**Smart Pointers:**

A **pointer** holds a memory address pointing to some data. The most common pointer in Rust is a **reference** (&), which borrows data without ownership or overhead.

**Smart pointers** go beyond regular references:

* They behave like pointers **but include metadata and extra capabilities**.
* In Rust, there is an additional difference between references and smart pointers: while references only borrow data, in many cases smart pointers *own* the data they point to.
* Examples include String and Vec<T> count as smart pointers because they own some memory and allow you to manipulate it. They also have metadata and extra capabilities or guarantees.

Extra information about metadata and capabilities:

**Metadata = Extra Information About the Data**

**Metadata** is additional information stored alongside or inside the smart pointer to help manage the data it points to. For example, metadata can be length, capacity, element type, etc..

**Capabilities = Extra Behaviors or Features**

**Capabilities** refer to the **additional powers** or behaviors smart pointers provide on top of simple references. These behaviors can be Heap allocation, shared ownership, runtime checks, etc..

Smart pointers are **implemented using structs** and typically implement two key traits:

* **Deref**: lets the smart pointer act like a reference.
* **Drop**: lets you define custom behavior when the pointer goes out of scope.

The most common smart pointers in the standard library:

* **Box<T>,** for allocating values on the heap
* **Rc<T>,** a reference counting type that enables multiple ownership
* **Ref<T>** and **RefMut<T>,** accessed through **RefCell<T>,** a type that enforces the borrowing rules at runtime instead of compile time

**Using Box<T> to Point to Data on the Heap:**

The most straightforward smart pointer is a ***box***, whose type is written **Box<T>.** Boxes allow you to store data on the heap rather than the stack. What remains on the stack is the pointer to the heap data.

Boxes don’t have performance overhead, other than storing their data on the heap instead of on the stack. But they don’t have many extra capabilities either.

Used when:

* Type size is unknown at compile time and you want to use a value of that type in a context that requires an exact size.
* You need to transfer ownership of large data without copying it.
* You want to work with trait objects (covered in Chapter 18).

**Basic Syntax Example:**

A group of symbols on a white background

AI-generated content may be incorrect.

We define the variable b to have the value of a *Box* that points to the value 5, which is allocated on the heap. It behaves like a regular value, deallocated automatically when it goes out of scope,  as b does at the end of main.

**Recursive Types and Why Box Is Needed:**

**The Problem with Recursive Types**

Rust must know a type’s size at compile time. Recursive types (e.g., a list containing itself) can be **infinitely large** so Rust can’t know how much space the value needs and throws a compile-time error. Because boxes have a known size, we can enable recursive types by inserting a box in the recursive type definition.

Example cons list:



Each item in a cons list contains two elements: the value of the current item and the next item. The last item in the list contains only a value called Nil without a next item. For example, enum definition for a cons list:

A computer screen shot of a list

AI-generated content may be incorrect.

However, when we try to compile the code, it will show the error:

A screenshot of a computer program

AI-generated content may be incorrect.

The error shows this type “has infinite size.” The reason is that we’ve defined List with a variant that is recursive: it holds another value of itself directly. As a result, Rust can’t figure out how much space it needs to store a List value. First we’ll look at how Rust decides how much space it needs to store a value of a non-recursive type.

**Computing the Size of a Non-Recursive Type**

Here we will see how Rust complier computes size of Non-Recursive Type by the message enum example:

A computer code with text

AI-generated content may be incorrect.

To calculate the space to allocate for Message value, Rust goes through each of the variants,  to see which variant needs the most space. Rust sees that Message::Quit doesn’t need any space, Message::Move needs enough space to store two i32 values, and so forth. Because only one variant will be used, the most space a Message value will need is the space it would take to store the largest of its variants (the largest variant in on a 64-bit system Message enum is string(3\*usize): 24 bytes).

