



NOVEL PROGRAMMING LANGUAGES

Syllabus

Catalog Description

Higher education level	<i>First (Undergraduate)</i>
Knowledge field	<i>Information Technologies</i>
Profession	<i>121 Software Engineering</i>
Curriculum	<i>Software Engineering of Intelligent Cyberphysical Systems in Energy Industry</i>
Course status	<i>Elective</i>
Form of training	<i>Full-time</i>
Grade, term	<i>Fourth grade, fall semester</i>
Credits (hours)	<i>4 credits / 120 hours (36 hours of lectures, 18 hours of practice, 66 hours of individual assignments)</i>
Term control	<i>Exam, modular test</i>
Schedule	<i>http://schedule.kpi.ua/</i>
Teaching language	<i>Ukrainian/English</i>
Instructors	Lecturer: <i>DSc. (Econ), professor Andrii Sihaiov</i> Seminars: Laboratory work: <i>Andrii Sihaiov</i>
URL	<i>GitHub Classroom, eCampus</i>

Course Program

1 Course description, aim, subject, and expected outcomes

Why future specialist should study this course?

It wasn't always so clear, but the Rust programming language is fundamentally about empowerment: no matter what kind of code you are writing now, Rust empowers you to reach farther, to program with confidence in a wider variety of domains than you did before.

Take, for example, "systems-level" work that deals with low-level details of memory management, data representation, and concurrency. Traditionally, this realm of programming is seen as arcane, accessible only to a select few who have devoted the necessary years learning to avoid its infamous pitfalls. And even those who practice it do so with caution, lest their code be open to exploits, crashes, or corruption.

Rust breaks down these barriers by eliminating the old pitfalls and providing a friendly, polished set of tools to help you along the way. Programmers who need to "dip down" into lower-level control can do so with Rust, without taking on the customary risk of crashes or security holes, and without having to learn the fine points of a fickle toolchain. Better yet, the language is designed to guide you naturally towards reliable code that is efficient in terms of speed and memory usage.

Programmers who are already working with low-level code can use Rust to raise their ambitions. For example, introducing parallelism in Rust is a relatively low-risk operation: the compiler will catch the classical mistakes for you. And you can tackle more aggressive optimizations in your code with the confidence that you won't accidentally introduce crashes or vulnerabilities.

But Rust isn't limited to low-level systems programming. It's expressive and ergonomic enough to make CLI apps, web servers, and many other kinds of code quite pleasant to write — you'll find simple examples

of both later in the book. Working with Rust allows you to build skills that transfer from one domain to another; you can learn Rust by writing a web app, then apply those same skills to target your Raspberry Pi.

This course fully embraces the potential of Rust to empower its users. It's a friendly and approachable text intended to help you level up not just your knowledge of Rust, but also your reach and confidence as a programmer in general. So dive in, get ready to learn—and welcome to the Rust community!

Course Aim. To familiarize students with modern Rust programming language.

Course Subject. An overview of Rust, including:

- ownership, borrowing, generic types, lifetimes, and traits;
- guaranteed code safety;
- testing, error handling, and effective refactoring;
- smart pointers, concurrency, patterns and matching;
- working with Cargo and Crates.io;
- advanced features of Unsafe Rust.

Expected Outcomes.

Professional Competencies.

FC 16. To know script and declarative programming languages.

Program Learning Outcomes.

PLO 15. To choose programming languages and development technologies in a substantiated way for task solving of creating and maintaining software.

PLO 32. To apply in practice fundamental concepts, paradigms and basic functioning principles of linguistic, instrumental and computing tools of software engineering.

2 Course prerequisites (Where the course fits into our curriculum)

The course is taken in fall semester of final year. Discrete Math, Programming Basics, and Systems Software Architecture are the prerequisites. There is no required course that has this course as a prerequisite.

3 Course contents

1. Getting Started. Programming a Guessing Game. Common Programming Concepts.
2. Understanding Ownership.
3. Using Structs to Structure Related Data.
4. Enums and Pattern Matching.
5. Managing Growing Projects with Packages, Crates, and Modules.
6. Common Collections.
7. Error Handling.
8. Generic Types, Traits, and Lifetimes.
9. Writing Automated Tests.
10. An I/O Project: Building a Command Line Program.
11. Functional Language Features: Iterators and Closures.
12. More About Cargo and Crates.io.

