip attendees with a broad knowledge of many areas they may be required to master to have a successful career in **Linux** system administration.

Preparation Resources

- Before doing anything else, download:
<https://training.linuxfoundation.org/download-free-certification-prep-guide>
- Sections on:
 - Domains and Competencies
 - Free Training Resources
 - Paid Training Resources
 - Taking the Exam
- Also get the **Candidate Handbook** at:
https://training.linuxfoundation.org/go/candidate_handbook
- Also read exam-specific information at:
<https://training.linuxfoundation.org/certification/lfcs>

We will discuss some (but not all!) of the issues in the documents quoted above, all of which can also be easily be found through links at **The Linux Foundation** web page at <https://training.linuxfoundation.org/certification>.

Topics covered include:

- What material is covered in the exam
- Candidate Requirements, including identification, authentication, eligibility, accessibility, confidentiality requirements.
- Exam registration and fees, refund policy, etc.
- **Linux** distribution choices
- Checking your hardware and software environment for suitability for the exam
- How to start and complete the exam
- Exam Interface and format
- Exam results, scoring and re-scoring requests, and retake policy
- Certificate issuance, verification, expiration, renewal etc.
- Accessing tech support

1.4 The Linux Foundation Digital Badges

The Linux Foundation Digital Badges

- Digital Badges communicate abilities and credentials
- Can be used in email signatures, digital resumes, social media sites (**LinkedIn**, **Facebook**, **Twitter**, etc)
- Contain verified metadata describing qualifications and process that earned them.
- Available to students who successfully complete **The Linux Foundation** courses and certifications
- Details at <https://training.linuxfoundation.org/badges/>

The Linux Foundation is committed to providing you with the tools necessary to achieve your professional goals. We understand communicating your abilities and credentials can be challenging. For this reason, we have partnered with **Credly** to provide you with a digital version of your credentials through its **Acclaim** platform. These badges can be used in email signatures or digital resumes, as well as on social media sites such as **LinkedIn**, **Facebook**, and **Twitter**. This digital image contains verified metadata that describes your qualifications and the process required to earn them.

- Badges are shareable via any digital platform: social media, embedded in your résumé, email signature, or the web.
- All badges can be verified by anyone simply by clicking on the badge.
- Badges will be issued to everyone who passes one of our certification exams as well as those who purchase training courses directly from The Linux Foundation or an authorized training partner.
- Badges will also be issued to those who contributed on our exam or course development teams

How it Works

1. You will receive an email notifying you to claim your badge at our partner **Credly's Acclaim** platform website.
2. Click the link in that email.
3. Create an account on the **Acclaim** platform site and confirm your email.
4. Claim your badge.
5. Start sharing.

1.5 Laboratory Exercises, Solutions and Resources

Labs

- Hands-on exercises provided at the end of each session.
- Can be done on either virtual machines or bare-metal
 - Unless otherwise specified
- Some exercises marked as optional
- Solutions available at the end of each exercise.

Obtaining Course Solutions and Resources

- If this course has such material, lab exercise solutions, suggestions and other resources may be downloaded from: <https://cm.lf.training/LFD480>

- If you do not have a browser available you can obtain with:

```
$ wget --user=LFtraining --password=<ask instructor> \
  https://cm.lf.training/LFD480/LFD480_V1.73.SOLUTIONS.tar.xz
```

and similarly for the RESOURCES file.

```
$ wget --user=LFtraining --password=<ask instructor> \
  https://cm.lf.training/LFD480/LFD480_V1.73.RESOURCES.tar
```

- Any errata, updated solutions, etc. will also be posted on that site
- These files may be unpacked with:

```
$ tar xvf LFD480_V1.73.SOLUTIONS.tar.xz
$ tar xvf LFD480_V1.73.RESOURCES.tar
```

You will see subdirectories in the both the **SOLUTIONS** and **RESOURCES** directories such as `s_01`, `s_10`, `s_13` for each section that has files you either need or may find of interest. We may refer to these files in future sections.

Depending on the course, there may not be a **RESOURCES** file, which is intended for binary files such as archives and videos.

Binary files, such as source tarballs for various labs, will be resourced with instructions as needed.

If the course has demonstration videos included, these are mostly intended for supplementary study outside of lecture time. However, the instruction might elect to show some of them to get a hopefully clean demonstration, or give an example of practices on alternative **Linux** distributions.

1.6 Things Change in Linux and Open Source Projects

Open Source Software Changes Often

- **The Linux Foundation** courses are constantly updated to synchronize with upstream versions of projects such as **Kubernetes** or the **Linux** kernel.
- There will be unavoidable breakage from time to time and deprecated features will disappear.
- We try to minimize problems, but often we have to respond to upstream changes we cannot control.
- The alternative is to have stale material which is lacking important features.
- Working with these shifting sources and targets is part of the *fun* of working with open source software!

The Linux Foundation courses are constantly updated, some of them with every new version release of projects they depend on, such as the **Linux** kernel or **Kubernetes**.

For example, no matter how hard we have worked to stay current, **Linux** is constantly evolving, both at the technical level (including kernel features) and at the distribution and interface level.

So please keep in mind we have tried to be as up to date as possible at the time this class was released, but there will be changes and new features we have not discussed. It is unavoidable.

As a result of this churn, no matter how hard we try there are inevitably features in the material that might suffer from one of the following:

- They will stop working on a **Linux** distribution we support.
- A change in **API** will break a laboratory exercise with a particular kernel or upstream library or product.
- Deprecated features we have been supporting will finally disappear.

There are students who just expect everything in the class to always work all the time. While we strive to achieve this, we know:

- It is an unrealistic expectation.
- Figuring out the problems and how to solve this is an active part of the class, part of the “adventure” of working with cutting edge open source projects. Moving targets are always harder to hit.

If we tried to avoid all such problems, we would have material that goes stale quickly and does not keep up with changes. We rely on our instructors and students to notice any such problems and inform courseware designers so they can incorporate the appropriate remedies in the next release.

This is the true open source way of doing things.

1.7 Distribution Details

Software Environment

- Class material is designed for multiple environments
- Focuses on three current major **Linux** distribution families:
 - **Red Hat / Fedora**
 - **OpenSUSE / SUSE**
 - **Debian**



Please Note

The material in this section is aimed at courses which either require or strongly recommend they be run only on a **Linux**-based operating system. Some courses can be run using any environment that has a browser and perhaps an **ssh** (secure shell) utility. However, while in that case you can skip over this material, it is still recommended you absorb its information.

The material produced by **The Linux Foundation** is distribution-flexible. This means that technical explanations, labs and procedures should work on most modern distributions and we do not promote products sold by any specific vendor (although we may mention them for specific scenarios).

In practice, most of our material is written with the three main **Linux** distribution families in mind: **Red Hat / Fedora**, **OpenSUSE / SUSE** and **Debian**. Distributions used by our students tend to be one of those three alternatives, or a product that's derived from them.

Which Distribution to Choose

- Several factors to consider:
 - Has your employer already standardized?
 - Do you want to learn more?
 - Do you want to certify?
- You are encouraged to experiment with more than one distribution

You should ask yourself several questions when choosing a new distribution. While there are many reasons that may force you to focus on one **Linux** distribution versus another, we encourage you to gain experience on all of them. You will quickly notice that technical differences are mainly about package management systems, software versions and file locations. Once you get a grasp of those differences it becomes relatively painless to switch from one **Linux** distribution to another.

Some tools and utilities have vendor supplied front-ends, especially for more particular or complex reporting. The steps included in the text may need to be modified to run on a different platform.

Red Hat / Fedora Family

- Current material based upon the latest releases of **Red Hat Enterprise Linux (RHEL)**.
- Supports **x86**, **x86-64**, **Itanium**, **PowerPC** and **IBM System Z**
- RPM-based, uses **dnf** to install and update
- Long release cycle; targets enterprise server environments
- Upstream for **CentOS** and **Oracle Linux**
- Downstream for **CentOS Stream**



CentOS Stream

CentOS Stream is used for demos and labs because it is available at no cost.

Fedora is the community distribution that forms the basis of **Red Hat Enterprise Linux**, **CentOS**, and **Oracle Linux**. **Fedora** contains significantly more software than **Red Hat**'s enterprise version. One reason for this is a diverse community is involved in building **Fedora**; it is not just one company.

The **Fedora** community produces new versions every six months or so. For this reason, we decided to standardize the **Red Hat / Fedora** part of the course material on the latest version of **CentOS/CentOS Stream**, which provides much longer release cycles. Once installed, **CentOS Stream** is also very close to to **Red Hat Enterprise Linux (RHEL)**, which is the most popular **Linux** distribution in enterprise environments.



CentOS and CentOS Stream

- **CentOS** historically has been basically a copy of **RHEL** with some time delay after updates.
- **CentOS Stream** gets updates **before RHEL**, but otherwise is quite close to it. Thus newer features will be absorbed quicker.
- **Red Hat** ended support for **CentOS 8** at the end of 2021. Thus, this course is tested with the **CentOS Stream** distribution; any variance from **CentOS 8** or **RHEL 8** will be minor and should not even be noticeable.

OpenSUSE Family

- Current material based upon the latest release of **OpenSUSE** and should work well with later versions.
- RPM-based, uses **zypper** to install and update.
- **YaST** available for administration purposes
- **x86** and **x86-64**
- Upstream for **SUSE Linux Enterprise Server (SLES)**



Please Note

OpenSUSE is used for demos and labs because it is available at no cost.

The relationship between **OpenSUSE** and **SUSE Linux Enterprise Server** is similar to the one we just described between **Fedora** and **Red Hat Enterprise Linux**. In this case, however, we decided to use **OpenSUSE** as the reference distribution for the **OpenSUSE** family due to the difficulty of obtaining a free version of **SUSE Linux Enterprise Server**. The two products are extremely similar and material that covers **OpenSUSE** can typically be applied to **SUSE Linux Enterprise Server** with no problem.

Debian Family

- Commonly used on both servers and desktop
- **DPKG**-based, use **apt** and front-ends for installing and updating
- Upstream for **Ubuntu**, **Linux Mint** and others
- Current material based upon the latest release of **Ubuntu** and should work well with later versions.
- **x86** and **x86-64**
 - Long Term Release (LTS)



Please Note

Ubuntu is used for demos and labs because it is available at no cost, as is **Debian**, but has a wider base with new **Linux** users.

The **Debian** distribution is the upstream for several other distributions including **Ubuntu**, **Linux Mint** and others. **Debian** is a pure open source project and focuses on a key aspect: stability. It also provides the largest and most complete software repository to its users.

Ubuntu aims at providing a good compromise between long term stability and ease of use. Since **Ubuntu** gets most of its packages from **Debian**'s unstable branch, **Ubuntu** also has access to a very large software repository. For those reasons, we decided to use **Ubuntu** as the reference **Debian**-based distribution for our lab exercises.

New Distribution Similarities

- Current trends and changes to the distributions have reduced some of the differences between the distributions.
 - **systemd**, system startup and service management
 - **journald**, manage system logs
 - **firewalld**, firewall management daemon
 - **ip**, network display and configuration tool



Please Note

Since these utilities are common across distributions, the lecture and lab information will be mostly based on them.

If your choice of distribution or release does not support these commands, please translate accordingly.

systemd is used by the most common distributions replacing the **SysVinit** and **Upstart** packages. Replaces **service** and **chkconfig** commands.

journald is a **systemd** service that collects and stores logging data. It creates and maintains structured, indexed journals based on logging information that is received from a variety of sources. Depending on the distribution text based system logs may be replaced.

firewalld provides a dynamically managed firewall with support for network/firewall zones to define the trust level of network connections or interfaces. It has support for IPv4, IPv6 firewall settings and for ethernet bridges. This replaces the **iptables** configurations.

The **ip** program is part of the **net-tools** package and is designed to be a replacement for the **ifconfig** command. The **ip** command will show or manipulate routing, network devices, routing information and tunnels.

These documents may be of some assistance translating older commands to their **systemd** counterparts:

https://fedoraproject.org/wiki/SysVinit_to_Systemd_Cheatsheet

<https://wiki.debian.org/systemd/CheatSheet>

https://en.opensuse.org/openSUSE:Cheat_sheet_13.1#Services

AWS Free Tier

- **Amazon Web Services (AWS)** offers a wide range of virtual machine products (**instances**) which remote users can access in the cloud.
- In particular, one can use their **Free Tier** account level for up to a year and the simulated hardware and software choices available may be all one needs to perform the exercises for **The Linux Foundation** training courses and gain experience with **Linux**. Or they may furnish a very educational supplement to working on local hardware, and offer opportunities to easily study more than one **Linux** distribution. Please download our guide to help you experiment with the **AWS** free tier from <https://prep.lf.training/aws.pdf>.
- Note that **AWS** will not give access to the console and will not present a Graphical interface, so you will not be able to perform tasks which require either of these facilities. Furthermore, for kernel-level courses there are some particular challenges.

1.8 Labs

Exercise 1.1: Configuring the System for `sudo`

It is very dangerous to run a **root shell** unless absolutely necessary: a single typo or other mistake can cause serious (even fatal) damage.

Thus, the sensible procedure is to configure things such that single commands may be run with superuser privilege, by using the **sudo** mechanism. With **sudo** the user only needs to know their own password and never needs to know the root password.

If you are using a distribution such as **Ubuntu**, you may not need to do this lab to get **sudo** configured properly for the course. However, you should still make sure you understand the procedure.

To check if your system is already configured to let the user account you are using run **sudo**, just do a simple command like:

```
$ sudo ls
```

You should be prompted for your user password and then the command should execute. If instead, you get an error message you need to execute the following procedure.

Launch a root shell by typing **su** and then giving the **root** password, not your user password.

On all recent **Linux** distributions you should navigate to the `/etc/sudoers.d` subdirectory and create a file, usually with the name of the user to whom root wishes to grant **sudo** access. However, this convention is not actually necessary as **sudo** will scan all files in this directory as needed. The file can simply contain:

```
student ALL=(ALL) ALL
```

if the user is `student`.

An older practice (which certainly still works) is to add such a line at the end of the file `/etc/sudoers`. It is best to do so using the **visudo** program, which is careful about making sure you use the right syntax in your edit.