However, when it calculates the size of enum List, the compiler starts by looking at the Cons variant, which holds a value of type i32 and a value of type List. Therefore, Cons needs an amount of space equal to the size of an i32 plus the size of a List. To figure out how much memory the List type needs, the compiler keeps looking at the variants, starting with the Cons variant. The Cons variant holds a value of type i32 and a value of type List, and this process continues infinitely:

A diagram of cons and cons

AI-generated content may be incorrect.

**Solution: Use Box<T> for Indirection**

Using a Box<T> breaks the infinite recursion:

A white background with black text

AI-generated content may be incorrect.

Because a Box<T> is a pointer, Rust always knows how much space a Box<T> needs: a pointer’s size doesn’t change based on the amount of data it’s pointing to. The Box<T> will point to the next List value that will be on the heap rather than inside the Cons variant. Conceptually, we still have a list, created with lists holding other lists, but this implementation is now more like placing the items next to one another rather than inside one another like the picture above.

The Cons variant needs the size of an i32 plus the space to store the box’s pointer data. The Nil variant stores no values, so it needs less space than the Cons variant, now the Cons variant looks like:

A black and white diagram with black text

AI-generated content may be incorrect.

Boxes provide only the indirection and heap allocation; they don’t have any other special capabilities, They also don’t have the performance overhead that these special capabilities incur, so they can be useful in cases like the cons list where the indirection is the only feature we need.

The Box<T> type is a smart pointer because it implements the **Deref** trait, which allows Box<T> values to be treated like references. When a Box<T> value goes out of scope, the heap data that the box is pointing to is cleaned up as well because of the **Drop** trait implementation.

**Treating Smart Pointers Like Regular References with Deref**

Implementing the Deref trait allows you to customize the behavior of the *dereference operator* \*.  By implementing Deref in such a way that a smart pointer can be treated like a regular reference, you can write code that operates on references and use that code with smart pointers too.

**Following the Reference to the Value**

Example of using reference and dereference:

A close-up of a computer code

AI-generated content may be incorrect.

The variable x holds an i32 value 5. We set y equal to a reference to x. We can assert that x is equal to 5. However, if we want to make an assertion about the value in y, we have to use \*y  (*dereference*) so the compiler can compare the actual value.

If we tried to write assert\_eq!(5, y); instead, we would get this compilation error:

A screen shot of a computer code

AI-generated content may be incorrect.

Comparing a number and a reference to a number isn’t allowed because they’re different types. We must use the dereference operator to follow the reference to the value it’s pointing to.

**Using Box<T> Like a Reference**

We can use a Box<T> instead of a reference; the dereference operator of Box<T> can be used as the same way as the dereference operator use on reference:

A computer screen shot of a number of text

AI-generated content may be incorrect.

The main difference between reference and Box<T> is that here we set y to be an instance of a box pointing to a copied value of x (A box Box::new(x) **copies** the value of x from the original x and owns it on the **heap**.) rather than a reference pointing to the value of x.

**Defining Our Own Smart Pointer**

We can build our own smart pointer as same as Box<T>:

A screenshot of a computer code

AI-generated content may be incorrect.

The Box<T> type is ultimately defined as a tuple struct with one element, so we define a MyBox<T> type in the same way. The MyBox::new function is a constructor that takes one parameter of type T and returns a MyBox instance that holds the value passed in.

Let’s try adding the main function and changing it to use the MyBox<T> type we’ve defined instead of Box<T>:

A computer code with numbers

AI-generated content may be incorrect.

The code won’t compile because Rust doesn’t know how to dereference MyBox. The Deref trait provided by the standard library, requires us to implement one method named deref that borrows self and returns a refer to the inner data:

A screenshot of a computer code

AI-generated content may be incorrect.

The *type Target = T*; syntax defines an associated type for the Deref trait to use. Associated types are a slightly different way of declaring a generic parameter (detail in chapter 20, The Rust Programming Language Book).

In the body of the deref method with &self.0 so deref returns a reference to the value we want to access with the \* operator, .0 accesses the first value in a tuple struct.

Without the Deref trait, the compiler can only dereference & references. The **deref method gives the compiler the ability to take a value of any type that implements Deref** and call the deref method to get an & reference that it knows how to dereference.