13. *Smart Pointers.*
14. *Fearless Concurrency.*
15. *Object-Oriented Programming Features of Rust.*
16. *Patterns and Matching.*
17. *Advanced Features.*
18. *Final Project: Building a Multithreaded Web Server.*

4 Course textbooks and materials

Required reading:

Klabnik, S., Nichols, C. *The Rust Programming Language: San Francisco, CA: No Starch Press, 2019. 560 c.* URL: <http://libgen.rs/book/index.php?md5=83D9E4EBE32F219D6436A438076216A3>

Optional reading:

1. McNamara, T. S. *Rust in Action: Shelter Island, NY: Manning Publications, 2021. 456 c.* URL: <http://libgen.rs/book/index.php?md5=38FD5C0062ED46D7574BE9987952D655>
2. *Learn Rust in Y Minutes.* URL: <https://learnxinyminutes.com/docs/rust/> , *Learn X in Y minutes: Scenic Programming Language Tours*, (дата звернення: 31.08.21)
3. Amos. *A half-hour to learn Rust.* URL: <https://fasterthanli.me/articles/a-half-hour-to-learn-rust> , *fasterthanli.me*, (дата звернення: 02.01.21)
4. Biedert, R. *Rust Language Cheat Sheet.* URL: <https://cheats.rs/> , *Rust Language Cheat Sheet*, (дата звернення: 25.04.21)
5. *Rust Design Patterns.* URL: <https://rust-unofficial.github.io/patterns/> , *Rust Design Patterns*, (дата звернення: 03.01.21)
6. Nethercote, N. *The Rust Performance Book.* URL: <https://nnethercote.github.io/perf-book/title-page.html> , *The Rust Performance Book*, (дата звернення: 25.04.21)
7. Messier, R. *Beginning Rust Programming: Indianapolis, IN: Wiley, 2021. 416 c.* URL: <http://libgen.rs/book/index.php?md5=0FB67E531BF1CB7B188BF84F817291AE>
8. Wolverson, H. *Hands-on Rust: Effective Learning through 2D Game Development and Play: Raleigh, NC: Pragmatic Bookshelf, 2021. 332 c.* URL: <http://libgen.rs/book/index.php?md5=9C11FE8FF7D64EE9E10BBC8817EE3882>

Educational Content

5 Pedagogical advice

1. *Getting Started. Programming a Guessing Game. Common Programming Concepts.*
 - 1.1. *Rust Installation.*
 - 1.1.1. *Installing Rust on Linux or macOS.*
 - 1.1.2. *Installing rustup on Windows.*
 - 1.1.3. *Updating and Uninstalling.*
 - 1.1.4. *Troubleshooting.*
 - 1.1.5. *Local Documentation.*
 - 1.2. *Programming a Guessing Game. Setting Up a New Project.*
 - 1.3. *Processing a Guess.*

- 1.3.1. *Storing Values with Variables.*
- 1.3.2. *Handling Potential Failure with the Result Type.*
- 1.3.3. *Printing Values with println! Placeholders.*
- 1.3.4. *Testing the First Part*
- 1.4. *Generating a Secret Number.*
 - 1.4.1. *Using a Crate to Get More Functionality.*
 - 1.4.2. *Generating a Random Number.*
- 1.5. *Comparing the Guess to the Secret Number.*
- 1.6. *Allowing Multiple Guesses with Looping.*
 - 1.6.1. *Quitting After a Correct Guess.*
 - 1.6.2. *Handling Invalid Input.*
- 1.7. *Common Programming Concepts. Variables and Mutability.*
 - 1.7.1. *Differences Between Variables and Constants.*
 - 1.7.2. *Shadowing.*
- 1.8. *Data Types.*
 - 1.8.1. *Scalar Types.*
 - 1.8.2. *Compound Types.*
- 1.9. *Functions.*
 - 1.9.1. *Function Parameters.*
 - 1.9.2. *Statements and Expressions in Function Bodies.*
 - 1.9.3. *Functions with Return Values.*
- 1.10. *Comments.*
- 1.11. *Control Flow.*
 - 1.11.1. *if Expressions.*
 - 1.11.2. *Repetition with Loops.*
- 2. *Understanding Ownership.*
 - 2.1. *What Is Ownership?*
 - 2.1.1. *Ownership Rules.*
 - 2.1.2. *Variable Scope.*
 - 2.1.3. *The String Type.*
 - 2.1.4. *Memory and Allocation.*
 - 2.1.5. *Ownership and Functions.*
 - 2.1.6. *Return Values and Scope.*
 - 2.2. *References and Borrowing.*
 - 2.2.1. *Mutable References.*
 - 2.2.2. *Dangling References.*
 - 2.2.3. *The Rules of References.*