You probably also need to set proper permissions on the file by typing:

```
$ sudo chmod 440 /etc/sudoers.d/student
```

(Note some **Linux** distributions may require 400 instead of 440 for the permissions.)

After you have done these steps, exit the root shell by typing `exit` and then try to do `sudo ls` again.

There are many other ways an administrator can configure **sudo**, including specifying only certain permissions for certain users, limiting searched paths etc. The `/etc/sudoers` file is very well self-documented.

However, there is one more setting we highly recommend you do, even if your system already has **sudo** configured. Most distributions establish a different path for finding executables for normal users as compared to root users. In particular the directories `/sbin` and `/usr/sbin` are not searched, since **sudo** inherits the `PATH` of the user, not the full root user.

Thus, in this course we would have to be constantly reminding you of the full path to many system administration utilities; any enhancement to security is probably not worth the extra typing and figuring out which directories these programs are in. Consequently, we suggest you add the following line to the `.bashrc` file in your home directory:

```
PATH=$PATH:/usr/sbin:/sbin
```

If you log out and then log in again (you don't have to reboot) this will be fully effective.

Chapter 2

Preliminaries



This lab provides instructions on how to set up the Rust Software Development Kit (SDK) using Rustup on Linux, MacOS, a

The tool `rustup` is supported on a variety of architectures and operating systems:

Architecture	Operating System	Notes
x86_64	Windows (10, 11)	Standard desktop editions.
x86_64	macOS (10.7 and later)	Including Intel and Apple Silicon.
x86_64	Linux	All general purpose distributions
ARMv7	Linux	Primarily for ARMv7 based devices.
ARMv8 (AArch64)	Linux	Suitable for modern ARM servers and devices.
x86_64	FreeBSD	Requires the latest stable release.
RISCV64	Linux	Suitable for development on RISC-V 64-bit architecture platforms.

This lab provides instructions for installing Rust on Linux, MacOS and Windows. Kindly execute the instructions belonging to the platform you want to install the Rust SDK on.

Prerequisites

Before beginning the installation, ensure you have a terminal application (Linux/MacOS) or command prompt/powershell (Windows) and internet access.

✎ Exercise 2.1: Installing Rust SDK on Linux

Follow these steps to install Rust on Linux:

1. Open your terminal.
2. Execute the following command to download and start the installation script:

```
SH Install Rust with rustup  
  
curl --proto '=https' --tlsv1.2 -sSf https://sh.rustup.rs | sh
```

3. Follow the on-screen instructions to complete the installation. Typically, pressing **1** to proceed with the default installation is recommended.
4. Once the installation is complete, you may need to restart your terminal or run the following command to configure your current shell:

```
SH Sourcing the cargo env  
  
source $HOME/.cargo/env
```

✎ Exercise 2.2: Installing Rust SDK on MacOS

The installation steps for MacOS are similar to Linux:

1. Open the Terminal app.
2. Run the installation script with the following command:

```
SH Installing Rust SDK with rustup  
  
curl --proto '=https' --tlsv1.2 -sSf https://sh.rustup.rs | sh
```

3. Follow the on-screen prompts to complete the installation. It's usually best to accept the default installation by pressing **1**.
4. After installation, you might need to restart your terminal or source your shell profile using:

```
SH Sourcing the cargo env  
  
source $HOME/.cargo/env
```

✎ Exercise 2.3: Installing Rust SDK on Windows


Installing Rust on Windows requires a different approach:

1. Download the Rust installer from the official Rust website: <https://rustup.rs/>

2. Run the downloaded `rustup-init.exe` file. This will start the installation process.
3. Follow the instructions in the installer, which typically involves choosing the default settings for most users.
4. To start using Rust, you may need to restart your computer or manually add the Rust installation path to your system's PATH environment variable. The installer usually takes care of this step automatically.

Exercise 2.4: Verify Rust SDK installation

To verify Rust has been installed correctly on any system, you can run the following command in your terminal (Linux/macOS) or command prompt/powershell (Windows):

 **Verify rust SDK install**

```
rustc --version
```

You should see the version of Rustc (the Rust compiler) printed to the terminal or command prompt.

You have successfully installed Rust on your system using Rustup. Whether you are using Linux, macOS, or Windows, you're now ready to start building Rust applications.

Chapter 3

Introduction to Rust



Exercise 3.1: Exercise 1: A simple rust program

We are going to create a Rust program called `greeting`. Of course, we are going to setup the environment for our code using Cargo.

```
cargo init greeting
cd greeting
```

Using your favorite editor, replace the contents of the “src/main.rs” with the following program:



greeting

```
//
// greeting
//

fn greeting() {
    println!("Hello World!");
}

fn main() {
    greeting();
}
```

Here, `//` denote comments. `greeting` is a function prefixed as such with `fn`, it's body enclosed in curly braces. The entry point of the program is the `main` function.

In the body of `greeting` function, you will see we will call `println!("Hello World!");`. Mind that this is NOT a function but a macro. It's code that generates code. We will use these macro's a lot in Rust programming and discuss how to create them ourselves later in the course. Anyhow this `println!` macro enables us to print something to `stdout` and finish it with a newline.

We already wrote that Cargo is a very useful tool while writing Rust programs. Now that we have written the code for our `greeting` program, let's consult `clippy` about any issues it has with our code, whether it is syntax errors or logical errors that could cause the program to do the things we say, but not the things we want.

Run `clippy` using the following command.

```
cargo clippy
```

output

```
Checking greeting v0.1.0 (/home/pascal/Courses/pvd816-rust/labs/source/code/greeting)
Finished dev [unoptimized + debuginfo] target(s) in 0.60s
```

In this case, Clippy has no constructive criticism on our code.

Let's look at another example.



clippy-2

```
fn sum(a: i64, b: i64) -> i64 {
    return a+b
}

fn main() {
    let a: i64 = 10 ;
    let b: i64 = 20 ;

    let s = sum(a,b) ;
    println!("Sum of {} and {} is {}", a, b, s) ;
}
```

We see some things we haven't learned yet, but that will be discussed in the modules further in the course. Key is this code will compile without any issue. Please try it.

Create the clippy-2 project with `cargo new clippy-2` and put above program text in `clippy-2/src/main.rs` Then build the program using `cargo build`.

```
cd clippy-2
cargo build
```

output

```
Compiling clippy-2 v0.1.0 (/home/pascal/Courses/pvd816-rust/labs/source/code/clippy-2)
Finished dev [unoptimized + debuginfo] target(s) in 0.58s
```

The code is correct and if you execute it with `cargo run`, it will give you the expected and correct results. However, it is NOT idiomatic rust. Kindly consult clippy with `cargo clippy` and see what constructive advise it has for us.

```
cargo clippy
```

output

```
Checking clippy-2 v0.1.0 (/home/pascal/Courses/pvd816-rust/labs/source/code/clippy-2)
warning: unneeded `return` statement
--> src/main.rs:2:5
   |
 2 |     return a+b ;
   |     ^^^^^^^^^^^^^ help: remove `return`: `a+b`
   |
= note: `#[warn(clippy::needless_return)]` on by default
= help: for further information visit
      ↪ https://rust-lang.github.io/rust-clippy/master/index.html#needless\_return

warning: `clippy-2` (bin "clippy-2") generated 1 warning
Finished dev [unoptimized + debuginfo] target(s) in 0.27s
```

What is the meaning of this? Well, when we will learn how to use functions and expressions in generic we will learn that Rust has the concept of tail expressions that can make Rust code more readable. What this means for a function is that if the last line of a function is an expression and it's not ended with a `;`, the value of that expression will be used as the return value of

the function. E.g. no need to use the `return` statement explicitly. The use of the `return` statement at the end of a function is correct and compilable Rust, however, it's not Idiomatic Rust. Your friendly Code Coach Clippy helps you in crafting Idiomatic Rust code.

You can rewrite the code yourself or ask Clippy to do that for you:

```
cargo clippy --fix
```

output

```
Checking clippy-2 v0.1.0 (/home/pascal/Courses/pyd816-rust/labs/source/code/clippy-2)
Fixed src/main.rs (1 fix)
Finished dev [unoptimized + debuginfo] target(s) in 0.20s
```

✍ Exercise 3.2: Displaying output in rust

To learn Rust or any language, it is very handy to know how to print output on the screen to see what results our programs produce. As seen in a previous lab Rust introduces the `println!` macro for this.

In this section we will learn how to work with this `println!` macro to produce orderly output from our Rust programs.

Kindly create a new project called `printing-1` with `cargo init` and put the following program text in `printing-1/src/main.rs`:



printing-1

```
fn main() {
    let a: i64 = 42 ;
    let s = "Hello Rustaceans" ;
    let h: u32 = 4207849484 ;
    let fname = "Larry" ;
    let lname = "Wall" ;

    println!() ;
    println!("a == {}",a) ;
    println!("s == {}",s) ;
    println!("x == {:X}",h) ;
    println!("x == {:x}",h) ;

    println!("{0} {1}",fname,lname)
}
```

There are so called placeholders that will be replaced with the values of the expressions supplied. These placeholders can also be numbered, after which they refer to the first, second, third etc expression specified in the `println!` macro. This macro can also do more nice things like formatting our output or printing it in a certain base like hex, octal, decimal, binary etc. For more info please refer to: <https://doc.rust-lang.org/rust-by-example/hello/print.html>

Now let's do an exercise. We are going to print a first and last name in normal and lastname, firstname order. Kindly use two methods to accomplish this.

Create a new project called `printing-2` and use the following code as basis for your `printing-2/src/main.rs` file:



printing-2

```
// printing-2

fn main() {
    let fname = "Linus" ;
    let lname = "Torvalds" ;

    // Add printing of firstname lastname and lastname, firstname in two ways
    // 1) Using {} placeholders
    // 2) Using positional placeholders {n}
    //

    println!("Change me!")
}
```

Exercise 3.3: Exploring Rust

Introduction

This lab introduces you to the basics of displaying output in Rust. You will learn to use the debug display format specifiers to control how information is presented in the console.

Debug Format Specifier

Here's a Rust array. Use the Debug Format Specifier `{:?}` to print its contents.



array debug print

```
1 fn main() {
2     let numbers = [1, 2, 3, 4, 5];
3     // Use Debug Format Specifier here
4     println!("Numbers: ");
5 }
```

Run the program and explain what you see.

Pretty Printed Debug

Given a Rust tuple, print it using the Pretty Printed Debug Format Specifier `{:#?}`.



Pretty Printed Debug

```
1 fn main() {
2     let person = ("Alice", 30, "Engineer");
3     // Use Pretty Printed Debug Format Specifier here
4     println!("Person: ");
5 }
```

Run the program and explain what you see.

Exercise 3.4: Lab: Instrumenting Code with `dbg!()`

Introduction

This lab focuses on using the `'dbg!()'` macro for debugging Rust programs. You'll instrument given code snippets to print variable values and their locations.

Using `'dbg!()'` Macro

Given the Rust code below, use the `'dbg!()'` macro to print the value of `'x'` after each operation.



language=Rust

```
1 fn main() {
2     let x = 5;
3     let y = 10;
4     // Use dbg!() to print x here
5
6     let x = x + y;
7     // Use dbg!() to print x here
8 }
```

Run the program and explain what you see.

Exercise 3.5: Exploring macro expansion with `cargo-expand`

Introduction

In this lab, you will learn how to use `'cargo-expand'` to see the expanded form of macros in your Rust programs. Understanding macro expansion can be crucial for debugging and learning how macros work under the hood.

Setup: Installing 'cargo-expand'


Before you can expand macros, you need to install the 'cargo-expand' utility. This tool requires Rust's package manager, Cargo, to be installed on your system.

Open a terminal and run the following command to install 'cargo-expand':




```
SH cargo-expand  
cargo install cargo-expand
```

1. Create a new Rust project using Cargo:



```
SH Cargo new  
cargo new expand_example  
cd expand_example
```

2. In the 'src/main.rs' file, add a simple use of the 'println!' macro:



```
Macro expansion  
1 fn main() {  
2     println!("Hello, macro!");  
3 }
```

3. Use 'cargo-expand' to see the expanded form of the 'println!' macro:



```
SH cargo expand  
cargo expand
```

4. Observe the output. Document what the println macro expands to.

5. Do the same with a #[derive(Debug)] macro:



derive expansion

```
1  #[derive(Debug)]
2  struct Course {
3      name: String,
4      lenght: u8,
5      trainer: String,
6  }
7
8  fn main() {
9      let my_course = Course {
10         trainer: "Koobenstein".to_string(),
11         lenght: 4,
12         name: "Rusty Realms".to_string(),
13     };
14     println!("{:?}", my_course)
15 }
```

Chapter 4

My 1st Rust program



In this lab, we will practice with:

- Working with Ownership and Borrowing in Rust
- Using Literal Strings and Object Strings in Rust
- Using Arrays in Rust

Exercise 4.1: Exploring Ownership with Strings

Objective

Understand the ownership and scope rules in Rust by experimenting with String types.

Tasks

1. String Creation and Move
 - Create a String variable named text1 and initialize it with some text.
 - Create another String variable text2 and assign text1 to text2.
 - Try to use text1 after assigning it to text2 and observe the compiler error.
2. Attempt to Use After Move
 - Add a println! statement to print text1 after the assignment to text2.
 - Observe the compiler error and understand why it happens.
3. Clone to Retain Ownership
 - Create a String variable named text3 and initialize it by cloning text2.
 - Print both text2 and text3.
4. Modify and Understand Mutable Ownership
 - Create a mutable String variable text4 and initialize it with some text.
 - Append some text to text4 using the .push_str() method.
 - Print text4 after modification.

Exercise 4.2: Borrowing

Introduction


In this lab, you will practice with the concept of borrowing in Rust

1. Create a new Rust project called borrowing-1 with the cargo tool

 **Cargo new**

```
cargo new borrowing-1
cd borrowing-1
```

2. In the 'src/main.rs' file, put the following content:

 **borrowing-1**

```
1 fn main() {
2     let name = String::from("Rust");
3     greet(name);
4     println!("{}", name);
5 }
6
7 fn greet(name: str) {
8     println!("Hello, {}!", name);
9 }
```

3. The output of the program should be 'Hello, Rust'. However currently it doesn't compile due to violation of ownership rules. Kindly fix the program by introducing borrowing of the name variable and compile/run the program.