When we entered \*y behind the scenes Rust actually ran this code:



Rust substitutes the \* operator with a call to the deref method and then a plain dereference so we don’t have to think about whether or not we need to call the deref method.

Deref method returns a reference to a value, if it return the value we know that the value will move out of self because of the ownership system.

Note that the \* operator is replaced with a call to the deref method and then a call to the \* operator just once, each time we use a \* in our code. Because the substitution of the \* operator does not recurse infinitely.

**Implicit Deref Coercions with Functions and Methods**

**Deref coercion** is a feature in Rust that automatically converts a reference to a type implementing the Deref trait into a reference to another type. It allows smart pointers to behave like regular references in function and method calls.

Deref coercion was added to Rust so that programmers writing function and method calls don’t need to add as many explicit references and dereferences with & and \*. The deref coercion feature also **lets us write more code that can work for either references or smart pointers.**

Example:

A computer screen shot of a code

AI-generated content may be incorrect.

The hello function with the argument &m, which is a reference to a MyBox<String> value. Because we implemented the Deref trait on MyBox<T> , Rust performs &MyBox<String> → &String via MyBox's Deref and then &String → &str via String's Deref.

If Rust didn’t implement deref coercion, we would have to write the code like:

A close-up of a computer code

AI-generated content may be incorrect.

The (\*m) dereferences the MyBox<String> into a String. Then the & and [..] take a string slice of the String that is equal to the whole string to match the signature of hello. That is hard to read, write and understand.

When the Deref trait is defined for the types involved, Rust will analyze the types and **use Deref::deref** **as many times** as necessary **to get a reference to match the parameter’s type.** The number of times that Deref::deref needs to be inserted is resolved at compile time, so there is no runtime penalty for taking advantage of deref coercion!

**How Deref Coercion Interacts with Mutability**

Similar to how you use the Deref trait to override the \* operator on immutable references, you can use the DerefMut trait to override the \* operator on mutable references.

Rust does deref coercion when it finds types and trait implementations in three cases:

1. From &T to &U when T: Deref<Target=U>
2. From &mut T to &mut U when T: DerefMut<Target=U>
3. From &mut T to &U when T: Deref<Target=U>

The first case states that if you have a &T, and T implements Deref to some type U, you can get a &U transparently.

The second case states that the same deref coercion happens for mutable references.

Rust will also coerce a mutable reference to an immutable one. But the reverse is *not* possible: immutable references will never change to mutable references.

**Running Code on Cleanup with the Drop Trait:**

The Drop trait in Rust lets you specify custom behavior **when a value goes out of scope**, such as releasing resources like heap memory, file handles, sockets, or locks. Drop trait is almost always used when implementing a smart pointer.

Unlike some languages, Rust provides the mechanism (Drop triat) that programmer can specify the code be run when pointer goes out of scope and the complier will insert this code automatically, , for example when using C/C++ we have to call destructor manually to free up memory, but we don’t have to do that in Rust.

You can implement Drop triat for your custom pointer defining a drop(&mut self) method. For example, we create a CustomPointer:

A screen shot of a computer program

AI-generated content may be incorrect.

We can see that the Drop trait is included in the prelude, so we don’t need to bring it into scope. The body of the drop method is where you would place any logic that you wanted to run when an instance of your type goes out of scope.

Rust automatically called drop for us when our instances went out of scope, calling the code we specified. We can disable this mechanism but that not easy and unnecessary. However, if you want to clean your value early, you might want to force drop method:

A screenshot of a computer code

AI-generated content may be incorrect.

We will see the compile error:

A screenshot of a computer

AI-generated content may be incorrect.

Rust doesn’t let us call drop explicitly because Rust would still automatically call drop on the value at the end of main. This would cause a *double free* error because Rust would be trying to clean up the same value twice.

We can’t disable the automatic insertion of drop when a value goes out of scope, and we can’t call the drop method explicitly. So, if we need to force a value to be cleaned up early, we use the std::mem::drop function.