- 2.3. *The Slice Type.*
 - 2.3.1. *String Slices.*
 - 2.3.2. *Other Slices.*
- 3. *Using Structs to Structure Related Data.*
 - 3.1. *Defining and Instantiating Structs.*
 - 3.1.1. *Using the Field Init Shorthand When Variables and Fields Have the Same Name.*
 - 3.1.2. *Creating Instances from Other Instances with Struct Update Syntax.*
 - 3.1.3. *Using Tuple Structs Without Named Fields to Create Different Types.*
 - 3.1.4. *Unit-Like Structs Without Any Fields.*
 - 3.2. *An Example Program Using Structs.*
 - 3.2.1. *Refactoring with Tuples.*
 - 3.2.2. *Refactoring with Structs: Adding More Meaning.*
 - 3.2.3. *Adding Useful Functionality with Derived Traits.*
 - 3.3. *Method Syntax.*
 - 3.3.1. *Defining Methods.*
 - 3.3.2. *Methods with More Parameters.*
 - 3.3.3. *Associated Functions.*
 - 3.3.4. *Multiple impl Blocks.*
- 4. *Enums and Pattern Matching.*
 - 4.1. *Defining an Enum.*
 - 4.1.1. *Enum Values.*
 - 4.1.2. *The Option Enum and Its Advantages over Null Values.*
 - 4.2. *The match Control Flow Operator.*
 - 4.2.1. *Patterns That Bind to Values.*
 - 4.2.2. *Matching with Option<T>.*
 - 4.2.3. *Matches Are Exhaustive.*
 - 4.2.4. *The _ Placeholder.*
 - 4.3. *Concise Control Flow with if let.*
- 5. *Managing Growing Projects with Packages, Crates, and Modules.*
 - 5.1. *Packages and Crates.*
 - 5.2. *Defining Modules to Control Scope and Privacy.*
 - 5.3. *Paths for Referring to an Item in the Module Tree.*
 - 5.3.1. *Exposing Paths with the pub Keyword.*
 - 5.3.2. *Starting Relative Paths with super.*
 - 5.3.3. *Making Structs and Enums Public.*
 - 5.4. *Bringing Paths into Scope with the use Keyword.*
 - 5.4.1. *Creating Idiomatic use Paths.*

- 5.4.2. *Providing New Names with the as Keyword.*
- 5.4.3. *Re-exporting Names with pub use.*
- 5.4.4. *Using External Packages.*
- 5.4.5. *Using Nested Paths to Clean Up Large use Lists.*
- 5.4.6. *The Glob Operator.*

5.5. *Separating Modules into Different Files.*

6. *Common Collections.*

6.1. *Storing Lists of Values with Vectors.*

- 6.1.1. *Creating a New Vector.*
- 6.1.2. *Updating a Vector.*
- 6.1.3. *Dropping a Vector Drops Its Elements.*
- 6.1.4. *Reading Elements of Vectors.*
- 6.1.5. *Iterating over the Values in a Vector.*
- 6.1.6. *Using an Enum to Store Multiple Types.*

6.2. *Storing UTF-8 Encoded Text with Strings.*

- 6.2.1. *What Is a String?*
- 6.2.2. *Creating a New String.*
- 6.2.3. *Updating a String.*
- 6.2.4. *Indexing into Strings.*
- 6.2.5. *Slicing Strings.*
- 6.2.6. *Methods for Iterating over Strings.*
- 6.2.7. *Strings Are Not So Simple.*