1. Create a new Rust project called borrowing-2 with the cargo tool



Cargo new

```
cargo new borrowing-2
cd borrowing-2
```

2. In the 'src/main.rs' file, put the following content:



borrowing-2

```
1 fn main() {
2     let name = String::from("Rust");
3     greet(name);
4     println!("{}", name);
5 }
6
7 fn greet(name: String) {
8     name.push_str("acean");
9     println!("Hello, {}!", name);
10 }
```

3. The output of this program should be:



output

```
Hello, Rustacean!
Rustacean
\end{shlst]}
```

However it doesn't compile and it doesn't show the correct output. Please fix the program using mutable
 ↪ borrowing and correct choice of String type
 to get the desired output.

```
\newpage
```

1. Create a new Rust project called borrowing-3 with the cargo tool

```
\begin{shlst}[Cargo new]
cargo new borrowing-3
cd borrowing-3
```

2. In the 'src/main.rs' file, put the following content:

**borrowing-3**

```
1 fn main() {  
2     let mut mystr = "hello Rust".to_string();  
3     let partstr = &mystr[6..];  
4     mystr.push_str("aceans");  
5     println!("mystr == {} and partstr == {}", mystr, partstr);  
6 }  
7
```

3. This clearly doesn't compile. Read clippy and/or compiler output and try to explain why? How would you fix this?

Chapter 5

Program flow



The following exercises will help you familiarize yourself with loops in Rust.

Exercise 5.1: Loops - Counting

Create a Rust program called `loop-1` that will count from 0 to 100 and that will display all numbers that are divisible by 3 or 5.

Sample output

```
0
3
5
6
9
10
12
15
18
20
...
```

Exercise 5.2: Loops - Alphabet

Create a Rust program called `alphabet` that will loop through all 26 letters of the alphabet and print out the vowels.

Sample output

```
a
e
i
o
u
```

Exercise 5.3: Loops - Irish Spacing

Given the unicode string:

”♣ Dia dhuit, an Domhan! ♣”; (With some artistic freedom loosely translated as Hello World in Irish)

Kindly write a program in Rust to iterate over this string and print the individual characters in this string and insert extra spaces. E.g, the result should be:

♣ D i a d h u i t , a n D o m h a n ! ♣

✎ Exercise 5.4: Conditionals - User Input Handling



User Input Handling

```
use std::io;

fn main() {
    println!("Enter a number: ");
    let mut guess = String::new();

    io::stdin().read_line(&mut guess).expect("failed to readline");

    let my_nr: i32 = guess.trim().parse().unwrap();

    print!("You entered {}", my_nr);
}
```

Above program will request input from the user. In later modules we will explain how it works. This input is converted to a number.

Kindly extend the program to:

1. Print that the nr is zero if a 0 is keyed in.
2. Print if the nr is odd or even

For this exercise use the if conditional statements

✎ Exercise 5.5: Conditionals - Match

The same tasks as for exercise 1, but now implement it using `match`. Please do not forget the default arm.

✎ Exercise 5.6: Conditionals - Basic Calculator

Implement a basic calculator for the four primary arithmetic operations: addition, subtraction, multiplication, and division. The user provides an operation in the form of a string ("add", "subtract", "multiply", or "divide") and two numbers. Your program should then perform the requested operation on the numbers.

Steps:

1. Define an enum named `Operation` with variants for each of the 4 operations.
2. Implement a function that converts a string into the `Operation` enum.
3. Implement the calculator using a `match` expression to choose the appropriate arithmetic operation based on the enum variant.
4. Test your calculator in the main function with a few examples.

E.g. to help you with some scaffolding:



scaffolding

```
enum Operation {
    Add,
    Subtract,
    Multiply,
    Divide,
}

fn str_to_operation(s: &str) -> Option<Operation> {}

fn calculate(op: Operation, a: f64, b: f64) -> f64 {}

fn main() {
    let operation_str = "add";
    let x = 5.0;
    let y = 3.0;

    match str_to_operation(operation_str) {
        Some(op) => {
            let result = calculate(op, x, y);
            println!("Result of {}ing {} and {}: {}", operation_str, x, y, result);
        }
        None => println!("Unknown operation: {}", operation_str),
    }
}
```

Chapter 6

Complex data types



Exercise 6.1: Enums - Introduction

The data type `enum` plays a special role in Rust. They are more powerful than the `enums` we know from other languages like C and C++. They play an important part in error handling and getting optional results from functions.

The upcoming exercises will have you practice with `enums` in Rust.

- **Objective:** Learn how to define and use `enums` in Rust by creating a simple program that represents traffic light signals.
- **Steps:**
 1. Define an Enum for Traffic Light. Create an enum called `TrafficLight` with three variants: `Red`, `Yellow`, and `Green`.
 2. Write a Function to Print the Traffic Light State
 3. Write a function named `print_light` that takes a `TrafficLight` as input and prints a message indicating the color of the light.
 4. Write a Main Function to Test Your Code. In the main function, create instances of each `TrafficLight` variant (`Red`, `Yellow`, and `Green`). Call the `print_light` function for each traffic light to print its state.
 5. Compile, run and test your program

Exercise 6.2: Enums - weekly calendar

- **Objective:** Understand the basic usage of `enums` in Rust by implementing days of the week.
- **Steps:**
 1. Define an enum named `Day` that represents the days of the week. Each day should be a variant in the enum.
 2. Create a variable for each day of the week and assign the corresponding variant of the `Day` enum to it.
 3. Using a `match` expression, print a small piece of information or activity you might do on each day. For instance, "Monday is Rust practice day"

Exercise 6.3: Enums - Coffee bar

- **Objective:** Learn to use `enums` in Rust that hold data by modeling a simple coffee ordering system.
- **Scenario:** Customers can order coffee in different sizes and specify if they want milk or not. The size can be "Small", "Medium", or "Large", and milk can be "None", "Regular", or "Soy".

- **Steps:**

1. Define the CoffeeSize and MilkOption enums:
2. Create a sample order like this:



Rust

```
1 let my_order = CoffeeOrder::Order {
2     size: CoffeeSize::Medium,
3     milk: MilkOption::Soy,
4 }
```

3. In the main function describe this order using a match expression.

E.g. possible output with above order:

"You've ordered a Medium coffee with soy milk."

✍ Exercise 6.4: Structs - Introduction

The following exercises will help you practice with `structs` in Rust. The data type `struct` plays an important role in Rust by create custom data types grouping together related pieces of data. Not only for structuring data but like `enums` we will see later in the course that we can implement methods on them, forming the base of OOP in Rust.

✍ Exercise 6.5: Structs - People

Steps

1. Define a Persons struct with fields `first_name`, `last_name` and `age`. Choose appropriate data types for the field
2. Instantiate the struct in main (`let person = Person{...};`)
3. Print the contents of these structs
4. Print adult if the person has reached the adult age

✍ Exercise 6.6: Structs - Employees

In this exercise we are going to practice with embedded structs.

Steps

1. Define an Employee struct
An Employee should have:
 - A first name
 - A last name
 - A job title
2. Define a Department struct
A Department should have:
 - A department name
 - A list of the 5 (fixed) employees in that department
3. Instantiate the Department struct and 3 Employee structs.
4. Print the department info (all fields) and the Employees in that department

✍ Exercise 6.7: Vectors - Introduction

This next batch of exercises will help you practice with defining vectors, iterating over them and updating the elements in them.

✍ Exercise 6.8: Vectors - Basics

- **Objective:** practice with a vector of strings
- **Steps:**
 1. Create a vector containing the words "hello", "world", and "rust".
 2. Convert the vector of words into a single string where words are separated by a space.
 3. Print this string

Exercise 6.9: Vectors - square & double

Write a Rust program that declares and initializes a vector of i64. Populate this vector with the nrs from 0 to 100.

After that:

Iterate over the elements in the vector and double the element's value if the value is odd and square the element's value when the original value is even. Lastly, print out each value in the vector including their element nr;

E.g:

```
Element 0 -> 0
```

```
Element 1 -> 2
```

```
etc.
```

Exercise 6.10: Vectors - Bookstore

Like the Gophers the Rustlings would like to run their own bookstore, but then full of books about Rust. They need to manage the inventory of books. Each book will have a title, author, and stock count.

- **Objective:**

Create a basic inventory management system to add books, check stock, and sell books.
- **Structures:**
 - Book:

Represents a book with a title, author, and stock count.
 - Inventory:

Represents the bookstore's inventory. It uses a vector to hold the collection of books.
- **Functions:**

Add a book to the inventory.
Check the stock of a specific book by title.
List the entire inventory
Sell a book (decrease its stock by 1).

Exercise 6.11: Hashmaps - Introduction

The last set of exercises will help you practice with hashmaps.

Exercise 6.12: Hashmaps - Students

Write a Rust program that declares and initializes a hashmap `devstudents`. The hashmap should map student names to programming languages. Key of the hashmap is the name of the student, value is the name of the programming languages.

Steps

1. Populate the hashmap with the following records:

```
Fred,Golang  
Alice,Perl  
Kjell,Python  
Jarmo,Rust  
Sander,Zig
```

2. Print out all the elements in the hashmap.
3. Add a new student and language (yours to choose).

Check before you add the student if he's not already in the hashmap. If so, print a message.

4. Sander decides to join the Rust developers. Kindly change his programming language to Rust.
5. Extra credit:

Iterate over the HashMap and convert all programming languages to uppercase.

Chapter 7

Functions in Rust



In this lab we will practice with writing functions in Rust.

Exercise 7.1: Functions - Square Root

Steps

1. Write a function that returns the square root of a f64 and returns a f64. Call this function in the main body.
2. Adapt the functions so it will refuse to calculate the square root if the argument $x < 0$
Use the Result enum to return an error condition or result.

After calling the function, determine whether you have a valid result or an error and print this on the screen.

Exercise 7.2: Functions - Fibonacci Sequence

Write a recursive function that will calculate the n-th fibonacci nr

```
Fib(1) = 1
Fib(2) = 1
```

```
Fib (n) = Fib(n-1) + Fib(n-2)
```

Call the recursive function in the main() part.

Exercise 7.3: Transferring Ownership

Objective: Practice transferring ownership between functions.

Steps

1. Write a function `consume_vector` that takes ownership of a vector of `String` and prints each element.
2. In the `main` function, create a vector of `Strings` and pass it to `consume_vector`.
3. Try to use the original vector in `main` after it has been passed to `consume_vector`, and observe the compiler error. This illustrates how ownership prevents access to moved values.

Exercise 7.4: Borrowing with Functions

Objective: Understand the concept of borrowing in Rust by passing references to a function.

Steps

1. Write a function `calculate_length` that takes a reference to a `String` as an argument and returns its length without taking ownership.
2. In the `main` function, create a `String` variable and pass a reference to this variable to `calculate_length`.
3. After calling `calculate_length`, use the original `String` variable again in `main` to demonstrate that its ownership was not transferred.

Exercise 7.5: Mutable Borrowing

Objective: Use mutable borrowing to modify data within a function.

Steps

1. Write a function `append_exclamation` that takes a mutable reference to a `String` and appends an exclamation mark to the end of the string.
2. In the `main` function, create a mutable `String` variable and pass a mutable reference to this variable to `append_exclamation`.
3. Print the modified string in `main` after calling `append_exclamation` to see the effect of the mutable borrow.
4. Try creating a second mutable borrow in `main` before the first borrow's scope ends, and observe the compiler error. This demonstrates Rust's rule against having multiple mutable references to the same data in the same scope.

Chapter 8

Error handling in Rust



In this lab we are going to practice error handling in Rust.

Exercise 8.1: Quadratic Equation Solver

The first exercise concerns the quadratic equation solver again. Right now it doesn't have proper errorhandling and just displays a message when it detects that there will be no "real" solutions. But it still returns a tuple of two zeroes. You are going to transform this function so that it uses the `Result` type as output parameter/return type.

Step 1: Please clone the rust-qeq lab again.

```
git clone https://thegitcave.org/pascal/rust-qeq.git
```

Step 2: Change the program in such a way that it returns a `Result` enum.

In `Ok()` cases it should return the tuple of solutions. In `Err()` cases it should return the String "This equation does not have any real solutions"

Test this using a main function that calls this function. Also have the calling function (main) evaluate the result act upon it. If it receives an error, it should mention that and ignore the results. If there's no error, the solutions are valid and should be printed.

Exercise 8.2: Reading a File

Create a program that will open a file called `rust.txt` and read it in a string. Opening the file and reading the string will be implemented in 2 different functions. If the file cannot be opened, propagate the error to the calling function/main. If the string does not contain "Ferris", return a different Error as well.

Have the calling function 'handle' the error.

Exercise 8.3: Panic Trigger

Create a function that deliberately triggers a panic if a certain condition is met, simulating a scenario where something has gone irreversibly wrong in your program.

Step 1: Write a function named `check_age` that takes an integer representing an age.

Step 2: If the age is less than 0, the function should trigger a panic with a message indicating that age cannot be negative.

Step 3: Test this function in `main` with both a valid and an invalid age to see the panic in action.

Exercise 8.4: Result for File Operations

Practice using the `Result` type for handling potential errors when performing file operations.

Step 1: Write a function to create a file and write some text into it. The function should return `Result<(), std::io::Error>`.

Step 2: In `main`, call this function and use pattern matching to handle the `Result`. Print a success message or an error message based on the outcome.

Exercise 8.5: Option for Configuration

Implement a function that might not always return a value and use the `Option` enum to handle these cases.

Step 1: Define a function that takes a string representing a configuration key and returns an `Option<String>` representing the configuration value.

Step 2: If the key matches a predefined configuration, return the value inside `Some`. Otherwise, return `None`.

Step 3: Demonstrate using this function in `main` with both existing and non-existing keys. Use pattern matching to handle the `Option` returned by the function.

Exercise 8.6: Error Propagation with ? Operator

Learn to propagate errors in functions that return a `Result` using the `?` operator.

Step 1: Write a function that attempts to open a non-existent file, deliberately causing an error.

Step 2: The function should return `Result<(), std::io::Error>` and use the `?` operator to propagate errors.

Step 3: In `main`, call this function and handle the potential error, printing a message to the console.