The std::mem::drop function is different from the drop method in the Drop trait. We call it by passing as an argument the value we want to force-drop. The function is in the prelude, so we can modify main  to call the drop function: A screenshot of a computer code

AI-generated content may be incorrect.

Running this code will print the following: A black text on a white background

AI-generated content may be incorrect.

You can use code specified in a Drop trait implementation in many ways to make cleanup convenient and safe and don’t have to worry about problems resulting from accidentally cleaning up values still in use: the ownership system that makes sure references are always valid also ensures that drop gets called only once when the value is no longer being used.

**Rc<T>, the Reference Counted Smart Pointer**

In Rust, sometime a single value might have multiple owners. For example, in a graph data structures, s, multiple edges might point to the same node, and that node is conceptually owned by all of the edges that point to it. A node shouldn’t be cleaned up unless it doesn’t have any edges pointing to it and so has no owners. You have to enable multiple ownership explicitly by using the Rust type Rc<T>.

Rc<T> stands for Reference Counted smart pointer. It enables multiple ownership of immutable data in single-threaded programs. Rc<T> pointer follow these concept: it keeps a **count** of how many Rc pointers exist for a value, the value is **deallocated** when the last reference goes out of scope.

Rc<T> great for **tree**, **graph**, or **shared data structure** scenarios, however, Rc<T> is only used for single threaded scenarios, if we want to use for multiple threaded, Arc<T> is the choice (we will talk about it later).

**Using Rc<T> to Share Data**

Given an example:

A diagram of a number and arrows

AI-generated content may be incorrect.

We’ll create list “a” that contains 5 and then 10. Then we’ll make two more lists: “b” that starts with 3 and “c” that starts with 4. Both b and c lists will then continue to the first “a” list containing 5 and 10. In other words, both lists will share the first list containing 5 and 10.

Trying to implement this scenario using our definition of List with Box<T> won’t work:A computer code with red and black text

AI-generated content may be incorrect.

When we compile this code, we get this error:

A computer code with black text

AI-generated content may be incorrect.

We could change the definition of Cons to hold references instead, but then we would have to specify lifetime parameters. By specifying lifetime parameters, we would be specifying that every element in the list will live at least as long as the entire list. This creates a tone of work to do, instead of that, we will replace Box<T> by using Rc<T> pointing to a List.

Now, let fix this:

A computer screen shot of a code

AI-generated content may be incorrect.

Each Cons variant will now hold a value and an Rc<T> pointing to a List. When we create b, instead of taking ownership of a, we’ll clone the Rc<List> that a is holding, thereby increasing the number of references from one to two and letting a and b share ownership of the data in that Rc<List>.

We’ll also clone a when creating c, increasing the number of references from two to three. Every time we call Rc::clone, the reference count to the data within the Rc<List> will increase, and the data won’t be cleaned up unless there are zero references to it.

We could have called a.clone() rather than Rc::clone(&a), but Rust’s convention is to use Rc::clone in this case. The implementation of Rc::clone doesn’t make a deep copy of all the data like most types of implementations of clone do. The call to Rc::clone only increments the reference count, which doesn’t take much time.

When looking for performance problems in the code, we only need to consider the deep-copy clones and can disregard calls to Rc::clone.

**Cloning a Rc<T> Increases the Reference Count**

Here is an example to see the reference count changing when as we create and drop references the Rc<List> in “a”:

A screenshot of a computer program

AI-generated content may be incorrect.

This code prints the following:

A black text on a white background

AI-generated content may be incorrect.

We print the reference count, which we get by calling the Rc::strong\_count function. This function is named strong\_count rather than count because the Rc<T> type also has a weak\_count.

We can see that the Rc<List> in a has an initial reference count of 1; then each time we call clone, the count goes up by 1. When c goes out of scope, the count goes down by 1. We don’t have to call a function to decrease the reference count like we have to call Rc::clone to increase the reference count: the implementation of the Drop trait decreases the reference count automatically when an Rc<T> value goes out of scope.