6.3. *Storing Keys with Associated Values in Hash Maps.*

- 6.3.1. *Creating a New Hash Map.*
- 6.3.2. *Hash Maps and Ownership.*
- 6.3.3. *Accessing Values in a Hash Map.*
- 6.3.4. *Updating a Hash Map.*
- 6.3.5. *Hashing Functions.*

7. *Error Handling.*

7.1. *Unrecoverable Errors with panic!*

- 7.1.1. *Using a panic! Backtrace.*

7.2. *Recoverable Errors with Result.*

- 7.2.1. *Matching on Different Errors.*
- 7.2.2. *Shortcuts for Panic on Error: unwrap and expect.*
- 7.2.3. *Propagating Errors.*

7.3. *To panic! or Not to panic!*

- 7.3.1. *Examples, Prototype Code, and Tests.*

- 7.3.2. *Cases in Which You Have More Information Than the Compiler.*
 - 7.3.3. *Guidelines for Error Handling.*
 - 7.3.4. *Creating Custom Types for Validation.*
 - 8. *Generic Types, Traits, and Lifetimes.*
 - 8.1. *Removing Duplication by Extracting a Function.*
 - 8.2. *Generic Data Types.*
 - 8.2.1. *In Function Definitions.*
 - 8.2.2. *In Struct Definitions.*
 - 8.2.3. *In Enum Definitions.*
 - 8.2.4. *Decision Trees.*
 - 8.2.5. *In Method Definitions.*
 - 8.2.6. *Performance of Code Using Generics.*
 - 8.3. *Traits: Defining Shared Behavior.*
 - 8.3.1. *Defining a Trait.*
 - 8.3.2. *Implementing a Trait on a Type.*
 - 8.3.3. *Default Implementations.*
 - 8.3.4. *Traits as Parameters.*
 - 8.3.5. *Returning Types that Implement Traits.*
 - 8.3.6. *Fixing the largest Function with Trait Bounds.*
 - 8.3.7. *Using Trait Bounds to Conditionally Implement Methods.*
 - 8.4. *Validating References with Lifetimes.*
 - 8.4.1. *Preventing Dangling References with Lifetimes.*
 - 8.4.2. *The Borrow Checker.*
 - 8.4.3. *Generic Lifetimes in Functions.*
 - 8.4.4. *Lifetime Annotation Syntax.*
 - 8.4.5. *Lifetime Annotations in Function Signatures.*
 - 8.4.6. *Thinking in Terms of Lifetimes.*
 - 8.4.7. *Lifetime Annotations in Struct Definitions.*
 - 8.4.8. *Lifetime Elision.*
 - 8.4.9. *Lifetime Annotations in Method Definitions.*
 - 8.4.10. *The Static Lifetime.*
 - 8.5. *Generic Type Parameters, Trait Bounds, and Lifetimes Together.*
 - 9. *Writing Automated Tests.*
 - 9.1. *How to Write Tests.*
 - 9.1.1. *The Anatomy of a Test Function.*
 - 9.1.2. *Checking Results with the `assert!` Macro.*
 - 9.1.3. *Testing Equality with the `assert_eq!` and `assert_ne!` Macros.*