Exercise 8.7: Error handling and main

Implement a Rust program that reads the contents of a file and prints them to the console. Practice using Rust's error handling mechanism with `Result` to gracefully handle potential failures in file operations.

1. Function Implementation:

- Write a function named `read_file_to_string` that takes a `filename` as input and returns the file contents as a `String`.
- The function should return a `Result<String, io::Error>` to handle the possibility of an I/O error occurring during the file operation.

2. Error Handling:

- Use Rust's error propagation operator `?` to simplify error handling within the `read_file_to_string` function.
- Ensure that if opening the file or reading to a string fails, the error is properly returned to the caller.

3. Main Function:

- In the `main` function, call `read_file_to_string` with a filename (e.g., "example.txt") to read its contents.

- Use the `?` operator to handle any errors that might occur during this operation, ensuring the program exits gracefully if an error occurs.
- If the file is read successfully, print the file contents to the console.

4. Result Propagation:

- Modify the `main` function to return a `Result<(), io::Error>`. This change propagates errors from file operations back to the Rust runtime, which will handle the error if one occurs.

Hints

- Remember to import necessary modules from `std::fs` and `std::io` for file operations and error handling.
- The `?` operator can be used to return the error early if an operation fails, simplifying the code for error handling.

Constraints

- Do not use unwrapping (`unwrap()`) for error handling in this exercise. The goal is to practice proper error propagation with `Result`.

Sample Code Structure



Scaffolding

```
1 use std::fs::File;
2 use std::io;
3 use std::io::Read;
4
5 // Your function implementation here
6
7 fn main() -> Result<(), io::Error> {
8     // Your code to call the function and handle errors
9 }
```

Expected Output

When you run your program with an existing file named "example.txt", it should print the contents of that file to the console. If the file does not exist or an error occurs, the program should gracefully handle the error and not panic.

Exercise 8.8: Custom Error Types 1

This exercise is a more hands-on guide approach to get you familiar with creating **Custom Error Types** in Rust.

The program includes functionality for dividing two numbers provided by the user, handling errors such as division by zero and non-numeric input.

Step 1: Define Custom Error Types

First, we define an enum `AppError` to represent possible errors in our application.



Define enum and Display and Error methods on it

```

1 use std::fmt;
2 use std::num::ParseFloatError;
3 use std::error::Error;
4
5 #[derive(Debug)]
6 enum AppError {
7     DivisionByZero,
8     ParseError(ParseFloatError),
9 }
10
11 impl fmt::Display for AppError {
12     fn fmt(&self, f: &mut fmt::Formatter) -> fmt::Result {
13         match *self {
14             AppError::DivisionByZero => write!(f, "Cannot divide by zero"),
15             AppError::ParseError(ref err) => write!(f, "Parse error: {}", err),
16         }
17     }
18 }
19
20 impl Error for AppError {}
21
22 impl From<ParseFloatError> for AppError {
23     fn from(err: ParseFloatError) -> AppError {
24         AppError::ParseError(err)
25     }
26 }

```

1. The 'AppError' enum includes two variants: 'DivisionByZero' for attempting to divide by zero, and 'ParseError' for errors occurring when parsing strings to floating-point numbers.
2. To integrate seamlessly with Rust's error handling, we implement the 'fmt::Display' and 'std::error::Error' traits for 'AppError'.

Understanding fmt::Display and std::error::Error in Rust

In Rust, the `fmt::Display` and `std::error::Error` traits serve specific roles in the language's error handling and output formatting systems. These traits are called by various components and mechanisms within Rust, including:

1. 1. Error Reporting

When you use the `Result` type for error handling, Rust allows you to propagate errors up the call stack using the `?` operator. If an error occurs and you propagate it using `?`, the Rust runtime needs a way to present this error to the user or calling code. This is where `std::error::Error` comes into play. It is a trait that the standard library's error handling mechanisms use to work with errors polymorphically.

2. 2. Printing and Formatting

The `fmt::Display` trait is called whenever an object needs to be formatted as a string in a user-facing manner, such as when using macros like `println!`, `format!`, or when converting the error into a string with `.to_string()`. The implementation of `fmt::Display` determines how the error message is presented to the end-user, making it a crucial part of creating user-friendly error messages.

3. Specific Components and Situations Calling These Traits

- **The Rust Standard Library:** When errors are returned or unwrapped. For example, if you call `.unwrap()` on a `Result` type and the result is an `Err`, Rust will attempt to print the error by calling `fmt::Display` to get a user-friendly message.

- **Error Handling Libraries and Frameworks:** Many Rust libraries that provide error handling utilities or extensions over the standard library's error handling mechanisms rely on these traits to generalize error handling. Libraries like `anyhow` and `thiserror` interact with these traits to provide more ergonomic error handling and error definition capabilities.
- **Logging and Diagnostic Tools:** Tools and libraries used for logging or diagnostics might call `fmt::Display` for human-readable output and `std::error::Error` for more structured error reporting, including introspection of the error chain (via the `.source()` method).

How It Works:

- **For `fmt::Display`:** When you try to print the error (e.g., `println!("{}", err);`), Rust internally calls the `fmt` function of the `fmt::Display` trait implemented for the error type.
- **For `std::error::Error`:** This trait allows for polymorphic handling of errors through methods like `.source()`, which can be used to iterate over and inspect the entire "chain" of errors, providing context about the original cause of the error. This trait is also a marker for "error-ness" in Rust, allowing errors to be treated in a generic way.

Step 2: Implement Division Logic with Error Handling

Next, we implement a function to perform division while handling errors.



Division and error handling

```
1 fn divide_numbers(numerator: &str, denominator: &str) -> Result<f64, AppError> {
2     let numerator: f64 = numerator.parse()?;
3     let denominator: f64 = denominator.parse()?;
4
5     if denominator == 0.0 {
6         Err(AppError::DivisionByZero)
7     } else {
8         Ok(numerator / denominator)
9     }
10 }
```

Step 3: Main Function to Use the Division Logic

Finally, we write a `main` function to use our division function, simulating user input and demonstrating error handling.



main()

```
1 fn main() {
2     let num = "10";
3     let denom = "0"; // Try changing this to a non-zero value or something non-numeric
4
5     match divide_numbers(num, denom) {
6         Ok(result) => println!("The result is {}", result),
7         Err(e) => match e {
8             AppError::DivisionByZero => println!("Error: {}", e),
9             AppError::ParseError(_) => println!("Please provide valid numbers. Error: {}", e),
10         },
11     }
12 }
```

Exercise 8.9: Custom Error Types 2

Now let's do it ourselves:

Create custom error types to provide more detailed error information to the callers of your function.

Step 1: Define an enum named `MyError` with variants representing different kinds of errors your application might encounter.

Step 2: Implement the `Error` trait for your `MyError` enum.

Step 3: Write a function that returns a `Result<(), MyError>`, simulating an operation that can fail in multiple specific ways.

Step 4: In `main`, call this function and use pattern matching to handle the different error cases.

Chapter 9

Testing in Rust



In this lab we are going to practice with the testing functionality of the Rust toolchain.

Exercise 9.1: Quadratic Equations

We are going to create a function that will solve quadratic equations.

The function will have a signature of:

```
fn solveqeq (a: f64, b: f64, c: f64) -> (f64, f64)
```

We will first focus on the easy way and only test with arguments a, b and c that will have solutions

You can clone the crate with git from:

SH

git clone

```
git clone https://thegitcave.org/pascal/rust-qeq.git
```

Tasks:

1. Create a test case that tests the correct outcome for a certain aX^2+bx+c quadratic equation
2. Check what happens if you test an equation that does not have real solutions
3. Implement table driven testen that will check for at least 5 different combinations

Bonus: Alter the function to return a Result with an err if there are no real solutions. Test this Result.

Exercise 9.2: Testing Arithmetic Operations

Implement and test basic arithmetic operations in Rust, including unit tests and table-driven testing.

Tasks:

- Implement a Rust module `math_operations` containing functions for addition, subtraction, multiplication, and division:
- `fn add(a: i32, b: i32) -> i32`

- `fn subtract(a: i32, b: i32) -> i32`
- `fn multiply(a: i32, b: i32) -> i32`
- `fn divide(a: i32, b: i32) -> Option<i32>` (returns `None` if `b` is 0)
- Write a test case for each arithmetic operation, ensuring testing for both positive and negative numbers.
- Implement table-driven testing for at least 5 different combinations of inputs and outputs for each operation.
- Write a test to verify that `divide` correctly returns `None` when dividing by zero.

Bonus: Implement tests for overflow checks in `add` and `multiply` functions, altering them to return a `Result<i32, String>` in case of overflow.

Exercise 9.3: Testing String Manipulation

Write and test functions for string manipulation, focusing on edge cases and table-driven tests.

Tasks:

- Implement a Rust module `string_utils` with the following functions:
 - `fn reverse(s: &str) -> String`
 - `fn to_uppercase(s: &str) -> String`
 - `fn concatenate(s1: &str, s2: &str) -> String`
- Create unit tests for each function to assess basic functionality.
- Use table-driven tests for the `reverse` and `to_uppercase` functions with various strings, including empty strings and those containing special characters.
- Test `concatenate` for correct handling of an empty string concatenated with a non-empty string.

Exercise 9.4: Testing Structs and Methods

Test methods associated with a `Rectangle` struct for calculating area and perimeter.

Tasks:

- Define a `Rectangle` struct with `width: u32` and `height: u32`.
- Implement `area(&self) -> u32` and `perimeter(&self) -> u32` methods.
- Write unit tests for these methods across several instances of `Rectangle`.
- Use table-driven tests to check these methods with at least 5 different `Rectangle` instances.

Bonus: Add a `can_hold(&self, other: &Rectangle) -> bool` method and test cases, including for rectangles with identical dimensions.

Exercise 9.5: Testing Error Handling

Implement and test a function that parses a string into an integer, focusing on error handling.

Tasks:

- Implement `fn parse_number(s: &str) -> Result<i32, String>` for parsing a string to `i32`.
- Write a unit test for successful parsing.

- Use table-driven testing to assess both successful and failing parses with at least 5 different strings.
- Test the error message for descriptiveness upon failure.

Exercise 9.6: Testing Panics

Write and test a function that panics under specific conditions.

Tasks:

- Implement `fn assert_positive(n: i32)` that panics if `n` is not positive.
- Use `#[should_panic]` to test for correct panic message when passed a negative number.
- Implement table-driven testing for various negative inputs and ensure no panic for positive inputs.

Bonus: Extend `assert_positive` to panic with a specific message if the number is zero and add relevant tests.

Chapter 10

Debugging in Rust



In this lab we are going to practice with the Rust debuggers

Exercise 10.1: Buggy Program 1

Given the following Rust program:



buggy program 1

```

1 fn main() {
2     let numbers = vec![1, 2, 3, 4, 5];
3     let mut largest = numbers[0];
4
5     for number in &numbers { // Changed to borrow `numbers` instead of moving it
6         if *number < largest { // Dereference `number` since it's a reference to the value in `numbers`
7             largest = *number;
8         }
9     }
10
11     println!("The largest number is {}", largest);
12 }
```

Tasks:

1. Compile the program and evaluate the result. We should get 5 as the largest nr, but that isn't the case
2. Use rust-gdb or rust-lddb to debug the program.
3. Set a breakpoint on line 6
4. Run the program
5. Single step through the program with `n` and print the value of the variables `number` and `largest`
6. Analyze and fix the bug

Exercise 10.2: Buggy Program 2

Given the following Rust program:



buggy program 2

```
1 fn remove_negatives(numbers: &mut Vec<i32>) {
2     for i in 0..numbers.len() {
3         if numbers[i] < 0 {
4             numbers.remove(i);
5         }
6     }
7 }
8
9 fn main() {
10     let mut numbers = vec![10, -1, 32, -42, 55];
11     remove_negatives(&mut numbers);
12     println!("Numbers after removing negatives: {:?}", numbers);
13 }
```

Tasks:

1. Compile the program and evaluate the result. It isn't giving the right results. Why?
2. Use rust-gdb or rust-lldb to debug the program.
3. Analyze and fix the bug.

Chapter 11

Object Oriented Programming in Rust



This lab we will practice with some of the OOP techniques in Rust.

Exercise 11.1: Vectors

- Design and implement a datatype in Rust to hold a mathematical vector, e.g:

$$\begin{bmatrix} 3.0 \\ 1.5 \\ -2.0 \end{bmatrix}$$

The vector is one dimensional and has 3 nrs.

- Implement the Display trait on this datatype to pretty print it. E.g. something like shown above.
- Implement a method so that we can calculate the distance between 2 vectors using the Euclidean distance formula:
vector A(1,1,2) vector B(2,1,2)
Then the distance is $\sqrt{(x1 - x2)^2 + (y1 - y2)^2 + (z1 - z2)^2}$
- Overload the * operator so that two vectors can be multiplied with each other (use the dot product to keep it simple)
- Overload the * operator so you can multiply the vector with a scalar
- Extra credit for adding overloading of '**' for the Cross product of 2 vectors

Exercise 11.2: Geometry - Shapes

Let's take a look at structs.

Define 3 structs like this:

- Square → having side: f64 as it's only attribute
- Triangle → having side1, side2, side3 all of type f64 as attributes
- Circle → having radius as it's only attribute

Write for each of these 3 structs a static method called New so we can instantiate new Squares, Triangles and Circles

Exercise 11.3: Geometry - Methods

Write the following 2 methods for each of the 3 structs:

- Perimeter

This returns the perimeter for each of the shapes (square $\rightarrow 4 \times \text{side}$, triangle \rightarrow sum of all sides, circle $2 \times r \times \text{PI}$)

- Area

This returns the area of each of the shapes

Test all the methods for each of the shapes.

Exercise 11.4: Geometry - Altering Shapes

- Write for each of the shapes a method called “scale” that will alter the dimension of the shape. E.g. scale=2.0 will double the sides of a square, etc.
- Display the properties of the shapes to see if indeed they are ‘scaled’

The following exercises build upon the methods we programmed in the previous *Geometry* exercises. In this part we will define a trait called *Shape* that will define the methods *perimeter* and *area*. This means all structs/object that will implement these methods will fit the *Shape* trait and we can make Generic functions for *area* and *perimeter* that will accept struct/objects that will implement the *Shape* trait.

