Structuring Code in Rust:
Crates, Modules, Packages, Editions, and Versioning

Package

- A package is a collection of one or more crates that are bundled together.
- **Cargo.toml**: This is the configuration file for the package. It contains metadata about the package (like its name, version, authors), dependencies, and other settings.

```
my_project/              // name of the Rust project (or package) you create using Cargo
├── Cargo.toml           // Manifest file for the project
├── Cargo.lock           // Automatically generated file to lock dependency versions
├── src/                 // Source directory
│   ├── lib.rs           // Main source file for the library crate
│   └── main.rs          // Main source file for the default binary crate
└── bin/                 // Directory for additional binary crates
    ├── bin1.rs          // Source file for another binary target
    └── bin2.rs          // Source file for yet another binary target
```

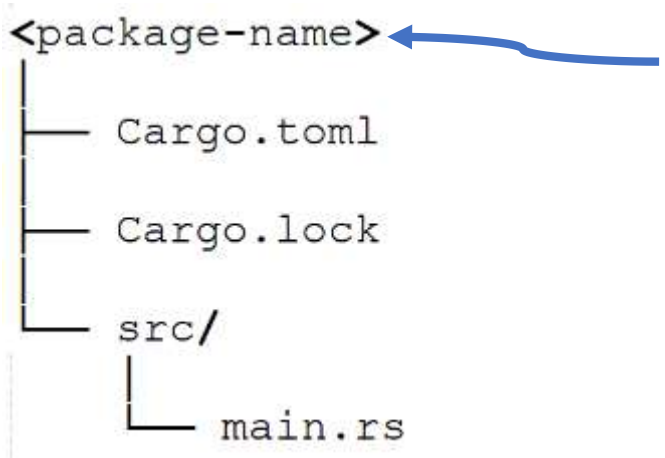
cargo new <package-name>

When you run the command cargo new <package-name>, Cargo creates a new package with the given name. By default, it creates a binary package, meaning it's set up to produce an executable program.

```
my_project/              // name of the Rust project (or package) you create using Cargo
├── Cargo.toml           // Manifest file for the project
├── Cargo.lock           // Automatically generated file to lock dependency versions
├── src/                 // Source directory
│   ├── lib.rs           // Main source file for the library crate
│   └── main.rs          // Main source file for the default binary crate
└── bin/                 // Directory for additional binary crates
    ├── bin1.rs          // Source file for another binary target
    └── bin2.rs          // Source file for yet another binary target
```

cargo new <package-name>

When you run the command cargo new <package-name>, Cargo creates a new package with the given name. By default, it creates a binary package, meaning it's set up to produce an executable program.



<package-name> is the directory that contains the whole package.

Cargo.toml Vs Cargo.lock

- **Cargo.toml** is like a blueprint, where you define what your project needs.
- **Cargo.lock** is a snapshot, locking down the exact versions of everything used in the project to ensure consistent builds across different environments.

Cargo.toml Vs Cargo.lock

- **Cargo.toml** is like a blueprint, where you define what your project needs.
- **Cargo.lock** is a snapshot, locking down the exact versions of everything used in the project to ensure consistent builds across different environments.

Crate

A crate is a package of code that can be compiled into a binary (executable) or library (reusable code).

- Binary crate
- Library crate

Binary crate

- A binary crate compiles into an executable program. It's meant to be run as a standalone application. When you build a binary crate using Rust's build system (Cargo), it produces an executable file

- The main characteristic of a binary crate is that it contains a **main** function, which serves as the entry point of the program.

- Binary crates are what you use when you're developing command-line applications or desktop applications

Library crate

- A library carte compiles into a library, not an executable.
- It's typically used to provide reusable code, such as functions, structs, enums, traits, etc., to other crates.
- A library crate is identified by the presence of a **lib.rs** file, which acts as the library’s root and defines its public API.
- Other projects can depend on these library crates as **dependencies**, reusing the code and functionality they provide.

- **cargo new** my_binary_project
  - This command creates a new directory called *my_binary_project* with the necessary files for a binary crate
  - **src** directory will be created containing a **main.rs** file

- **cargo new** my_library_project **--lib**
  - This command creates a new directory called *my_library_project* with a setup for a library crate
  - **src** directory will be created, but instead of a main.rs file, you'll find a **lib.rs file**. The lib.rs file is where you defnie the library's functionality.

Exercise

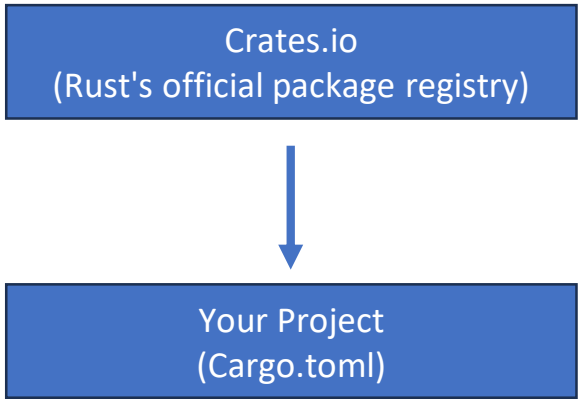
Write a Rust Application to Fetch and Display Top Headlines from The New York Times

(You may download the attached Cargo.toml and main.rs file from the resource section of this lecture.)