- 9.1.4. *Adding Custom Failure Messages.*
- 9.1.5. *Checking for Panics with `should_panic`.*
- 9.1.6. *Using `Result<T, E>` in Tests.*
- 9.2. *Controlling How Tests Are Run.*
 - 9.2.1. *Running Tests in Parallel or Consecutively.*
 - 9.2.2. *Showing Function Output.*
 - 9.2.3. *Running a Subset of Tests by Name.*
 - 9.2.4. *Ignoring Some Tests Unless Specifically Requested.*
- 9.3. *Test Organization.*
 - 9.3.1. *Unit Tests.*
 - 9.3.2. *Integration Tests.*
- 10. *An I/O Project: Building a Command Line Program.*
 - 10.1. *Accepting Command Line Arguments.*
 - 10.1.1. *Reading the Argument Values.*
 - 10.1.2. *Saving the Argument Values in Variables.*
 - 10.2. *Reading a File.*
 - 10.3. *Refactoring to Improve Modularity and Error Handling.*
 - 10.3.1. *Separation of Concerns for Binary Projects.*
 - 10.3.2. *Fixing the Error Handling.*
 - 10.3.3. *Extracting Logic from `main`.*
 - 10.3.4. *Splitting Code into a Library Crate.*
 - 10.4. *Developing the Library's Functionality with Test-Driven Development.*
 - 10.4.1. *Writing a Failing Test.*
 - 10.4.2. *Writing Code to Pass the Test.*
 - 10.5. *Working with Environment Variables.*
 - 10.5.1. *Writing a Failing Test for the Case-Insensitive `search` Function.*
 - 10.5.2. *Implementing the `search_case_insensitive` Function.*
 - 10.6. *Writing Error Messages to Standard Error Instead of Standard Output.*
 - 10.6.1. *Checking Where Errors Are Written.*
 - 10.6.2. *Printing Errors to Standard Error.*
- 11. *Functional Language Features: Iterators and Closures.*
 - 11.1. *Closures: Anonymous Functions That Can Capture Their Environment.*
 - 11.1.1. *Creating an Abstraction of Behavior with Closures.*
 - 11.1.2. *Closure Type Inference and Annotation.*
 - 11.1.3. *Storing Closures Using Generic Parameters and the `Fn` Traits.*
 - 11.1.4. *Limitations of the `Cacher` Implementation.*
 - 11.1.5. *Capturing the Environment with Closures.*