Exercise 11.5: Traits - Geometry

1. Define a trait called *Shape* that requires the methods *area()* and *perimeter()* to be implemented.
2. Define a generic function *area* that takes a *Shape* and returns its area.
3. Define a generic function *perimeter* that takes a *Shape* and returns its perimeter.
4. Test the functions in your *main()* by calling it and passing it *Circle*, *Square* and *Rectangle* objects.
5. Add the methods *scale()* to the trait and define a generic function *scale* that takes a *Shape* and will scale that *Shape*.
6. Test the generic *scale()* function by passing it *Circle*, *Square* and *Rectangle* objects.

Exercise 11.6: Traits - Temperatures

1. Design and implement a user type that represents temperature in degrees Celsius
2. Design and implement a user type that represents temperature in degrees Fahrenheit
3. Implement on both types the *Display* trait (method *fmt*) that will pretty print the degrees with the correct unit.
4. Design and implement a method that will convert from degrees Celsius to Fahrenheit and use the right return type
5. Design and implement a method that will convert from degrees Fahrenheit to Celsius and use the right return type
6. Test your program using the conversion method of C to F and vice versa. The right value and the right units should be printed.

Exercise 11.7: Traits - Students

1. Design and implement a struct called *Student* that will have *firstname*, *lastname*, *timestamp* and a vector with all the languages this person masters.
2. Design and implement (not derive) the *clone* trait for this struct and add as extra functionality an update of the timestamp when the cloning is taking place.
3. Test the cloning of the struct

Exercise 11.8: Generics - Sums

- Create a Generic function *sum()* that will calculate the sum of all types of integers and floats in a vector.
- Use proper Trait bounds to get your generic function working.



sum

```
fn main() {
    let v1 = vec![ 1,2,3,4,5,6,7,8,9 ];
    let v2 = vec![ 1.2,28.1,1.2,9.1 ];

    println!("{}",sum(&v1));
    println!("{}",sum(&v2));
}
```

- Tip; assigning the 0 value to start with can be tedious if you don't know the type. You can use something like this:

**tip**

```
let mut sum: T = Default::default();
```

It provides a suitable default value for the type of var involved.

Test the Generic function.

Bonus: try to write a Generic function that will calculate the average of a vector of floats or integers. (non-trivial)

Exercise 11.9: Generics - Statistics

For a real challenge create generic functions that will calculate sum, avg, variance and standard deviation of a vector of generic types. E.g. the function should work for all integers and all floats.

Chapter 12

Closures



In this lab we will practice writing and using closures in Rust

 **Exercise 12.1: Populations** The following struct is given:



cities

```
struct City {
    name: String,
    province: String,
    population: u64
}

fn main() {
    let c = City{ name: "Bergen".to_string(), province: "Limburg".to_string(), population:
        ↪ 5105 };
}
```


It contains fields name, population and province. Write a program in which you create a list of cities using a vector and create a closure to sort this vector in ascending order on province and print the results.

You can use the following data to populate your list:

Next, sort the list on population in descending order and print the results.

Table 12.1: **Cities**

City	Province	Population
Bergen	Limburg	5105
Nijmegen	Gelderland	81002
Maastricht	Limburg	63201
Amsterdam	Noordholland	982102
Rotterdam	Zuidholland	1030101

 **Exercise 12.2: Currying** Currying is when a function with many arguments is being made more accessible by having a helper function that has fewer arguments by implementing default. Goal is to write such a 'curried' function in Rust:



Currying

```
fn add(a: u32, b: u32) -> u32 {
    a + b
}

fn main() {
    let add5 = |x| add(5, x);
    println!("{}", add5(5));
}
```

Above example shows you how to curry a addition function.

Now write a function `greeting()` that will have two parameters: `salutation` → that will contain for example "Gutentag", "Buenas Dias" etc `name` → That will contain the name of the one to be greeted

The salutation should be a default "Hasta la vista" in the closure calling the function. E.g. your closure called *quickhi* should only have name as argument and use `Hasta la vista` as the default greeting part supplied to your `greeting()` function.

Chapter 13

Iterators



In this lab we will practice writing and using closures in Rust

Exercise 13.1: Numbers

1. Create a vector with the following nrs: [-4,2,3,1,-7,21,42,31,-7]
2. Print all the nrs using an iterator in the vector
3. Using a adapter iterator and a closure, double the even nrs and half the odd nrs.
4. Using the filter adapter, filter out the nrs that are divisable by 5

Exercise 13.2: Custom Iterators

We are now going to create an iterator ourselves.

The example shows an iterator that generates a prime on each call to next():

**example**

```

pub fn is_prime(n: u64) -> bool {
    if n < 4 {
        n > 1
    } else if n % 2 == 0 || n % 3 == 0 {
        false
    } else {
        let max_p = (n as f64).sqrt().ceil() as u64;
        match (5..=max_p)
            .step_by(6)
            .find(|p| n % p == 0 || n % (p + 2) == 0)
        {
            Some(_) => false,
            None => true,
        }
    }
}

pub struct Prime {
    curr: u64,
    next: u64,
}

impl Prime {
    pub fn new() -> Prime {
        Prime { curr: 2, next: 3 }
    }
}

impl Iterator for Prime {
    type Item = u64;

    fn next(&mut self) -> Option<Self::Item> {
        let prime = self.curr;
        self.curr = self.next;
        loop {
            self.next += match self.next % 6 {
                1 => 4,
                _ => 2,
            };
            if is_prime(self.next) {
                break;
            }
        }
        Some(prime)
    }
}

fn main() {
    let mut p = Prime::new();

    for i in 1..=10 {
        let _ = if let Some(v) = p.next() {
            println!("Prime {} = {}", i, v);
        };
    }
}

```

1. Create an iterator that generates Faculty nrs.
2. Create an iterator that generates Fibonacci nrs.

Test your iterators.

Chapter 14

Lifetimes



In this lab we are going to practice with lifetimes in Rust

Requirements:

- A fresh mind
- Spirit
- Coffee at arms length

Exercise 14.1: Authors & Citations

Given the following *struct*, which represents an author and a citation:



AuthorsAndExcerpts

```
struct AuthorAndExcerpt {
    author: &str,
    excerpt: &str,
}

fn main() {
    let name = "Hemingway";
    let text =
        → "The world breaks everyone, and afterward, some are strong at the broken places.";
    let record = AuthorAndExcerpt::new_author_and_excerpt(name, text);
}
```

- Modify the *AuthorAndExcerpt* struct to use appropriate lifetime annotations so that it compiles correctly.
- Implement a function *new_author_and_excerpt* that accepts two string slices: one for the author's name and one for an excerpt of their writing. The function should return an instance of *AuthorAndExcerpt*.

Exercise 14.2: Faulty lifetimes

Kindly make the following Rust program compile and work by bringing on proper lifetime annotations where applicable:



lifetimes

```
struct Person {  
    name: &str,  
}  
  
fn make_person(name: &str) -> Person {  
    Person { name }  
}  
  
fn main() {  
    let name = "Alice";  
    let alice = make_person(name);  
    println!("Name: {}", alice.name);  
}
```

Exercise 14.3: Slicing through life

Implement a function that accepts two string slices with potentially different lifetimes and returns the slice that starts with the character 'a' or the first slice if neither starts with 'a'. However, because they can have different lifetimes, you need to handle lifetimes carefully.



slicing

```
fn starts_with_a<'a, 'b>(s1: &'a str, s2: &'b str) -> &'a str {}  
  
fn main() {  
    let string1 = "example";  
    let string2 = "sample";  
    let result = starts_with_a(string1, string2);  
}
```


Chapter 15

Crates



In this set of exercises we will practice with setting up a more complex Rust project.

We will learn Create a library crate and publishing it

Exercise 15.1: Creating, Publishing, and Using a Rust Library Crate

Creating the Crate

Begin by creating a new library crate. Open a terminal and run the following command:

A terminal window icon with a blue tab labeled 'cargo'. The terminal content shows the command 'cargo new --lib snake_case_converter' being entered.

```
cargo new --lib snake_case_converter
```

This command generates a new crate named `snake_case_converter` with the basic structure for a library.

Writing the Library Code

Navigate into your new crate directory. You will write a function to convert strings to snake case in the `src/lib.rs` file. Here is the code for the `lib.rs` file:

**lib.rs**

```

1  /// Converts a given string to snake_case.
2  ///
3  /// # Examples
4  ///
5  /// ```
6  /// let text = "HelloWorld";
7  /// assert_eq!(snake_case_converter::to_snake_case(text), "hello_world");
8  /// ```
9  pub fn to_snake_case(s: &str) -> String {
10     s.chars().enumerate().map(|(i, c)| {
11         if c.is_uppercase() {
12             if i != 0 { format!("_{}", c.to_lowercase()) }
13             else { c.to_lowercase().to_string() }
14         } else {
15             c.to_string()
16         }
17     }).collect()
18 }

```

Testing Your Library

Add tests to ensure your function works correctly. Below the function in the `lib.rs` file, include:

**Add unit tests**

```

1  #[cfg(test)]
2  mod tests {
3      use super::*;
4
5      #[test]
6      fn it_converts_to_snake_case() {
7          assert_eq!(to_snake_case("HelloWorld"), "hello_world");
8          assert_eq!(to_snake_case("RustIsAwesome"), "rust_is_awesome");
9      }
10 }

```

Run the tests with the following command:

**Cargo test**

```
cargo test
```

Documenting Your Library

Use `rustdoc` comments to document your library. The example provided in the `lib.rs` section includes documentation for the `to_snake_case` function.

Preparing for Publication

Update the `Cargo.toml` file with the necessary metadata:

**Cargo.toml**

```
1 [package]
2 name = "snake_case_converter"
3 version = "0.1.0"
4 authors = ["Your Name <you@example.com>"]
5 edition = "2018"
6 description = "A simple library to convert strings to snake case."
7 license = "MIT OR Apache-2.0"
8 repository = "https://github.com/yourusername/snake_case_converter"
9
10 [dependencies]
```

Publishing the Crate

To publish crates to crates.io, you need an API token for authentication. This document outlines the steps to retrieve your API token from crates.io.

Steps to Retrieve Your API Token

1. Navigate to <https://crates.io> in your web browser and log in to your account. If you do not have an account, you will need to create one by clicking on the "Sign In" link and following the instructions.
2. Once logged in, click on your username in the top right corner of the page to go to your account page.
3. On your account page, click on the "Settings" tab to access your account settings.
4. Scroll down to the "API Access" section. Here, you will see a button labeled "New Token."
5. Click on the "New Token" button. A dialog will appear prompting you to name your token. It's helpful to give it a descriptive name related to its intended use, such as "Cargo Publishing."
6. After naming your token, click the "Create Token" button in the dialog.
7. Your new API token will be displayed. Be sure to copy this token and store it in a safe place. You will not be able to view the token again once you navigate away from this page.

Important Note

Keep your API token secure and do not share it with others. If you believe your token has been compromised, you should generate a new one from the crates.io account settings page and revoke the old token.

Before publishing, you must login to crates.io:

**cargo login**

```
cargo login <your_api_key>
```


Then, publish your crate:

**cargo publish**

```
cargo publish
```

Using Your Published Crate

Create a new package;



```
cargo publish

cargo use-snake
```


To use your newly published crate in another Rust project, add it as a dependency in that project's `Cargo.toml`:



```
Cargo.toml

1 [dependencies]
2 snake_case_converter = "0.1.0"
```

Then, you can use the `to_snake_case` function in that project's code:



```
language=Rust

1 use snake_case_converter::to_snake_case;
2
3 fn main() {
4     let text = "HelloWorld";
5     println!("{}", text, snake_case_converter::to_snake_case(text));
6 }
```

Exercise 15.2: My 2nd Crate

Objective: Creating a binary crate using multiple modules

Remember:

- A package is the largest unit of distribution in Rust. It contains a `Cargo.toml` file, which is the manifest file that describes the package. A package can contain zero or one library crates and any number of binary crates. When we say "crate" in this context, we are referring to a "compilable artifact" – essentially a library or a binary. Most functionality in the Rust ecosystem is provided in the form of packages.
- A crate is a binary or library. If a package has multiple binaries (as seen in a previous example where we had multiple `.rs` files in a `bin/` directory), each one is a separate crate. A library crate is another type of crate that provides reusable functionality and can be depended upon by other crates. Every crate has a root module, which can optionally have child modules.
- Modules are the primary way to organize and split code within a single crate. They form a hierarchical namespace system. Rust uses a file-based module system, which means that each file can correspond to a module.

So let's get started:

1. Create a new rust project called `cases` e.g. `cargo new cases`
2. In the `src` directory create a file called `lib.rs`

3. In this file create a function with signature `pub fn to_pascal_case(s: &str) → String` that will accept a string slice and return that string in PascalCase. Your input string can be anything, including already in PascalCase, snake_case, SCREAMING_SNAKE_CASE or camelCase
4. Create unit tests to test the function
5. Create a `usage.rs` Rust example program in `examples` to use your library
6. If all your tests are ok and you are happy with your crate, publish it using `cargo publish`

Chapter 16

OS functions



Exercise 16.1: Environment Variables and Command Line Arguments

Objective: Practice working with environment variables and processing command line arguments in Rust.

Instructions:

1. Create a new Rust project named `env_and_args`:

```
cargo new env_and_args
cd env_and_args
```

2. Modify `src/main.rs` to parse command line arguments and read an environment variable:



Rust code

```
1 use std::env;
2
3 fn main() {
4     let args: Vec<String> = env::args().collect();
5     println!("Command line arguments: {:?}", args);
6
7     match env::var("MY_ENV_VAR") {
8         Ok(val) => println!("MY_ENV_VAR: {:?}", val),
9         Err(e) => println!("Couldn't read MY_ENV_VAR ({})", e),
10    };
11 }
```

3. Run your program with command line arguments and an environment variable set:

```
MY_ENV_VAR=test cargo run -- arg1 arg2
```

Extra Challenges:

1. ****Add support for optional arguments with default values.**** Modify your program to support an optional `-t` or `--timeout` argument with a default timeout value if not specified.
2. ****Implement a help option `-h` or `--help`.** Add a help option that prints out detailed usage instructions for your program, including descriptions for all command line arguments and environment variables.