External crates

Developers publish their libraries as crates on this platform, making them available for others to use

To use an external crate in a Rust project, you need to declare it in your project's Cargo.toml file under the [dependencies] section



```toml
[package]
name = "my_project"
version = "0.1.0"
authors = ["Your Name <you@example.com>"]
edition = "2021"
description = "An example project with multiple sections"

[dependencies]
serde = "1.0.130"
reqwest = "0.11.6"

[build-dependencies]
cc = "1.0.71"

[dev-dependencies]
criterion = "0.3.5"

[features]
default = ["logging"]
logging = []

[target.'cfg(target_os="linux")'.dependencies]
openssl = "0.10.38"

[profile.release]
optimizations = true
lto = "thin"
```

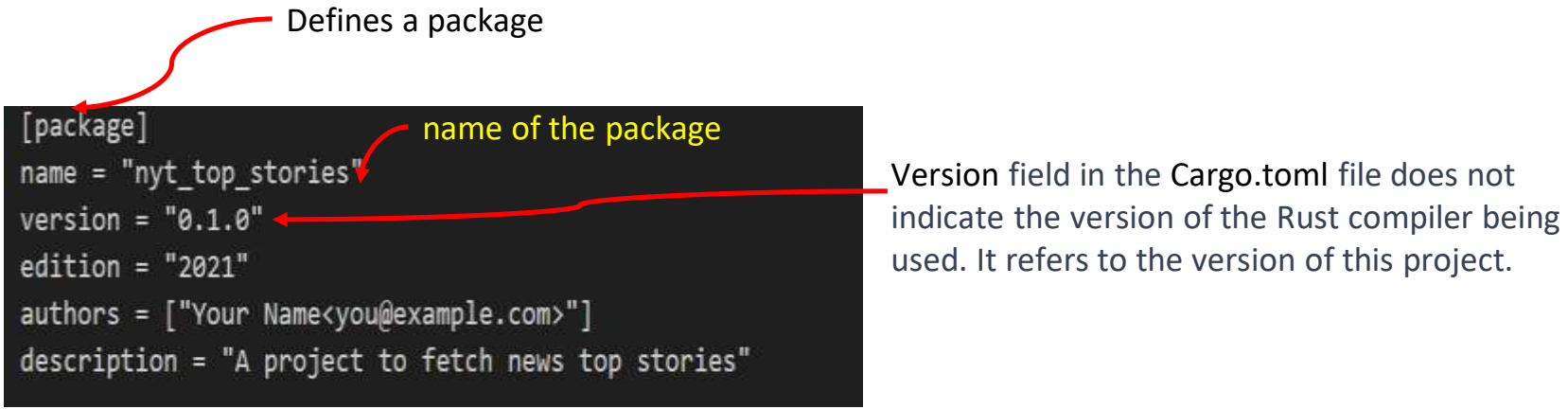
An example of Cargo.toml

- **[]** in the Cargo.toml file are **sections** defined by Cargo's specification
- Each section typically contains a set of key-value pairs that pertain to that specific section. These key-value pairs are used to configure various aspects related to the section

Read more :
https://doc.rust-lang.org/cargo/reference/manifest.html

Section: [package]

[package] section is used to define and configure the package itself. It contains key-value pairs that provide metadata about the package, such as the name, version, authors, and edition.



Semantic Versioning (SemVer) convention (https://semver.org/)

In SemVer, a version number is composed of three parts: **MAJOR.MINOR.PATCH**

1. **MAJOR version**: Incremented when you make incompatible API changes. If you make a significant change that breaks backward compatibility, you would increment the MAJOR version (e.g., 2.0.0).

1. **MINOR version**: Incremented when you add functionality in a backward-compatible manner. If you add new features without breaking backward compatibility, you increment the MINOR version (e.g., 2.1.0).

1. **PATCH version**: Incremented when you make backward-compatible bug fixes. For bug fixes and minor changes, increment the PATCH version (e.g., 2.1.1).

Version = “0.1.0”

A version number like 0.1.0 in software development generally following SemVer signals that the package is in its early, unstable stages of development. Users and developers should expect that changes between versions, even minor or patch versions, could potentially break existing functionality.

Package structuring rules

1. A Rust package must contain at least one crate, either a library or a binary crate.
2. A Rust package can contain multiple binary crates but only zero or one library crate

Package structuring rules

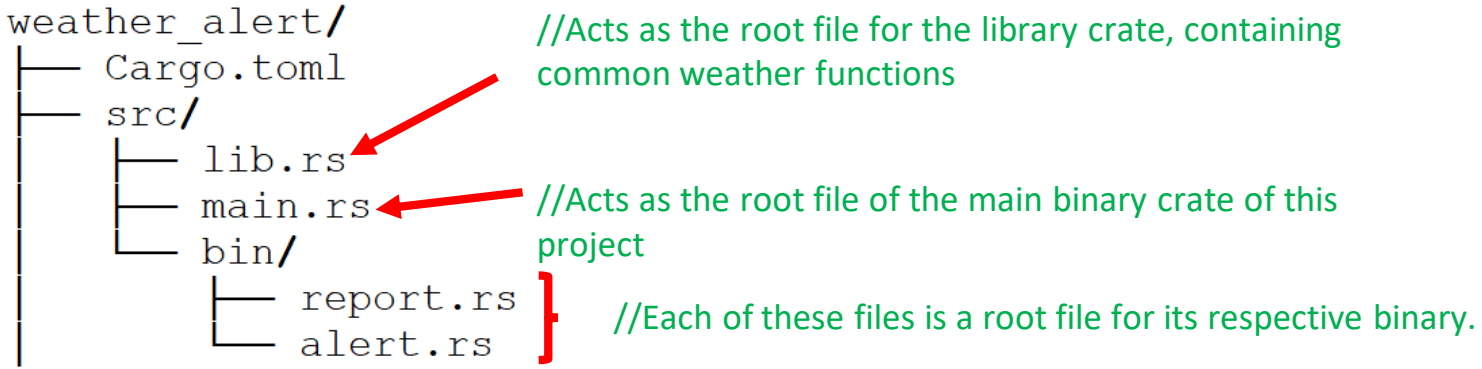
- If you have only **src/lib.rs** in your package, then you have one library crate and no binary crates.
- If you have only **src/main.rs** in your package, then you have one binary crate and no library.
- If you have both **src/lib.rs** and **src/main.rs**, then you have one library crate and one binary crate.
- If you add more **.rs** files under the **src/bin** directory, each file represents an additional binary crate in your package.

Package structure

```
weather_alert/
├── Cargo.toml
├── src/
│   ├── lib.rs          // Library crate with common weather functions
│   ├── main.rs         // Main application for weather forecasting
│   └── bin/
│       ├── report.rs   // Utility to generate weather reports
│       └── alert.rs    // Utility to send weather alerts
```

Crate Root file

In the context of Rust's project structure, the term "root file" refers to the primary source file where the compiler starts processing the crate.