11.2. Processing a Series of Items with Iterators.

11.2.1. The Iterator Trait and the next Method.

11.2.2. Methods That Consume the Iterator.

11.2.3. Methods That Produce Other Iterators.

11.2.4. Using Closures That Capture Their Environment.

11.2.5. Creating Our Own Iterators with the Iterator Trait.

11.3. Improving Our I/O Project.

11.3.1. Removing a clone Using an Iterator.

11.3.2. Making Code Clearer with Iterator Adaptors.

11.4. Comparing Performance: Loops vs. Iterators.

12. More About Cargo and Crates.io.

12.1. Customizing Builds with Release Profiles.

12.2. Publishing a Crate to Crates.io.

12.2.1. Making Useful Documentation Comments.

12.2.2. Exporting a Convenient Public API with pub use.

12.2.3. Setting Up a Crates.io Account.

12.2.4. Adding Metadata to a New Crate.

12.2.5. Publishing to Crates.io.

12.2.6. Publishing a New Version of an Existing Crate.

12.2.7. Removing Versions from Crates.io with cargo yanks.

12.3. Cargo Workspaces.

12.3.1. Creating a Workspace.

12.3.2. Creating the Second Crate in the Workspace.

12.4. Installing Binaries from Crates.io with cargo install.

12.5. Extending Cargo with Custom Commands.

13. Smart Pointers.

13.1. Using Box<T> to Point to Data on the Heap.

13.1.1. Using a Box<T> to Store Data on the Heap.

13.1.2. Enabling Recursive Types with Boxes.

13.2. Treating Smart Pointers Like Regular References with the Deref Trait

13.2.1. Following the Pointer to the Value with the Dereference Operator.

13.2.2. Using Box<T> Like a Reference.

13.2.3. Defining Our Own Smart Pointer.

13.2.4. Treating a Type Like a Reference by Implementing the Deref Trait.

13.2.5. Implicit Deref Coercions with Functions and Methods.

13.2.6. How Deref Coercion Interacts with Mutability

13.3. Running Code on Cleanup with the Drop Trait.

- 13.3.1. *Dropping a Value Early with `std::mem::drop`.*
- 13.4. *Rc<T>, the Reference Counted Smart Pointer.*
 - 13.4.1. *Using Rc<T> to Share Data.*
 - 13.4.2. *Cloning an Rc<T> Increases the Reference Count.*
- 13.5. *RefCell<T> and the Interior Mutability Pattern.*
 - 13.5.1. *Enforcing Borrowing Rules at Runtime with RefCell<T>.*
 - 13.5.2. *Interior Mutability: A Mutable Borrow to an Immutable Value.*
 - 13.5.3. *Having Multiple Owners of Mutable Data by Combining Rc<T> and RefCell<T>.*
- 13.6. *Reference Cycles Can Leak Memory.*
 - 13.6.1. *Creating a Reference Cycle.*
 - 13.6.2. *Preventing Reference Cycles: Turning an Rc<T> into a Weak<T>.*
- 14. *Fearless Concurrency.*
 - 14.1. *Using Threads to Run Code Simultaneously.*
 - 14.1.1. *Creating a New Thread with `spawn`.*
 - 14.1.2. *Waiting for All Threads to Finish Using `join` Handles.*
 - 14.1.3. *Using `move` Closures with Threads.*
 - 14.2. *Using Message Passing to Transfer Data Between Threads.*
 - 14.2.1. *Channels and Ownership Transference.*
 - 14.2.2. *Sending Multiple Values and Seeing the Receiver Waiting.*
 - 14.2.3. *Creating Multiple Producers by Cloning the Transmitter.*
 - 14.3. *Shared-State Concurrency.*
 - 14.3.1. *Using Mutexes to Allow Access to Data from One Thread at a Time.*
 - 14.3.2. *Similarities Between RefCell<T>/Rc<T> and Mutex<T>/Arc<T>.*
 - 14.4. *Extensible Concurrency with the Sync and Send Traits.*
 - 14.4.1. *Allowing Transference of Ownership Between Threads with `Send`.*
 - 14.4.2. *Allowing Access from Multiple Threads with `Sync`.*
 - 14.4.3. *Implementing `Send` and `Sync` Manually Is Unsafe.*
- 15. *Object-Oriented Programming Features of Rust.*
 - 15.1. *Characteristics of Object-Oriented Languages.*
 - 15.1.1. *Objects Contain Data and Behavior.*
 - 15.1.2. *Encapsulation That Hides Implementation Details.*
 - 15.1.3. *Inheritance as a Type System and as Code Sharing.*
 - 15.2. *Using Trait Objects That Allow for Values of Different Types.*
 - 15.2.1. *Defining a Trait for Common Behavior.*
 - 15.2.2. *Implementing the Trait.*
 - 15.2.3. *Trait Objects Perform Dynamic Dispatch.*
 - 15.2.4. *Object Safety Is Required for Trait Objects.*

15.3. *Implementing an Object-Oriented Design Pattern.*

- 15.3.1. *Defining Post and Creating a New Instance in the Draft State.*
- 15.3.2. *Storing the Text of the Post Content.*
- 15.3.3. *Ensuring the Content of a Draft Post Is Empty.*
- 15.3.4. *Requesting a Review of the Post Changes Its State.*
- 15.3.5. *Adding the approve Method that Changes the Behavior of content.*
- 15.3.6. *Trade-offs of the State Pattern.*

16. *Patterns and Matching.*

16.1. *All the Places Patterns Can Be Used.*

- 16.1.1. *match Arms.*
- 16.1.2. *Conditional if let Expressions.*
- 16.1.3. *while let Conditional Loops.*
- 16.1.4. *for Loops.*
- 16.1.5. *let Statements.*
- 16.1.6. *Function Parameters.*

16.2. *Refutability: Whether a Pattern Might Fail to Match.*

16.3. *Pattern Syntax.*

- 16.3.1. *Matching Literals.*
- 16.3.2. *Matching Named Variables.*
- 16.3.3. *Multiple Patterns.*
- 16.3.4. *Matching Ranges of Values with the ... Syntax.*
- 16.3.5. *Destructuring to Break Apart Value.*
- 16.3.6. *Ignoring Values in a Pattern.*
- 16.3.7. *Extra Conditionals with Match Guards.*
- 16.3.8. *@ Bindings.*