In conclusion, Rc<T> is **safe for multiple readers**, but **not for writing**. To **mutate** shared data, wrap the value in RefCell<T> conjunction with a Rc<T>. We’ll discuss it later.

**Using Arc<T> to Share Data Across Threads:**

* Arc<T> is like Rc<T>, but it’s designed for thread-safe reference counting because it use atomic .
* Use Arc<T> when multiple threads need to read the same data safely.
* Just like Rc<T>, Arc<T> gives you immutable access only.

**Given an Example:**

**A computer screen shot of a program code

AI-generated content may be incorrect.**

**Why Rc<T> Won’t Work in Multithreaded Code**

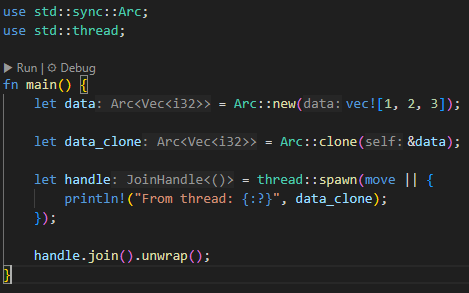
If we try to implement this scenario using our previous definition with Rc<T>, the compiler will reject it. That’s because Rc<T> is not thread-safe. It is intended for single-threaded use only. If we try to pass a Rc<T> between threads, we’ll get an error like:

A computer screen with white text

AI-generated content may be incorrect.

This happens because Rc<T> does not implement the Send and Sync traits, which are required for sharing across threads.

**The Fix: Use Arc<T> Instead:**

****

To solve this, we’ll replace Rc<T> with Arc<T> — an **atomic reference-counted** smart pointer. Arc<T> stands for *Atomic Reference Counted*, and is just like Rc<T>, but **safe for concurrent use** between threads.

Note about drop in Rc<T> and Arc<T>:

Give an example:

A screenshot of a computer code

AI-generated content may be incorrect.

The value inside a reference-counted pointer is only ever dropped once, so "drop" is only printed once. The initial drop(x) decrements the reference count, but does not drop the value because y is still live. Then dropping y finds that the reference count is 0, and drops in Example.

**RefCell<T> and the Interior Mutability Pattern:**

Interior mutability is a Rust design pattern that allows modification of data even when accessed through an immutable reference. This action is not allowed by Rust's usual compile-time borrowing rules. However, by using unsafe code, we can indicate to the compiler that we are checking the rules manually instead of relying on the compiler to check them for us.

RefCell<T> is a key example of this pattern, enabling runtime-checked borrowing while appearing immutable on the outside.

Note that unsafe here does not mean the API is unsafe to use. Instead, it means the implementers of RefCell<T> used unsafe internally to bypass compile-time checks—but wrapped it in a safe interface for users.

**Enforcing Borrowing Rules at Runtime with RefCell<T>:**

With references Box<T> and Rc<T> the borrowing rules’ invariants are enforced at compile time. With RefCell<T>, these invariants are enforced *at runtime*.  With references, if you break these rules, you’ll get a compiler error. With RefCell<T>, if you break these rules, your program will panic and exit.

**The advantage of checking the borrowing rules at compile time** will be caught sooner in the development process, and there is no impact on runtime performance because all the analysis is completed beforehand. As a result, checking the borrowing rules at compile time is the best choice in most cases, which is why this is Rust’s default.

**The advantage of checking borrowing rules at runtime** is that it allows some code to run safely, even if the compiler would normally reject it. This is because the Rust compiler, which analyzes code without running it, is cautious and may block code it can’t be sure is safe. Some things, like whether a program will ever finish running, can’t be figured out just by looking at the code — a classic example is the Halting Problem.

The RefCell<T> type is useful when you’re sure your code follows the borrowing rules but the compiler is unable to understand and guarantee that. But this comes at the cost of runtime panics if you make a mistake and affect the performance.