Summary

- **src/lib.rs** is the root of a library crate and should reside in the **src** directory.
- **src/main.rs** is the root of the main binary crate if the package
- **src/bin/*.rs** contains separate additional binary crates. Each **.rs** file in this directory is compiled as a separate binary executable.
- If you place a **lib.rs** file in the **src/bin** directory, the Rust compiler will not recognise it as the root of a library crate. Instead, it will be treated as just another binary

Section : [dependencies]

- If your package has dependencies (other packages), it must be specified in Cargo.toml
- Each of those dependencies might be using a different edition
- Dependencies are essentially other packages, and they have their own Cargo.toml files where they specify their editions. When you depend on another package, Cargo manages the integration seamlessly, allowing for inter-edition compatibility.

Rust editions

What are rust editions?

When Rust needs to introduce changes that are not backward compatible, it does so through a new edition. This approach allows Rust to evolve and add new features while ensuring that existing code remains functional and stable

Editions so far:

- Rust 2015 (Rust version 1.0)
- Rust 2018 (Rust version 1.31.0)
- Rust 2021 (Rust version 1.56.0)

Editions

2015

2018

If your code specifies that it is using the 2015 edition, it will continue to compile and work as before, even with a newer compiler.

In this edition, ***async*** is not a reserved keyword

async becomes a reserved keyword,

Mixing editions

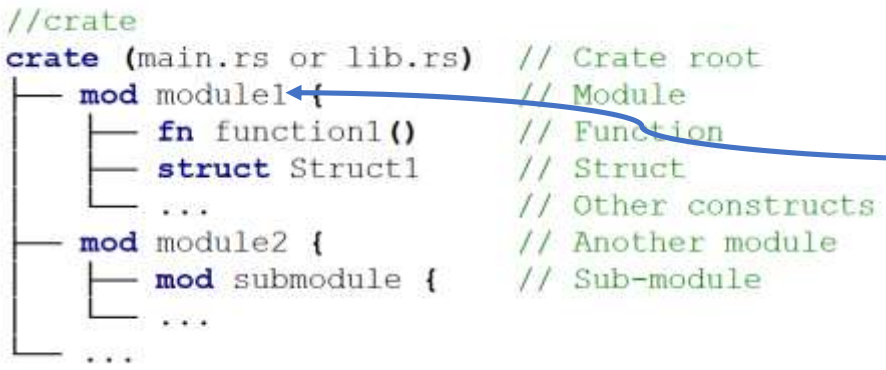
Different crates in the same project can be using different editions

```
my_project (2021 Edition)
├── src/
│   ├── main.rs  (Main binary crate, 2021 Edition, using async/await)
│
└── my_old_lib/  (Library Crate, 2015 Edition)
    ├── Cargo.toml
    └── src/
        └── lib.rs (Contains 'let async = 0')
```

Modules

Modules

- Rust uses a module system which is used for organising, reusing, and controlling the privacy and namespace of code within the project
- A crate can contain a hierarchical module structure, allowing you to organise and structure your code into different namespaces for better organisation, encapsulation, and clarity.



A module in Rust is a container for grouping related functions, structs, traits, constants, and other modules, providing a way to organize code and control its visibility within a program.

Why modules?

Key reasons for using modules in your project

Hierarchical Structure
Rust's module system enables you to create a tree-like hierarchy of modules within a crate. This hierarchy can reflect logical divisions in your code, separating different functionalities, components, or domains.

```
math_library_crate
|
|-- mod arithmetic
|    |-- mod operations
|         |-- fn add
|         |-- fn subtract
|
|-- mod geometry
     |-- mod shapes
          |-- mod rectangle
          |    |-- fn area
          |
          |-- mod circle
               |-- fn area
```

Key reasons for using modules in your project

- **Namespaces**
  - Each module in Rust acts as its own namespace. This means you can have items (like functions, structs, enums, etc.) with the same name in different modules without conflict
- **Encapsulation and Privacy**
  - Modules can be used to encapsulate code, as Rust enforces privacy boundaries at the module level. By default, items (functions, structs, etc.) in a module are private and only accessible within that module unless explicitly declared as public (**pub**).
- **Organization and Clarity**
  - Modules help to organize code for easier navigation, understanding, and maintenance.
- **Reusability**
  - Modules enable code reusability, allowing functionalities to be defined once and used in multiple places, reducing redundancy.

Keywords

When working with modules in Rust, you may encounter the following keywords:

1.mod: This keyword defines a new module. It's used to organize code into logical units and can be used to create both file-based and inline modules.

1.use: This keyword brings a module, type, function, or other items into the current scope, making them accessible without requiring full path specification.

1.pub: This keyword makes items public. By default, module contents (functions, structs, enums, etc.) are private and can only be accessed within their module or child modules. pub allows these items to be accessible from outside their module.

Keywords

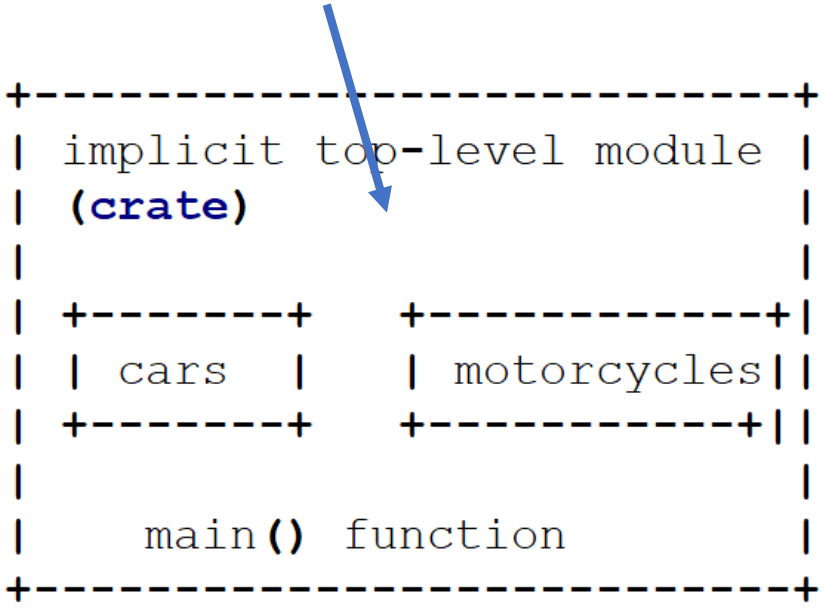
When working with modules in Rust, you may encounter the following keywords:

4. crate: In the context of modules, **crate** refers to the current crate (the top-level package or library). It's used to refer to items in the root of the crate.