17. *Advanced Features.*

17.1. *Unsafe Rust.*

- 17.1.1. *Unsafe Superpowers.*
- 17.1.2. *Dereferencing a Raw Pointer.*
- 17.1.3. *Calling an Unsafe Function or Method.*
- 17.1.4. *Accessing or Modifying a Mutable Static Variable.*
- 17.1.5. *Implementing an Unsafe Trait.*
- 17.1.6. *When to Use Unsafe Code.*

17.2. *Advanced Traits.*

- 17.2.1. *Specifying Placeholder Types in Trait Definitions with Associated Types.*
- 17.2.2. *Default Generic Type Parameters and Operator Overloading.*
- 17.2.3. *Fully Qualified Syntax for Disambiguation: Calling Methods with the Same Name.*

- 17.2.4. *Using Supertraits to Require One Trait's Functionality Within Another Trait.*
- 17.2.5. *Using the Newtype Pattern to Implement External Traits on External Types.*
- 17.3. *Advanced Types.*
 - 17.3.1. *Using the Newtype Pattern for Type Safety and Abstraction.*
 - 17.3.2. *Creating Type Synonyms with Type Aliases.*
 - 17.3.3. *The Never Type That Never Returns.*
 - 17.3.4. *Dynamically Sized Types and the Sized Trait.*
- 17.4. *Advanced Functions and Closures.*
 - 17.4.1. *Function Pointers.*
 - 17.4.2. *Returning Closures.*
- 17.5. *Macros.*
 - 17.5.1. *The Difference Between Macros and Functions.*
 - 17.5.2. *Declarative Macros with macro_rules! for General Meta-programming.*
 - 17.5.3. *Procedural Macros for Generating Code from Attributes.*
 - 17.5.4. *How to Write a Custom derive Macro.*
 - 17.5.5. *Attribute-like macros.*
 - 17.5.6. *Function-like macros.*
- 18. *Final Project: Building a Multithreaded Web Server.*
 - 18.1. *Building a Single-Threaded Web Server.*
 - 18.1.1. *Listening to the TCP Connection.*
 - 18.1.2. *Reading the Request.*
 - 18.1.3. *A Closer Look at an HTTP Request.*
 - 18.1.4. *Writing a Response.*
 - 18.1.5. *Returning Real HTML.*
 - 18.1.6. *Validating the Request and Selectively Responding.*
 - 18.1.7. *A Touch of Refactoring.*
 - 18.2. *Turning Our Single-Threaded Server into a Multithreaded Server.*
 - 18.2.1. *Simulating a Slow Request in the Current Server Implementation.*
 - 18.2.2. *Improving Throughput with a Thread Pool.*
 - 18.3. *Graceful Shutdown and Cleanup.*
 - 18.3.1. *Implementing the Drop Trait on ThreadPool.*
 - 18.3.2. *Signaling to the Threads to Stop Listening for Jobs.*

6 Individual Assignments

Students are expected to spend 3-4 hours a week outside of class on course material.

Course Rules and Assessment Policy

7 Course study rules

Students receive points relative to the correct and timely completion of coursework. The total grade consists of: 1) laboratories (programming assignments) 60%, 2) final exam 40%.

Presently there are three programming assignments, each worth up to 20% of the total grade. Student have to submit correctly fulfilled assignment during fortnight period from the date of give out to obtain full score for it, otherwise penalty points are applied but not more than 40% of the total score for laboratory work.

8 Assessment policy

How students are assessed: *modular test, programming assignments*

Calendar control: conducted twice a term to monitor the current state of compliance with the requirements of the syllabus.

Term assessment: *final exam*

Admission condition of term assessment: *all programming assignment submission, start score not less than 40 points.*

Exam scores map to the course grade according to the table:

Score	Grade
100-95	Excellent
94-85	Very Good
84-75	Good
74-65	Satisfactory
64-60	Sufficient
Less than 60	Unsatisfactory
Conditions for exam admission not met	Not allowed

9 Additional topics

- *Exam questions (see appendix).*

Syllabus:

Developed by Software Engineering in Energy Industry Department Professor, Sc. D. Andrii Sihaiov

Approved by Software Engineering in Energy Industry Department (minutes #28 on May 15, 2023)

Endorsed by Methodical Commission of Heat Power Faculty (minutes #9 on May 26, 2023)