3. ****Environment variable override.**** Allow an environment variable to override a command line argument. For example, if `MY_ENV_VAR` is set, it could automatically set or override a specific command line option in your program, demonstrating the precedence of environment variables over command line arguments.

✍ Exercise 16.2: Directory Traversal and File Metadata

Objective: Practice traversing directory structures, finding files, and displaying their metadata.

Instructions:

1. Create a new Rust project named `dir_traverse_meta`:

```
cargo new dir_traverse_meta
cd dir_traverse_meta
```

2. In `src/main.rs`, implement code to traverse directories, find a specific file type (e.g., ".rs"), and display its metadata:



Rust code

```
1 use std::env;
2 use std::fs::{self, DirEntry};
3 use std::io;
4 use std::path::Path;
5
6 fn visit_dirs(dir: &Path, cb: &dyn Fn(&DirEntry) -> io::Result<()>) {
7     if dir.is_dir() {
8         for entry in fs::read_dir(dir)? {
9             let entry = entry?;
10            let path = entry.path();
11            if path.is_dir() {
12                visit_dirs(&path, cb)?;
13            } else {
14                cb(&entry);
15            }
16        }
17    }
18    Ok(())
19 }
20
21 fn main() {
22     let current_dir = env::current_dir().expect("Failed to determine current directory");
23     visit_dirs(&current_dir, &|entry: &DirEntry| {
24         let path = entry.path();
25         if path.extension().map_or(false, |ext| ext == "rs") {
26             println!("Rust file: {:?}", path);
27             let metadata = entry.metadata().expect("Failed to get file metadata");
28             println!("Last modified: {:?}",
29                 ↪ metadata.modified().expect("Failed to get modification time"));
30         }
31     }).expect("Failed to traverse directories");
32 }
```

3. Run your program to find Rust files and display their last modified timestamp.

```
cargo run
```

Extra Challenges:

1. ****Filter by file size:**** Extend the directory traversal to filter files by a minimum size, specified as a command line argument.

2. ****Calculate directory size:**** Modify your program to calculate and display the total size of all files within a directory, including its subdirectories.
3. ****Watch for file changes:**** Use the notify crate to monitor the directory structure for changes in real-time, printing out changed file paths as they occur.

Exercise 16.3: Spawning Processes and IPC

Objective: Learn how to spawn processes, share information between parent and child processes, and execute a command from the child process.

Instructions:

1. Create a new Rust project named `spawn_and_ipc`:

```
cargo new spawn_and_ipc
cd spawn_and_ipc
```

2. Modify `src/main.rs` to spawn a child process that executes a command and captures its output:



Rust code

```
1 use std::process::{Command, Stdio};
2
3 fn main() {
4     let output = Command::new("echo")
5         .arg("Hello from the child process!")
6         .output()
7         .expect("Failed to execute command");
8
9     println!("Child process output: {}", String::from_utf8_lossy(&output.stdout));
10 }
```

3. Run your program to observe the parent process spawning a child process that executes a command and captures its output.

```
cargo run
```

Extra Challenges:

1. ****Capture and Process Output:**** Modify the child process to capture its output. Then, filter or process this output in the parent process. For example, use a command that lists files and then filter these results in Rust.



Rust code

```
1 let output = Command::new("ls")
2     .output()
3     .expect("Failed to execute command");
4
5 let lines = String::from_utf8_lossy(&output.stdout);
6 lines.split('\n').filter(|line| line.contains(".rs")).for_each(|line| println!("{}", line));
```

2. ****Error Handling:**** Implement error handling for scenarios where the child process cannot be spawned or returns an error code. Display a custom error message to the user.

**Rust code**

```

1  if !output.status.success() {
2      eprintln!("Error executing command");
3  }

```

3. **Inter-process Communication (IPC):** Demonstrate a simple form of IPC by sending a string from the parent process to the child process's stdin, and then capture and display the output in the parent process.

**Rust code**

```

1  let mut child = Command::new("grep")
2      .arg("hello")
3      .stdin(Stdio::piped())
4      .stdout(Stdio::piped())
5      .spawn()
6      .expect("Failed to spawn child process");
7
8  if let Some(ref mut stdin) = child.stdin {
9      use std::io::Write;
10     stdin.write_all("hello world\n".as_bytes()).expect("Failed to write to child stdin");
11 }
12
13 let output = child.wait_with_output().expect("Failed to read child output");
14 println!("Child process filtered output: {}", String::from_utf8_lossy(&output.stdout));

```

Exercise 16.4: Using inotify for Filesystem Events

Objective: Implement a program that uses inotify to monitor filesystem events.

Instructions:

1. Create a new Rust project named `inotify_example`:

```

cargo new inotify_example
cd inotify_example

```

2. Add the notify crate to your `Cargo.toml`:

**TOML config**

```

1  notify = "5.0"

```

3. In `src/main.rs`, use the `notify` crate to monitor a directory for changes:

**Rust code**

```

1 use notify::{watcher, RecursiveMode, Watcher};
2 use std::sync::mpsc::channel;
3 use std::time::Duration;
4
5 fn main() {
6     let (tx, rx) = channel();
7     let mut watcher = watcher(tx, Duration::from_secs(10)).unwrap();
8     watcher.watch(".", RecursiveMode::Recursive).unwrap();
9
10    loop {
11        match rx.recv() {
12            Ok(event) => println!("{:?}", event),
13            Err(e) => println!("watch error: {:?}", e),
14        }
15    }
16 }

```

4. Run your program and make changes in the monitored directory to see the events.

`cargo run`

Extra Challenges:

1. **Filter Events:** Modify the program to only print events related to file creation. Ignore other types of events.

**Rust code**

```

1 match event {
2     Ok(notify::DebounceEvent::Create(path)) => println!("File created: {:?}", path),
3     _ => {} // Ignore other events
4 }

```

2. **Aggregate Events:** Implement logic to aggregate events over a short time period (e.g., 2 seconds) and report a summary of changes (e.g., number of files created).

**Rust code**

```
1 use std::collections::HashSet;
2 use std::thread;
3 use std::time::{Duration, Instant};
4
5 let mut paths = HashSet::new();
6 let start = Instant::now();
7
8 while start.elapsed() < Duration::from_secs(2) {
9     if let Ok(event) = rx.try_recv() {
10         if let notify::DebounceEvent::Create(path) = event {
11             paths.insert(path);
12         }
13     }
14 }
15
16 println!("Files created in the last 2 seconds: {:?}", paths.len());
```

3. ****Monitor Specific File Types:**** Enhance the watcher to notify only when files of a specific type (e.g., ".txt") are modified.

**Rust code**

```
1 match event {
2     Ok(notify::DebounceEvent::Write(path)) if path.extension().map_or(false, |ext| ext == "txt") => {
3         println!("Text file modified: {:?}", path)
4     },
5     _ => {}
6 }
```

Chapter 17

Benchmarking and Profiling Rust programs



In this lab we will practice with **benchmarking** functions using the external benchmark suite Criterion.

 **Exercise 17.1: benchmarking** Let's setup a new Rust library crate project with

```
cargo new myfac --lib
```

For the lib.rs file use the following content to implement a recursive “fac(n)” function.



faculty

```
#[inline]
pub fn fac(n: u64) -> u64 {
    match n {
        0 => 1,
        n => n*fac(n-1)
    }
}
```

To setup for Criterion benchmarking alter your Cargo.toml file like this:



Cargo.toml

```

1  [package]
2  name = "myfac"
3  version = "0.1.0"
4  edition = "2021"
5
6  # See more keys and their definitions at https://doc.rust-lang.org/cargo/reference/manifest.html
7
8  [dependencies]
9
10 [dev-dependencies]
11 criterion = { version="0.4.0", features=["html_reports"] }
12
13 [[bench]]
14 name = "rust-fac-bench"
15 harness = false

```

Create a directory called benches in the root of your project like this:

```

|-- benches
|   |-- rust-fac-bench.rs
|-- Cargo.lock
|-- Cargo.toml
|-- src
|   |-- lib.rs
|   |-- main.rs

```

In this benches directory create a file called rust-fac-bench.rs

with the following content:



faculty benchmark

```

use criterion::{black_box, criterion_group, criterion_main, Criterion};
use myfac::fac;

pub fn criterion_benchmark(c: &mut Criterion) {
    c.bench_function("fac 20", |b| b.iter(|| fac(black_box(20))));
}

criterion_group!(benches, criterion_benchmark);
criterion_main!(benches);

```

You can now run the Criterion benchmark like this:

```
cargo bench
```

Observe the output.

In the directory target/criterion/reports you will find an index.html that contains a HTML report. You can read it with your favorite webbrowser.

Now let's add an optimization:

Please alter the `fac` function in the `lib.rs` file like this:



faculty2

```
#[inline]
pub fn fac(num: u64) -> u64 {
    (1..=num).fold(1, |acc, v| acc * v)
}
```

And run the benchmarks again. Look at the timings/numbers in the output of the `cargo bench` command and in the generated html report. What do you observe?

Can you make more optimizations that will improve the timings?

Exercise 17.2: Fibonacci

1. Write a recursive function that will calculate the n -th fibonacci nr
 $\text{Fib}(1) = 1$ $\text{Fib}(2) = 1$
 $\text{Fib}(n) = \text{Fib}(n-1) + \text{Fib}(n-2)$
 Benchmark this function with Criterion.
2. Write an iterative implementation and benchmark this too. Observe the results.
3. Bonus: try to optimize the recursive implementation. E.g. use memoization or something else that will reduce the nr of recursive calls. Use criterion to verify the impact of your optimizations.

Exercise 17.3: Ackermann

The king of recursion can most probably be found in the so called Ackermann function. This one explodes quickly, starting at (4,0)



Ackermann

```
fn ackermann(m: u64, n: u64) -> u64 {
    match(m, n) {
        (0, n) => n+1,
        (m, 0) => ackermann(m-1, 1),
        (m, n) => ackermann(m-1, ackermann(m, n-1))
    }
}

fn main() {
    println!("_ Ackermann Function's Calculation _");

    for m in 0..5 {
        for n in 0..(16-m) {
            println!("ackermann({}, {}) = {}", m, n, ackermann(m, n));
        }
    }
}
```

Bonus: try to benchmark and optimize the `Ackermann()` function.

Chapter 18

Smart Pointers



Exercise 18.1: Exploring Dynamic Memory Allocation with Box

Objective: Practice using 'Box' for dynamic memory allocation and understand its role in Rust's ownership system.

Instructions:

1. Create a new Rust project named `dynamic_memory_box`:

```
cargo new dynamic_memory_box
cd dynamic_memory_box
```

2. Modify `src/main.rs` to demonstrate the use of 'Box' to allocate memory for a large vector on the heap:



Rust code

```
1 fn main() {
2     // Creating a large vector with 1 million elements
3     let large_vector: Box<Vec<i32>> = Box::new(vec![0; 1_000_000]);
4     println!("Successfully created a large vector with {} elements on the heap", large_vector.len());
5
6     // Demonstrating ownership transfer
7     let another_location = take_ownership(large_vector);
8     println!("The large vector now lives in a new function: {:?}", another_location.len());
9 }
10
11 fn take_ownership(data: Box<Vec<i32>>) -> Box<Vec<i32>> {
12     // Function takes ownership of the Box, simulating a transfer of ownership
13     data
14 }
```

3. Run your program to observe how 'Box' allows for heap allocation and ownership transfer:

```
cargo run
```

Extra Challenges:

1. ****Exploring Box with Structs:**** Modify your program to include a struct that contains several fields, including a 'Box' pointing to a large data structure. Demonstrate transferring ownership of this struct between functions.

2. **Boxing Traits:** Implement a trait with multiple methods and use 'Box' to create a dynamic dispatch scenario. Show how 'Box' allows for polymorphic behavior at runtime.
3. **Performance Measurement:** Add timing code to measure the difference in performance when allocating large data structures on the stack versus using 'Box' to allocate them on the heap. Discuss the implications of your findings.

Exercise 18.2: Creating a Recursive Data Structure with Box

Objective: Learn to use 'Box' for defining recursive data structures in Rust by creating a simple singly-linked list.

Instructions:

1. Create a new Rust project named `recursive_box`:

```
cargo new recursive_box
cd recursive_box
```

2. Modify `src/main.rs` to define a singly-linked list using 'Box' for its recursive nature:



Rust code

```
1 // Define the List enum with variants for the end of the list (Nil) and elements (Cons)
2 enum List {
3     Cons(i32, Box<List>),
4     Nil,
5 }
6
7 fn main() {
8     // Use the List enum to create a simple linked list of integers: 1 -> 2 -> 3
9     let list = List::Cons(1,
10         Box::new(List::Cons(2,
11             Box::new(List::Cons(3, Box::new(List::Nil))))));
12
13     // Implement function to print list elements to console, demonstrating traversal
14     print_list(&list);
15 }
16
17 // Function to traverse and print each element of the list
18 fn print_list(list: &List) {
19     match list {
20         List::Cons(value, next) => {
21             println!("{}", value);
22             print_list(next);
23         },
24         List::Nil => return,
25     }
26 }
```

3. Run your program to observe how 'Box' enables recursive data structures without causing infinite size issues at compile time:

```
cargo run
```

Extra Challenges:

1. ****Implement List Operations:**** Add functions to append elements to the end of the list, remove elements from the list, or find an element in the list. Each function should manipulate the list structure while maintaining proper ownership and borrowing rules.
2. ****Explore Deep Recursion:**** Modify your list to contain a large number of elements and observe how Rust manages memory for deep recursion. Discuss the implications of stack overflow and how 'Box' helps prevent it for the data structure itself.
3. ****Custom Box Implementation:**** As an advanced challenge, try implementing your own version of 'Box' (e.g., 'MyBox') with basic functionality. Use 'MyBox' in your recursive data structure to understand how 'Box' works under the hood.

Exercise 18.3: Creating a Binary Tree with Smart Pointers

Objective: Understand and apply Rust's smart pointers – 'Box', 'Rc', and 'RefCell' – to create a mutable binary tree structure.