5. **super**: This keyword refers to the parent module of the current module. It's useful for accessing items in a module one level above the current module's hierarchy.

6. self: This keyword refers to the current module itself. It's often used in **use** statements to refer to items in the same module

Top-level anonymous module and you can refeR to its items using the **crate::** prefix



Privacy rules

1. **Default Privacy**:
   - All items (functions, methods, structs, enums, modules, and constants) are private by default.
2. **Parent to Child Module Access**:
   - Items in a parent module cannot access private items inside child modules.
3. **Child to Parent (Ancestor) Module Access**:
   - Items in child modules can use the items in their ancestor modules (not just the immediate parent, but any module up the hierarchy).
4. **Sibling Module Access**:
   - Sibling modules cannot access each other's private items by default.

Organizing modules into separate files

```
src/
|-- main.rs
|-- circle.rs
|-- square.rs
```

```
src/
|-- main.rs
|-- circle/
    |-- mod.rs
    |-- [other submodule files, if any]
|-- square/
    |-- mod.rs
    |-- [other submodule files, if any]
```

Organising each module in separate files

Organising each module in a separate directory with **mod.rs** file

```rust
mod properties;

pub struct Circle {
    pub radius: f64,
}

impl Circle {
    pub fn area(&self) -> f64 {
        std::f64::consts::PI * self.radius * self.radius
    }
}
```

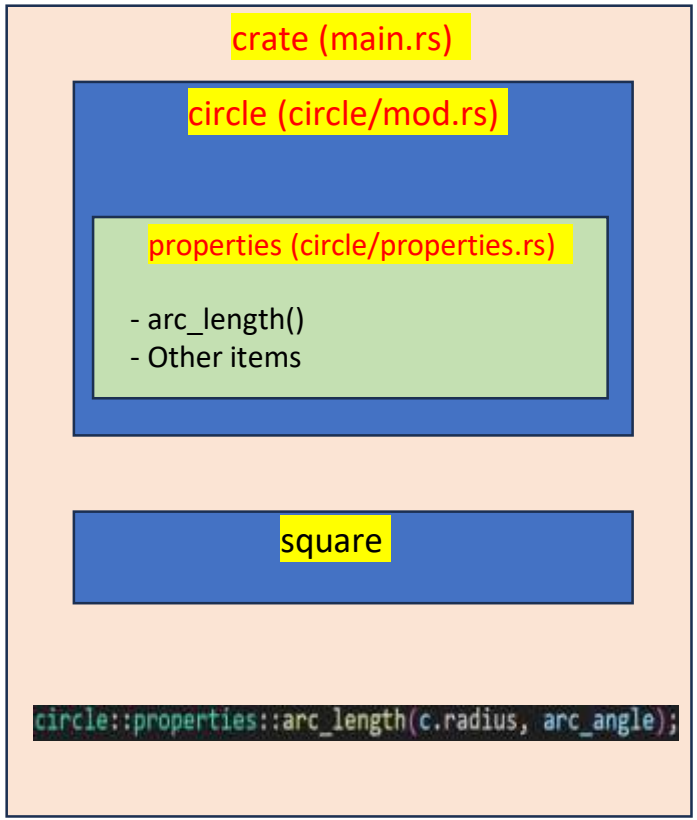
circle/mod.rs

```rust
impl Circle {
    pub fn circumference(&self) -> f64 {
        2.0 * 3.14159 * self.radius
    }
}

pub fn arc_length(radius: f64, angle_in_degrees: f64) -> f64 {
    let angle_in_radians = angle_in_degrees.to_radians();
    radius * angle_in_radians
}
```

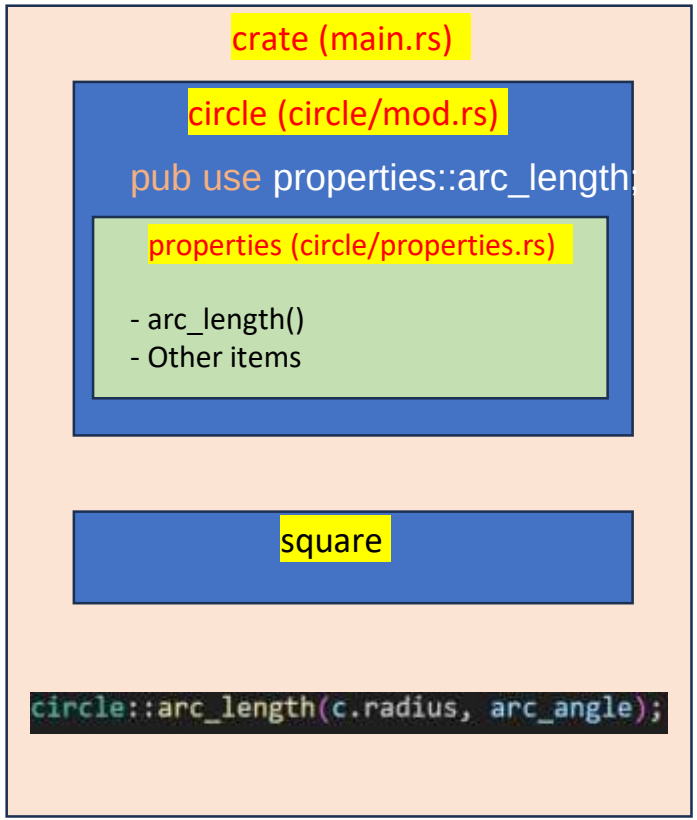
properties.rs (Submodule of circle)

Here **Circle** struct is public (declared with pub in the parent module), and any public methods (pub fn) within its implementation are also accessible wherever the **Circle** struct is accessible, regardless of the submodule's visibility.

Re-exporting



Re-exporting



Re-exporting

Re-exporting allows you to make an item from one module available in another module without requiring users to know about the internal structure of your code.

*For example, by re-exporting **arc_length** in the **circle** module **(pub use properties::arc_length;),** users can access it directly with **circle::arc_length**, without needing to know about the **properties** submodule.*

Smart Pointers and Raw Pointers

Raw pointers in Rust

```c
#include <stdio.h>
int main()
{
    int x = 5;
    int* raw_ptr = &x;
    printf("raw_ptr points to: %d\n", *raw_ptr);
    *raw_ptr = 10;
    printf("raw_ptr points to: %d\n", *raw_ptr);
    return 0;
}
```

A mutable pointer in ‘C’

Takes a mutable reference to x (&mut x) and casts it to a mutable raw pointer

```rust
fn main() {
    let mut x = 5;
    let raw_ptr = &mut x as *mut i32;
    unsafe {
        println!("raw_ptr points to: {}", *raw_ptr);
        *raw_ptr = 10;
        println!("raw_ptr points to: {}", *raw_ptr);
    }
}
```

A mutable raw pointer in ‘Rust’

Why dereferencing a raw pointer is unsafe?

- Rust cannot guarantee that the lifetime of the data being pointed to (the referent) is valid. The data might already have been deallocated or moved
- The raw pointer could point to an invalid memory location, leading to undefined behavior when dereferenced
- Unlike references (&T or &mut T), raw pointers (*const T or *mut T) are not tracked by the borrow checker. Rust's ownership and borrowing rules do not apply, making it impossible to ensure exclusive or shared access rules are being upheld
- A raw pointer might point to data that does not match the expected type
- Dereferencing a raw pointer without synchronization might lead to data races in multithreaded contexts

Types of raw pointers

1. **Mutable Raw Pointer (*mut T)**:
    - This raw pointer allows for mutation of the data it points to. It's the direct counterpart to a mutable reference (&mut T) but without borrowing and safety guarantees.

1. **Immutable Raw Pointer (*const T)**:
    - This is a raw pointer used to point to data where the intention is that the data should not be modified through this pointer. It is similar in concept to a "const pointer" in languages like C and C++.

*Note: The data pointed to by a ***const T** is not inherently constant or immutable in itself; rather, ***const T** is a way to express the intent that the data should not be modified through this pointer.*

What raw pointers lack compared to smart pointers in Rust

1. **No safety guarantees**: Dereferencing raw pointers is inherently unsafe.
2. **No automatic memory management**: Raw pointers don't manage memory allocation or deallocation.
3. **No borrowing and ownership eneforcement**: Raw pointers bypass Rust's ownership and borrowing rules.
4. **Ambiguous mutability**: Mutability and immutability distinctions are only by convention..
5. **No lifetimes**: Raw pointers don't enforce lifetimes, risking dangling pointers.
6. **Prone to data races**: Multiple mutable raw pointers can point to the same location without checks.
7. **No reference counting**: Unlike Rc<T> or Arc<T>, raw pointers don't keep track of references.
8. **No runtime borrow checking**: Unlike RefCell<T>, raw pointers don't check borrowing rules at runtime.
9. **Direct risk of undefined behavior**: Incorrect use can lead to undefined behavior without compiler warnings.
11. **No thread-safety guarantees**: Raw pointers by themselves don't offer any guarantees for thread safety

Raw pointer use cases

- **Interoperability with C and Other Low-Level Systems Code:** To interface with existing C libraries or operating systems' APIs in systems programming

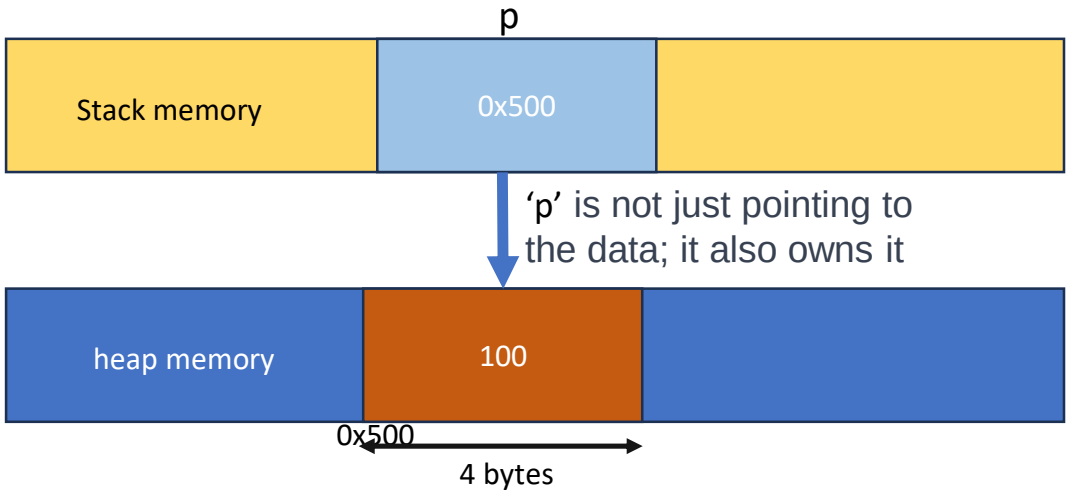
Smart pointers in Rust

1. Box<T>
2. Rc<T>
3. Arc<T>
4. Cell<T>
5. RefCell<T>
6. Mutex<T>
7. RwLock<T>
8. Cow<B>

Box<T>

- Box<T> type is used when there is a need to allocate a value of type T on the heap memory.
- **let mut** p: Box**<i32> =** Box**::**new**(**100**);** here**,** The i32 value 100 takes up 4 bytes on the heap, but p (the Box<i32>) on the stack only takes up the space needed for a single pointer (On a 64-bit system: The pointer size is 8 bytes, so **p** takes up 8 bytes on the stack)
- As the owner of the data, Box<T> is responsible for cleaning up the data when it goes out of scope
- The ownership by a Box<T> is unique. There can only be one owner of the data at a time, which prevents issues like double-free errors and data races that are common in manual memory management.