Instructions:

1. Create a new Rust project named `binary_tree_smart_pointers`:

```
cargo new binary_tree_smart_pointers
cd binary_tree_smart_pointers
```

2. Modify `src/main.rs` to define a binary tree using 'Box' for static ownership, 'Rc' for shared ownership, and 'RefCell' for interior mutability:



Rust code

```
1 use std::rc::Rc;
2 use std::cell::RefCell;
3
4 // Define a node in the binary tree
5 struct TreeNode {
6     value: i32,
7     left: Option<Rc<RefCell<TreeNode>>>,
8     right: Option<Rc<RefCell<TreeNode>>>,
9 }
10
11 // Function to create a new tree node
12 fn new_node(value: i32) -> Rc<RefCell<TreeNode>> {
13     Rc::new(RefCell::new(TreeNode { value, left: None, right: None }))
14 }
15
16 fn main() {
17     // Create the root node
18     let root = new_node(1);
19
20     // Create left and right children
21     let left_child = new_node(2);
22     let right_child = new_node(3);
23
24     // Attach children to the root
25     root.borrow_mut().left = Some(left_child);
26     root.borrow_mut().right = Some(right_child);
27
28     // Implement function to print tree nodes in order, demonstrating traversal
29     print_tree(&root);
30 }
31
32 // Function to traverse and print the tree in-order
33 fn print_tree(node: &Rc<RefCell<TreeNode>>) {
34     if let Some(ref left) = node.borrow().left {
35         print_tree(left);
36     }
37     println!("{}", node.borrow().value);
38     if let Some(ref right) = node.borrow().right {
39         print_tree(right);
40     }
41 }
```

3. Run your program to demonstrate the creation and traversal of a binary tree using smart pointers:

```
cargo run
```

Extra Challenges:

1. ****Mutable Tree Operations:**** Implement functions to add and remove nodes from the tree, taking advantage of 'Rc' and 'RefCell' to modify the tree structure while maintaining shared ownership.
2. ****Balancing the Tree:**** Add functionality to balance the tree automatically when nodes are added or removed, ensuring optimal traversal times.
3. ****Smart Pointer Efficiency:**** Discuss the trade-offs in memory usage and performance when using 'Rc' and 'RefCell' in a recursive data structure. Consider scenarios where smart pointer overhead is acceptable versus when it might be a limiting factor.

Exercise 18.4: Implementing a Thread-Safe Counter with RwLock

Objective: Learn to use 'RwLock' in Rust to implement a thread-safe counter that supports high concurrency for read operations while ensuring exclusive access for writes.

Instructions:

1. Create a new Rust project named `rwlock_counter`:

```
cargo new rwlock_counter  
cd rwlock_counter
```

2. Modify `src/main.rs` to define a thread-safe counter using 'RwLock':



Rust code

```

1 use std::sync::{Arc, RwLock};
2 use std::thread;
3
4 struct Counter {
5     count: RwLock<i32>,
6 }
7
8 impl Counter {
9     // Function to create a new Counter
10    fn new() -> Self {
11        Counter {
12            count: RwLock::new(0),
13        }
14    }
15
16    // Function to increment the counter
17    fn increment(&self) {
18        let mut count = self.count.write().unwrap();
19        *count += 1;
20    }
21
22    // Function to get the current value of the counter
23    fn get_count(&self) -> i32 {
24        let count = self.count.read().unwrap();
25        *count
26    }
27 }
28
29 fn main() {
30     let counter = Arc::new(Counter::new());
31
32     // Create threads to increment the counter
33     let mut handles = vec![];
34     for _ in 0..10 {
35         let counter_clone = Arc::clone(&counter);
36         let handle = thread::spawn(move || {
37             counter_clone.increment();
38         });
39         handles.push(handle);
40     }
41
42     // Wait for all threads to complete
43     for handle in handles {
44         handle.join().unwrap();
45     }
46
47     // Print the final value of the counter
48     println!("Final counter value: {}", counter.get_count());
49 }
50

```

3. Run your program to demonstrate the use of 'RwLock' for concurrent read and exclusive write access:

`cargo run`

Extra Challenges:

1. ****High Concurrency Test:**** Create a scenario with a high number of concurrent readers and a few writers. Measure and

compare the performance against a version using 'Mutex' instead of 'RwLock'.

2. ****Implement a Thread-Safe Map:**** Using 'RwLock', implement a thread-safe map that supports concurrent reads and exclusive writes. Add functionality for inserting, removing, and querying items by key.
3. ****Explore Starvation:**** Discuss the potential for reader or writer starvation with 'RwLock' and how Rust's implementation addresses (or does not address) this issue.

Exercise 18.5: Creating a Thread-Safe Linked List

Objective: Develop a thread-safe linked list in Rust by utilizing 'Arc' and 'Mutex' to manage shared access and modification across multiple threads.

Instructions:

1. Create a new Rust project named `thread_safe_linked_list`:

```
cargo new thread_safe_linked_list  
cd thread_safe_linked_list
```

2. Modify `src/main.rs` to define a thread-safe linked list using 'Arc' for shared ownership and 'Mutex' for mutual exclusion:



Rust code

```

1  use std::sync::{Arc, Mutex};
2  use std::thread;
3
4  struct ListNode {
5      value: i32,
6      next: Option<Arc<Mutex<ListNode>>>,
7  }
8
9  struct ThreadSafeLinkedList {
10     head: Option<Arc<Mutex<ListNode>>>,
11 }
12
13 impl ThreadSafeLinkedList {
14     // Function to create a new empty list
15     fn new() -> Self {
16         ThreadSafeLinkedList { head: None }
17     }
18
19     // Function to append a node to the list
20     fn append(&mut self, value: i32) {
21         let new_node = Arc::new(Mutex::new(ListNode { value, next: None }));
22         match self.head {
23             Some(ref head) => {
24                 let mut current = head.clone();
25                 while let Some(next) = current.lock().unwrap().next.clone() {
26                     current = next;
27                 }
28                 current.lock().unwrap().next = Some(new_node);
29             },
30             None => {
31                 self.head = Some(new_node);
32             },
33         }
34     }
35 }
36
37 fn main() {
38     let list = Arc::new(Mutex::new(ThreadSafeLinkedList::new()));
39
40     // Create threads to append elements to the list
41     let mut handles = vec![];
42     for i in 0..10 {
43         let list_clone = Arc::clone(&list);
44         let handle = thread::spawn(move || {
45             list_clone.lock().unwrap().append(i);
46         });
47         handles.push(handle);
48     }
49
50     // Wait for all threads to complete
51     for handle in handles {
52         handle.join().unwrap();
53     }
54
55     // Implement function to print list elements, demonstrating thread-safe access
56     // Note: This is a simplification. Printing in a real application might require additional
57     // ↪ synchronization.
58 }

```

3. Run your program to observe the thread-safe behavior of the linked list as it's modified by multiple threads:

```
cargo run
```

Extra Challenges:

1. ****Implement Safe Traversal:**** Design and implement a method to safely traverse and print the elements of the list from multiple threads, ensuring that access to nodes is properly synchronized.
2. ****Fine-Grained Locking:**** Research and implement a version of the linked list that uses more granular locking mechanisms to improve performance during concurrent access.
3. ****Concurrency Testing:**** Write tests to verify the thread safety of your linked list implementation. Consider edge cases such as concurrent appends, removals, and traversals.

Chapter 19

Concurrency in Rust



Exercise 19.1: Practicing Basic Concurrency in Rust

Objective: Gain hands-on experience with Rust's concurrency model by implementing a simple program that uses threads to perform parallel computations.

Instructions:

1. Create a new Rust project named `basic_concurrency`:

```
cargo new basic_concurrency
cd basic_concurrency
```

2. Modify `src/main.rs` to spawn multiple threads that perform a simple calculation or task in parallel:

**Rust code**

```

1  use std::thread;
2  use std::time::Duration;
3
4  fn main() {
5      // Vector to hold the thread handles
6      let mut handles = Vec::new();
7
8      for i in 0..4 { // Spawn 4 threads
9          let handle = thread::spawn(move || {
10             println!("Thread {} is executing", i);
11             // Simulate some work by sleeping the thread for 2 seconds
12             thread::sleep(Duration::from_secs(2));
13             println!("Thread {} has finished executing", i);
14         });
15         handles.push(handle);
16     }
17
18     // Wait for all threads to complete
19     for handle in handles {
20         handle.join().unwrap();
21     }
22
23     println!("All threads have completed.");
24 }

```

3. Run your program to observe how Rust manages the execution of concurrent threads:

`cargo run`

Extra Challenges:

1. **Thread Communication:** Modify the program to demonstrate thread communication. Use channels to send data from your threads back to the main thread and print out the results.
2. **Error Handling in Threads:** Implement error handling within threads and demonstrate how to safely return an error to the main thread for processing.
3. **Exploring Thread Pool:** Research and implement a simple thread pool manager that limits the number of threads running simultaneously. This will introduce the concept of reusing threads for multiple tasks.

Exercise 19.2: Inter-Thread Communication with Channels

Objective: Learn to use Rust's channels to facilitate safe and efficient communication between threads.

Instructions:

1. Create a new Rust project named `channel_communication`:

```
cargo new channel_communication
cd channel_communication
```

2. Modify `src/main.rs` to create a producer-consumer scenario where multiple threads produce data that is consumed by a separate thread through a channel:



Rust code

```
1 use std::sync::mpsc;
2 use std::thread;
3 use std::time::Duration;
4
5 fn main() {
6     let (tx, rx) = mpsc::channel();
7
8     // Spawn producer threads
9     for i in 0..4 {
10         let tx_clone = mpsc::Sender::clone(&tx);
11         thread::spawn(move || {
12             let message = format!("Message from thread {}", i);
13             tx_clone.send(message).unwrap();
14             println!("Thread {} has sent its message", i);
15         });
16     }
17
18     // Consume messages in the main thread
19     for _ in 0..4 {
20         let received = rx.recv().unwrap();
21         println!("Main thread received: {}", received);
22     }
23 }
```

3. Run your program to observe the communication between threads via the channel:

```
cargo run
```

Extra Challenges:

1. ****Multiple Consumers:**** Extend the program to include multiple consumer threads that receive messages from the producers. Implement a way to distribute messages among consumers evenly.
2. ****Non-blocking Communication:**** Modify your program to use `try_recv` for non-blocking message retrieval. Discuss how this affects the program's behavior and responsiveness.
3. ****Complex Data Transmission:**** Create a more complex scenario where threads communicate using custom data structures (e.g., structs) instead of simple strings. This could simulate a real-world use case, like processing tasks or commands.

✍ Exercise 19.3: Advanced Channel Communication with Multiple Producers and Consumers

Objective: Implement a more complex inter-thread communication pattern using channels in Rust, featuring multiple producer and consumer threads.

Instructions:

1. Create a new Rust project named `multi_prod_cons`:

```
cargo new multi_prod_cons
cd multi_prod_cons
```

2. Modify `src/main.rs` to implement a system with multiple producers and consumers using Rust's channels:

Rust code

```
1 use std::sync::mpsc;
2 use std::thread;
3 use std::time::Duration;
4
5 fn main() {
6     let (tx, rx) = mpsc::channel();
7     let rx = std::sync::Arc::new(std::sync::Mutex::new(rx));
8
9     // Create multiple producer threads
10    for i in 0..5 { // 5 producers
11        let tx_clone = mpsc::Sender::clone(&tx);
12        thread::spawn(move || {
13            let message = format!("Message from producer {}", i);
14            tx_clone.send(message).unwrap();
15            println!("Producer {} sent its message", i);
16        });
17    }
18
19    // Create multiple consumer threads
20    let num_consumers = 3;
21    let mut handles = vec![];
22    for i in 0..num_consumers {
23        let rx_clone = rx.clone();
24        let handle = thread::spawn(move || {
25            while let Ok(message) = rx_clone.lock().unwrap().recv() {
26                println!("Consumer {} received: {}", i, message);
27            }
28        });
29        handles.push(handle);
30    }
31
32    // Wait for all consumers to complete
33    for handle in handles {
34        handle.join().unwrap();
35    }
36 }
```

3. Run your program to observe the interaction between multiple producers and consumers through the channel:

```
cargo run
```

Extra Challenges:

1. ****Balanced Message Distribution:**** Implement a strategy to ensure messages are evenly distributed among consumers, even under varying workloads.
2. ****Graceful Shutdown:**** Design a mechanism for gracefully shutting down all producers and consumers. Ensure all messages are processed before shutdown.
3. ****Error Handling:**** Add comprehensive error handling for all possible failures in message transmission and reception, including dropped senders or receivers.

✍ Exercise 19.4: Using Atomics for Lock-Free Concurrency

Objective: Learn to use atomic types in Rust for lock-free concurrent programming. Implement a simple counter that can be safely incremented by multiple threads using atomic operations.

Instructions:

1. Create a new Rust project named `atomic_counter`:

```
cargo new atomic_counter
cd atomic_counter
```

2. Modify `src/main.rs` to implement a thread-safe atomic counter:



Rust code

```
1 use std::sync::atomic::{AtomicI32, Ordering};
2 use std::sync::Arc;
3 use std::thread;
4
5 fn main() {
6     let counter = Arc::new(AtomicI32::new(0));
7
8     let mut handles = vec![];
9
10    for _ in 0..10 { // Spawn 10 threads to increment the counter
11        let counter_clone = Arc::clone(&counter);
12        let handle = thread::spawn(move || {
13            for _ in 0..1000 {
14                // Use fetch_add to increment the atomic counter
15                counter_clone.fetch_add(1, Ordering::SeqCst);
16            }
17        });
18        handles.push(handle);
19    }
20
21    // Wait for all threads to complete
22    for handle in handles {
23        handle.join().unwrap();
24    }
25
26    println!("Resulting counter value: {}", counter.load(Ordering::SeqCst));
27 }
```

3. Run your program to observe the atomic counter being safely incremented across multiple threads:

```
cargo run
```

Extra Challenges:

1. **Different Orderings:** Experiment with different memory orderings ('Ordering::Relaxed', 'Ordering::Acquire', 'Ordering::Release', etc.) and discuss their impact on performance and correctness.
2. **Implement a Lock-Free Stack:** Use atomic pointers ('AtomicPtr') to implement a basic lock-free stack. Ensure that it can be safely used from multiple threads.
3. **Atomic Flags for Synchronization:** Implement a simple synchronization mechanism using 'AtomicBool' where one thread sets a flag to signal other threads to perform or stop performing an action.

Exercise 19.5: Getting Started with Async/Await in Rust

Objective: Learn the basics of asynchronous programming in Rust by writing a simple asynchronous function that simulates a network request.

Instructions:

1. Create a new Rust project named `async_example`:

```
cargo new async_example
cd async_example
```

2. Modify your `Cargo.toml` to include the 'tokio' runtime, which provides an asynchronous runtime for Rust:

```
[dependencies]
tokio = { version = "1", features = ["full"] }
```

3. Update `src/main.rs` to use 'tokio' and write an asynchronous function that simulates fetching data from a network:



Rust code

```
1 use tokio::time::{sleep, Duration};
2
3 #[tokio::main]
4 async fn main() {
5     println!("Fetching data...");
6     let data = fetch_data().await;
7     println!("Data fetched: {}", data);
8 }
9
10 async fn fetch_data() -> String {
11     // Simulate a network request by sleeping for 2 seconds
12     sleep(Duration::from_secs(2)).await;
13     "Data from the network".to_string()
14 }
```

4. Run your program using 'cargo run' and observe the asynchronous operation:

```
cargo run
```

Extra Challenges:

1. **Parallel Execution:** Use 'tokio::join' to fetch multiple pieces of data in parallel, demonstrating the efficiency of asynchronous programming.
2. **Error Handling:** Implement error handling for your asynchronous functions, using 'Result<T, E>' to handle potential errors in network requests.
3. **Implement a Timer:** Use asynchronous programming to create a simple timer that counts down from 10 to 0, printing the remaining time at each second.

✍ Exercise 19.6: Building an Asynchronous HTTP Client

Objective: Use asynchronous Rust to create a simple HTTP client that makes concurrent requests to multiple URLs and processes their responses.

Instructions:

1. Create a new Rust project named `async_http_client`:

```
cargo new async_http_client
cd async_http_client
```

2. Modify your `Cargo.toml` to include dependencies for 'tokio' and 'reqwest':

```
[dependencies]
tokio = { version = "1", features = ["full"] }
reqwest = "0.11"
```

3. Update `src/main.rs` to perform asynchronous GET requests to multiple URLs concurrently:



Rust code

```
1 use tokio::task;
2 use reqwest::Error;
3
4 #[tokio::main]
5 async fn main() -> Result<(), Error> {
6     let urls = vec![
7         "http://example.com",
8         "http://example.org",
9         "http://example.net",
10    ];
11
12    let fetches = urls.into_iter().map(|url| {
13        tokio::spawn(async move {
14            let resp = reqwest::get(url).await?;
15            resp.text().await
16        })
17    });
18
19    // Use `futures::future::join_all` to wait for all fetches to complete
20    let results = futures::future::join_all(fetches).await;
21
22    for result in results {
23        match result {
24            Ok(Ok(text)) => println!("Fetched:\n{}", text),
25            Ok(Err(e)) => eprintln!("Request failed: {}", e),
26            Err(e) => eprintln!("Task failed: {}", e),
27        }
28    }
29
30    Ok(())
31 }
```

4. Run your program to see the concurrent HTTP requests in action:

```
cargo run
```

Extra Challenges:

1. ****Response Handling:**** Modify the program to include more sophisticated response handling, such as parsing JSON responses.
2. ****Error Handling and Timeouts:**** Implement comprehensive error handling for failed requests and add timeouts to ensure that requests do not hang indefinitely.
3. ****Rate Limiting:**** Introduce a rate limiting mechanism to avoid overwhelming the server with too many concurrent requests.

Exercise 19.7: Building an Asynchronous API Server with Warp

Objective: Use the 'warp' framework to create an asynchronous API server that responds to GET requests with JSON data.

Instructions:

1. Create a new Rust project named `async_api_server`:

```
cargo new async_api_server
cd async_api_server
```

2. Modify your `Cargo.toml` to include dependencies for 'tokio', 'warp', and 'serde' for JSON serialization:

```
[dependencies]
tokio = { version = "1", features = ["full"] }
warp = "0.3"
serde = { version = "1.0", features = ["derive"] }
serde_json = "1.0"
```

3. Update `src/main.rs` to define an asynchronous API server that handles GET requests and responds with a simple JSON object:



Rust code

```
1 use warp::Filter;
2 use serde::{Serialize, Deserialize};
3
4 #[derive(Serialize, Deserialize)]
5 struct ApiResponse {
6     message: String,
7 }
8
9 async fn hello_world() -> Result<impl warp::Reply, warp::Rejection> {
10     Ok(warp::reply::json(&ApiResponse {
11         message: "Hello, World!".to_string(),
12     }))
13 }
14
15 #[tokio::main]
16 async fn main() {
17     // Define the API route
18     let hello_route = warp::path!("hello")
19         .and(warp::get())
20         .and_then(hello_world);
21
22     // Start the warp server
23     warp::serve(hello_route)
24         .run(([127, 0, 0, 1], 3030))
25         .await;
26 }
```

4. Run your server and use a tool like 'curl' or Postman to make a GET request to 'http://localhost:3030/hello' to see the JSON response:

`cargo run`

Extra Challenges:

1. **Parameterized Routes:** Extend the server to accept dynamic routes, such as `/hello/name`, and customize the JSON response based on the name provided in the URL.
2. **Post Requests:** Implement a route that accepts POST requests and processes JSON data sent in the request body. Respond with a modified version of the received JSON.
3. **Error Handling:** Add comprehensive error handling to the server, ensuring that it can gracefully handle invalid routes, unsupported methods, and malformed JSON data in request bodies.

Chapter 20

Using Rust in containers



Exercise 20.1: Containerizing a Rust Web Server Application with Podman

Objective: Develop and containerize a simple Rust web server application using Podman. The exercise involves creating the application, containerizing it with a libc-based distribution, and then optimizing the container using a musl-based distribution with a multi-stage Dockerfile.

Instructions:

Part 1: Creating the Rust Web Server

1. **Setup and Research:** - Familiarize yourself with basic web server development in Rust. We will use 'warp' as our web framework for its simplicity and efficiency in building API servers.
2. **Implementation:** - Create a new Rust project named `rust_web_server`:

```
cargo new rust_web_server
cd rust_web_server
```

- Modify your `Cargo.toml` to include dependencies for 'warp' and 'tokio':

```
[dependencies]
warp = "0.3"
tokio = { version = "1", features = ["full"] }
```

- Implement a simple "Hello, World!" web server in `src/main.rs`:

**Rust code**

```

1 use warp::Filter;
2
3 #[tokio::main]
4 async fn main() {
5     let hello = warp::path!("hello" / "world")
6         .map(|| warp::reply::html("Hello, World!"));
7
8     warp::serve(hello)
9         .run(([127, 0, 0, 1], 3030))
10        .await;
11 }

```

3. **Testing:** - Run your Rust web server locally and test it by visiting 'http://localhost:3030/hello/world' in your web browser or using 'curl':

```
cargo run
```

Part 2: Containerizing with libc-based Distribution

1. **Dockerfile Creation:** - Create a 'Dockerfile' in the root of your project:

```

FROM rust:1.56 as builder
WORKDIR /usr/src/rust_web_server
COPY . .
RUN cargo install --path .

FROM debian:buster-slim
COPY --from=builder /usr/local/cargo/bin/rust_web_server /usr/local/bin/rust_web_server
CMD ["rust_web_server"]

```

2. **Building and Running the Container:** - Use Podman to build and run the container:

```

podman build -t rust_web_server .
podman run -p 3030:3030 rust_web_server

```

Part 3: Optimizing with musl-based Distribution and Multi-stage Dockerfile

1. **Multi-stage Dockerfile with musl:** - Modify the 'Dockerfile' for musl optimization:

```

FROM rust:1.56-alpine as builder
WORKDIR /usr/src/rust_web_server
RUN apk add --no-cache musl-dev
COPY . .
RUN cargo build --release

FROM alpine:latest
COPY --from=builder /usr/src/rust_web_server/target/release/rust_web_server /usr/local/bin/
CMD ["rust_web_server"]

```

2. **Building and Running the Optimized Container:** - Build and run the optimized container:

```

podman build -t rust_web_server_musl -f Dockerfile.musl .
podman run -p 3030:3030 rust_web_server_musl

```


Chapter 21

Cross compiling in Rust



In this lab, we will explore cross-compiling in Rust alongside practicing how to make use of some OS functions. Cross-compilation allows you to compile your Rust program on a host system (e.g., Windows or macOS) to run on a different target system (e.g., ARM-based Linux systems). We will start by setting up a basic Rust project for cross-compilation, then create a small program that will display on what kind of platform the program is actually running.

Setup for Cross-Compiling:

1. Ensure Rust is installed on your system. Use `rustup` to manage Rust versions and toolchains.
2. Install the `cross` tool using `cargo install cross` for simplifying cross-compilation tasks.
3. Identify the target platform for cross-compilation (e.g., `aarch64-unknown-linux-gnu`) and add it using `rustup target add <target>`.
4. Use `cross build --target <target>` to compile your project for the specified target architecture.

Exercise 21.1: Cross-compiling and OS functions

1. Setup cross-compiling in Rust using `cargo-cross`
2. Create a RUST binary package that will:
 - Display the text `Hello World`
 - Display the OS name
 - Display the CPU architecture
3. Cross compile your program for `aarch64-unknown-linux-gnu` and another target of your choice
4. If possible run your binary on the real HW or using emulation like `qemu`.

Chapter 22

Advanced Topics



✍ Exercise 22.1: Calling a C Function from Rust

Objective: Familiarize with Rust FFI by calling a simple C function for arithmetic operations.

Instructions:

1. Create a new Rust project named `ffi_basics`:

```
cargo new ffi_basics
cd ffi_basics
```

2. Write a C function in a separate file:

```
// math_ops.c
#include <stdint.h>

int32_t add(int32_t a, int32_t b) {
    return a + b;
}
```

3. Create a Rust binding for the C function:



Rust code

```
1 extern "C" {
2     fn add(a: i32, b: i32) -> i32;
3 }
4
5 fn main() {
6     unsafe {
7         println!("3 + 4 = {}", add(3, 4));
8     }
9 }
```

4. Compile the C code and link it with Rust:

```
gcc -c math_ops.c -o math_ops.o
ar rcs libmath_ops.a math_ops.o
```

Update `Cargo.toml` to include the library:

```
[build-dependencies]
cc = "1.0"
```

```
[dependencies]
```

```
[build]
script = "build.rs"
```

And `build.rs`:



Rust code

```
1 fn main() {
2     println!("cargo:rustc-link-search=.");
3     println!("cargo:rustc-link-lib=static=math_ops");
4 }
```

5. Run your Rust application:

```
cargo run
```

Exercise 22.2: Interacting with C Structs from Rust

Objective: Deepen FFI knowledge by manipulating C structs in Rust.

Instructions:

1. Extend the C library with a struct and functions that operate on it. Create `person.c` and `person.h`.
2. Use `bindgen` in `build.rs` to generate Rust bindings for the C struct and functions.
3. Write Rust code to interact with the C struct, modifying and reading its fields.

Exercise 22.3: Creating Safe Wrappers around Unsafe FFI Code

Objective: Enhance safety by writing safe Rust wrappers around unsafe FFI interactions.

Instructions:

1. Identify unsafe FFI patterns in your Rust project.
2. Design and implement safe wrapper functions or structs in Rust that encapsulate the unsafe FFI calls, handling errors and resource management correctly.
3. Use your safe wrappers in a Rust application, ensuring safety guarantees are maintained.

Exercise 22.4: Dereferencing Raw Pointers

Objective: Understand how to safely use raw pointers in Rust by creating and dereferencing raw pointers to read and modify data.

Instructions:

1. Create a new Rust project named `raw_pointers`:

```
cargo new raw_pointers
cd raw_pointers
```

2. Implement a function in `src/main.rs` that demonstrates creating, dereferencing, and modifying data through raw pointers:



Rust code

```
1 fn main() {
2     let mut x = 10;
3     let raw_ptr = &mut x as *mut i32;
4
5     unsafe {
6         // Safely dereference raw_ptr to read and modify data
7         println!("Before: {}", *raw_ptr);
8         *raw_ptr += 5;
9         println!("After: {}", *raw_ptr);
10    }
11 }
```

3. Discuss the safety implications of using raw pointers and the necessity of the 'unsafe' block.

Exercise 22.5: Using Unsafe Functions

Objective: Learn to define and use unsafe functions in Rust, specifically focusing on scenarios where using such functions is unavoidable.

Instructions:

1. In the same Rust project `raw_pointers`, add a new unsafe function that takes a raw pointer as an argument and modifies the data it points to.



Rust code

```
1 unsafe fn increment(ptr: *mut i32) {  
2     *ptr += 1;  
3 }
```

2. From the 'main' function, create a raw pointer and use it to call your unsafe function:



Rust code

```
1 fn main() {  
2     let mut x = 10;  
3     let raw_ptr = &mut x as *mut i32;  
4  
5     unsafe {  
6         increment(raw_ptr);  
7         println!("x has been incremented to: {}", *raw_ptr);  
8     }  
9 }
```

3. Reflect on the use cases for unsafe functions and the importance of ensuring safety within the 'unsafe' block.

Chapter 23

Closing and Evaluation Survey



23.1	Evaluation Survey	124
------	-------------------	-----

23.1 Evaluation Survey

Evaluation Survey

Thank you for taking this course brought to you by **The Linux Foundation**. Your comments are important to us and we take them seriously, both to measure how well we fulfilled your needs and to help us improve future sessions.

- Please Evaluate your training using the link your instructor will provide.
- Please be sure to check the spelling of your name and use correct capitalization as this is how your name will appear on the certificate.
- This information will be used to generate your **Certificate of Completion**, which will be sent to the email address you have supplied, so make sure it is correct.

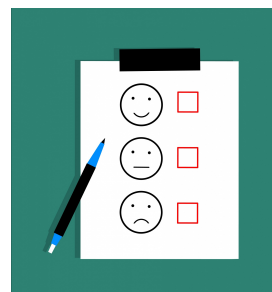


Figure 23.1: Course Survey

Appendices