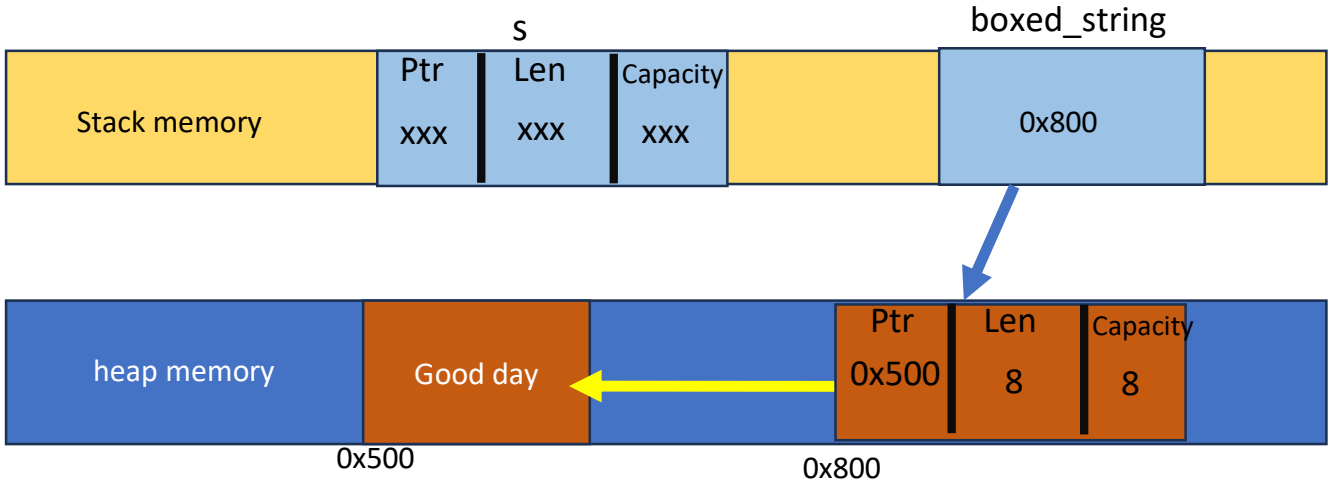
Memory layout



Box<String>

Memory layout

```
let boxed_string: Box<String> = Box::new(s);
```



Box<T> to handle Dynamically Sized Types(DSTs)

Dynamically Sized Types

Dynamically Sized Types (DSTs) in Rust are types for which the size cannot be known at compile time and is only determined at runtime.

Category	Examples	Description
Regular Types	i32, bool, char, struct, enum, [i32; 5]	These types have a statically known size, determined at compile time.
Dynamically Sized Types (DSTs)	[T], dyn Trait	These types do not have a size known at compile time and can only be fully understood at runtime

Dynamically Sized Types : Slices

Dynamically Sized Types (DSTs), like slices, have two primary methods for being utilized in a program

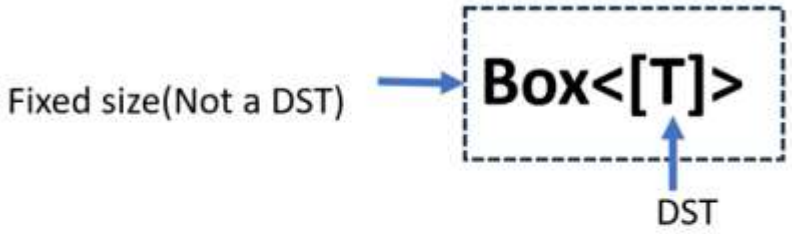
1. Through references like &[T] or &mut [T]
2. Through smart pointers such as Box<[T]>

Boxing an DST : Box<DST>

*Some smart pointers, like **Box, Rc, and Arc**, can be used to manage DSTs in rust by providing them with a statically known size at compile time*

Why use Box<[T]> instead of &[T]?

If you use &[T], you must ensure that the data the slice references outlives the borrow. This constraint doesn't exist for Box<[T]> since it owns the data.



Dynamically Sized Types : Trait objects

1. Trait objects are commonly used through references, such as **&dyn Trait** or **&mut dyn Trait**
2. Trait objects can also be used through smart pointers, such as **Box<dyn Trait>**

Storing trait objects in Vector using Box<dyn Trait>

Consider a situation where you need a '**Vec**' to hold objects of different types that implement the same trait.

The **Box<dyn Shape>** is a fixed-size pointer, which points to the heap-allocated memory for the DST

```
let shapes: Vec<Box<dyn Shape>> = vec![
    Box::new(Circle { radius: 3.0}),
    Box::new(Square { side: 2.0 }),
];
```

When you create an instance of Box<dyn Shape> like **Box::new(Circle { radius: 1.0 }),** Rust dynamically allocates enough heap memory to store the specific type (**Circle** in this case)

Converting a Box<T> into a raw pointer and vice versa.

Use case: **Foreign Function Interface (FFI) with C**

When calling C functions from Rust, you typically use raw pointers (*mut T or *const T) to pass data.

and, when receiving data from C functions, you may receive it as raw pointers, and you might want to convert these raw pointers back into Rust's types to manage them safely within Rust's ownership model.

Box::into_raw() and **Box::from_raw()**

Box::into_raw() is a method that consumes a **Box<T>** and returns a raw, mutable pointer to the heap-allocated T value.

```
let boxed_int = Box::new(123);
let raw_ptr_to_int: *mut i32 = Box::into_raw(boxed_int);
```

Rust's safe memory management is not applicable here.
memory must be manually managed.

Deallocating raw pointer in rust

- The safest and most idiomatic way to deallocate memory in rust, when starting with a raw pointer obtained from **Box::into_raw()**, is to convert it back to a **Box** with **Box::from_ra()** and let it go out of scope for automatic deallocation.

- Attempting to deallocate memory manually, without using Box, introduces unnecessary risk of errors, including memory leaks, double frees, or undefined behavior.

Vec<T> Vs Box<[T]>

- USe Vec<T> when you need a dynamically sized collection that can grow or shrink.
- Vec<T> is ideal for situations where the number of elements is not known at compile time or can change during execution.

- Use Box<[T]> for fixed-size collections when the size is known and won't change after allocation.
- A boxed slice ([T]) is useful for representing a heap-allocated array with a fixed number of elements.

Borrowing Box<T>

The **into_raw()** function consumes the **Box<T>**, transferring ownership of the heap-allocated data to a raw pointer, effectively "forgetting" the Box.

The **from_raw()** function is used to reconstitute the **Box<T>** from the raw pointer, taking back ownership and allowing Rust to manage the memory again.

How about borrowing Box<T> if we want to temporarily operate with the raw pointer?

Enabling Recursive Types using Box<T>

Recursive types are those where an instance of the type can contain another instance of the same type.

```
enum FileSystemNode {
    File(String),
    Directory(String, Vec<FileSystemNode>),
}
```

Recursive type with infinite size

```
struct Node {
    data: i32,
    next: Node,
}
```

A structure containing an instance of itself as a member field

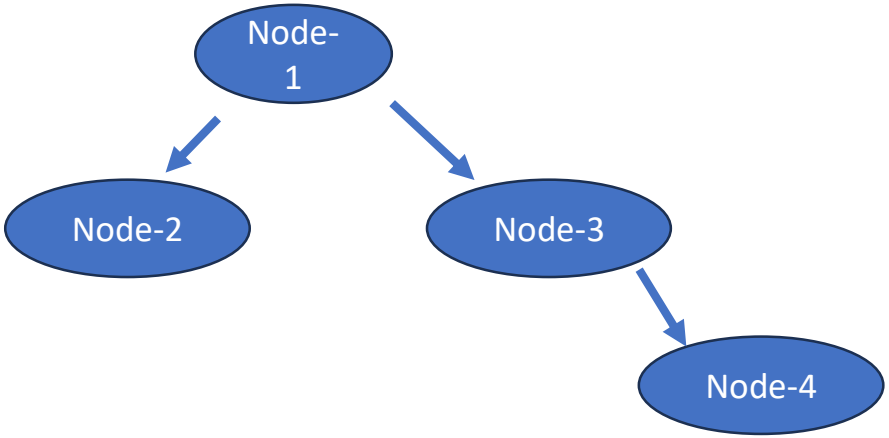
Recursive type with Indirection

```
struct Node {
    data: i32,
    next: Node,
}
```

```
struct Node {
    data: i32,
    next: Option<Box<Node>>,
}
```

Recursive type with infinite size

Recursive type with indirection with Box<T> leads to finite size

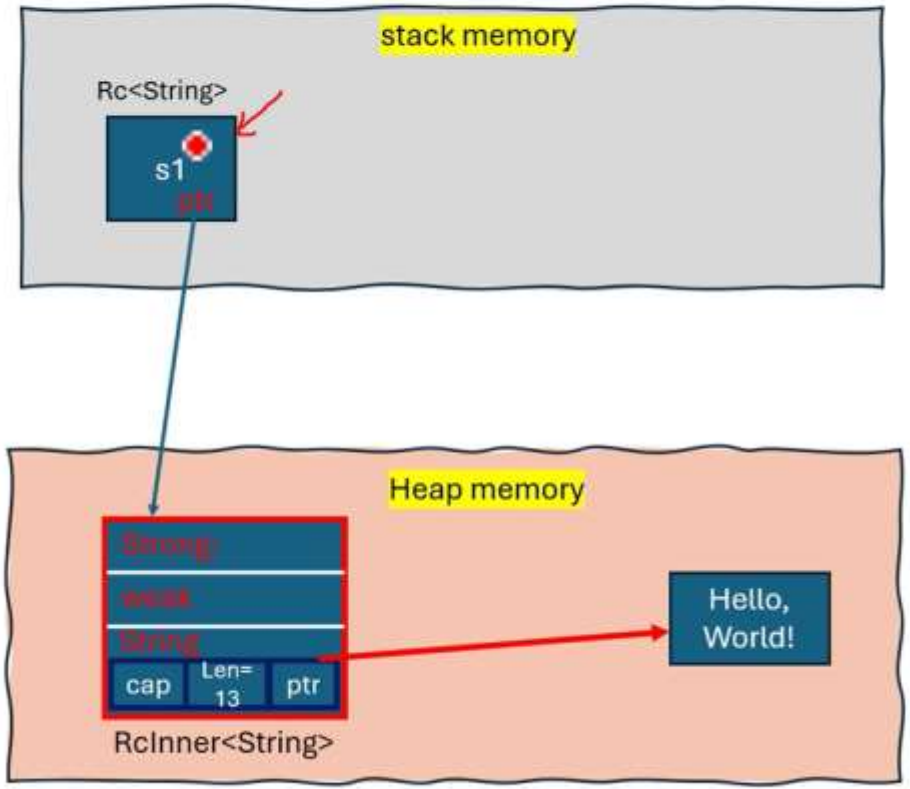

Node-
1
Node-2
Node-3
Node-4

Rc<T> : Single-threaded shared ownership

Rc<T>

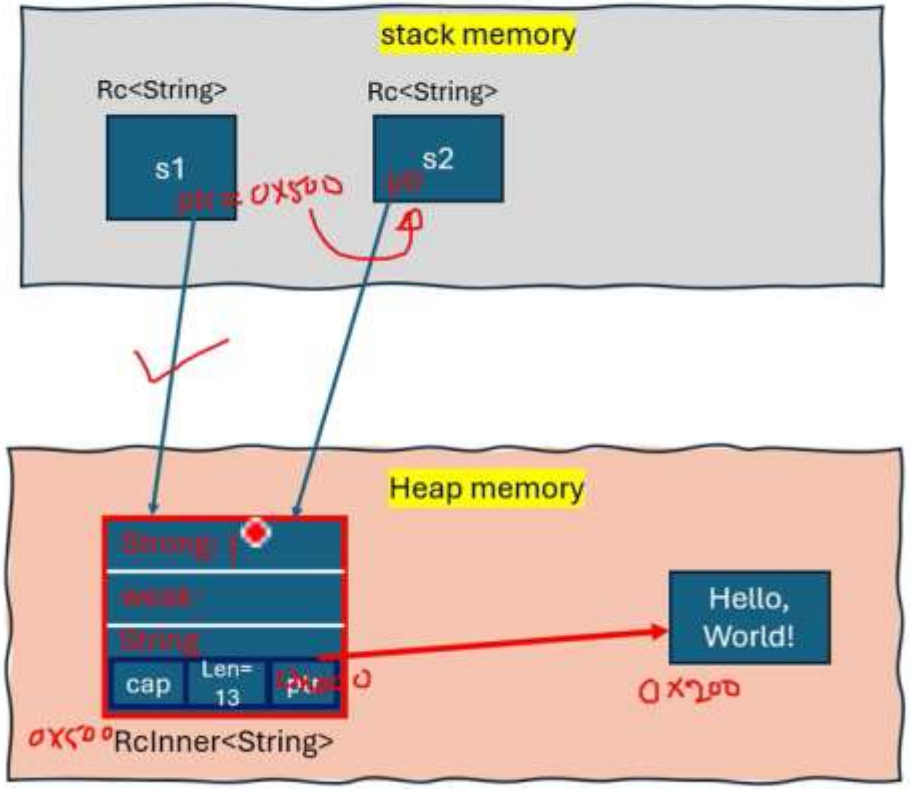
Rc<T> is useful in scenarios where multiple parts of a single-threaded application need to share ownership of the same data, and where the data's lifetime should not be tied to any single owner

Memory representation



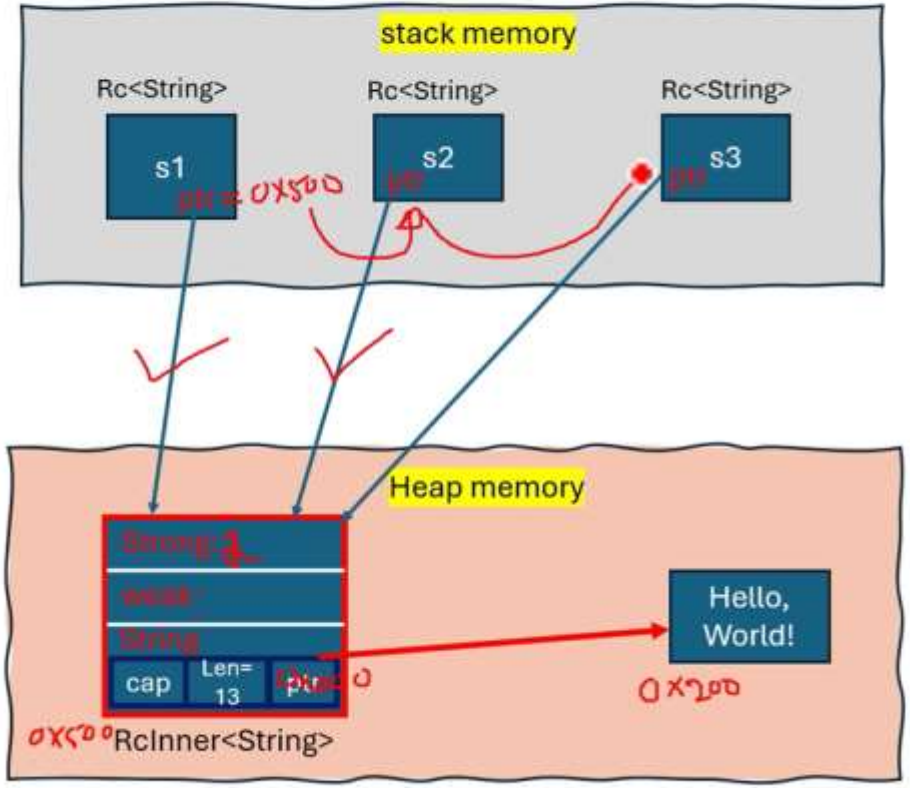

```
fn main() {

let s1 =
Rc::new(String::from("Hello, world!"));

}
```

Memory representation




```
fn main() {

let s1 =
Rc::new(String::from("Hello, world!"));

let s2 = Rc::clone(&s1);



}
```

Memory representation




```
fn main() {

let s1 =
Rc::new(String::from("Hello, world!"));

let s2 = Rc::clone(&s1);

let s3 = Rc::clone(&s1);

}
```

Shared ownership using Rc<T>

1) Rc<T> provides shared ownership of data of type `T`, allocated in the heap
2) Rc<T> is for immutable data only.
3) To mutate data that multiple owners share in a single-threaded scenario, you need to pair Rc<T> with a RefCell<T>. This combination gives you multiple owners while also allowing interior mutability
4) Rc<T> is lightweight because it assumes all its owners are in the same thread. It doesn't use atomic operations, which would add overhead
5) counterpart of Rc<T> for multi-threaded applications is Arc<T>
6) Use Arc<T> when you need to share data between multiple threads safely

Rc<T>: Strong and weak count

Strong count

- Strong and weak counts are used for internal bookkeeping to manage the lifecycle of shared data.
- As long as the strong count is greater than zero, the underlying data is guaranteed to remain alive.
- When you clone an Rc<T>, this creates another strong reference, incrementing the strong count by one.
- When the strong count reaches zero, the underlying data (T) is dropped (its Drop implementation is run), but the allocation containing the reference counts may still remain if there are any weak references.

```rust
use std::rc::Rc;

fn main() {

    let s1 = Rc::new(42);
    println!("{}", Rc::strong_count(&s1)); //prints 1

    {
        let s2 = Rc::clone(&s1);
        println!("{}", Rc::strong_count(&s1)); //prints 2
    }

    println!("{}", Rc::strong_count(&s1)); //prints 1

}
```

Weak count

- The number of Weak<T> references to the same allocation that don't keep the data alive.
- Weak references don’t influence the data’s lifetime and can’t be directly dereferenced
- They must be upgraded (via Weak::upgrade) to an Rc<T> to gain access to the data, and this upgrade only succeeds if the data is still alive (i.e., strong count > 0).
- Use **Rc::downgrade()** to create weak references (Rc::downgrate() creates an instance of Weak<T>)
- When strong count decrements or data is dropped it won’t affect weak count

Summary: Mutating ‘T’ in Rc<T>

- You cannot directly mutate data inside an Rc<T> because it ensures shared ownership and read-only access.
- To enable mutation, you can use:
  - **RefCell<T>**: Provides interior mutability for single-threaded contexts.
  - **Rc::get_mut()**: Allows direct mutation but works only if no other references exist.