**Recap note:**

* Rc<T> enables multiple owners of the same data; Box<T> and RefCell<T> have single owners.
* Box<T> allows immutable or mutable borrows checked at compile time; Rc<T> allows only immutable borrows checked at compile time; RefCell<T> allows immutable or mutable borrows checked at runtime.
* Because RefCell<T> allows mutable borrows checked at runtime, you can mutate the value inside the RefCell<T> even when the RefCell<T> is immutable.

**Interior Mutability: Mutating Data in an "Immutable" Value:**

A consequence of the borrowing rules is that when you have an immutable value, you can’t borrow it mutably. For example:  
A number and equation on a white background

AI-generated content may be incorrect.

If you tried to compile this code, you would get the error “cannot borrow as mutable”.

**When Would Mutating Immutable Data Be Useful?**

Sometimes, especially in methods of a struct, it would be helpful if the struct could **change its internal state**, even when it's **being accessed immutably** from the outside. This is where **interior mutability** comes in.

One way to achieve this in Rust is using **RefCell<T>**, which allows you to mutate internal data **at runtime**. The compiler still sees your struct as immutable, but RefCell<T> lets you borrow mutably from the inside.

**Use Case: Mock Objects in Testing**

Let’s say you're writing a library that tracks how close a user is to hitting a limit (like an API quota). When the usage gets too high, the library sends warning messages.

You define a trait called Messenger that provides a send(&self, msg: &str) method, so the app can define how it sends messages (email, log, etc.).

A screenshot of a computer program

AI-generated content may be incorrect.

In tests, you want to use a **mock** version of Messenger that doesn’t actually send messages — it just **records** them so you can check what would’ve been sent.

You create a MockMessenger struct with a Vec<String> to store sent messages:

A screen shot of a computer code

AI-generated content may be incorrect.

But there’s a problem: the send method in the trait uses &self, which is an immutable reference, so you can’t push messages into the vector **— the compiler will complain**.

**Solution: Use RefCell<T> for Interior Mutability**

To fix this, you can wrap the Vec<String> in a RefCell:

A screenshot of a computer code

AI-generated content may be incorrect.

Now, even though send takes an immutable reference to self, you can still mutate the internal Vec safely at runtime: A computer code with black text

AI-generated content may be incorrect.

And in your test, you can borrow immutably to check the results:

A computer screen shot of a message

AI-generated content may be incorrect.

**Keeping Track of Borrows at Runtime with RefCell<T>:**

In Rust, we normally use & for immutable references and &mut for mutable ones. However, with RefCell<T>, we can use the .borrow() and .borrow\_mut() methods instead. These return Ref<T> and RefMut<T> smart pointers, which behave like regular references thanks to the Deref trait.

RefCell<T> keeps track of borrows at runtime. You can have many immutable borrows or one mutable borrow, just like Rust’s usual rules. But if you try to break these rules—like taking two mutable borrows at once—your program will panic instead of giving a compile-time error.

For example, in the code below, calling borrow\_mut() twice causes a runtime panic because it creates two mutable borrows in the same scope:

**A computer screen shot of a program code

AI-generated content may be incorrect.**

When you run this, the test compiles but fails with a panic:  
already borrowed: BorrowMutError.

This shows how RefCell<T> enforces Rust's borrowing rules—but at runtime. While this introduces a small performance cost and pushes error detection later in development, it lets you do things that normal references can’t. For example, it allows mocking behavior inside structs that must appear immutable.

**Allowing Multiple Owners of Mutable Data with Rc<T> and RefCell<T>**

We can also combine RefCell<T> with Rc<T> to allow **shared ownership** of data that is still **mutable**. Normally, Rc<T> only allows immutable access. But if we wrap our value in a RefCell<T>, we can mutate shared data safely—at runtime.

**Here's an example using a simple linked list:**

A screenshot of a computer program

AI-generated content may be incorrect.

This prints:

A screenshot of a computer code

AI-generated content may be incorrect.

Even though all three lists share the same inner value, we successfully updated it through RefCell<T>. This pattern is powerful when building flexible data structures.

Just remember: RefCell<T> is not thread-safe. For shared mutability across threads, use Mutex<T> instead.

**Reference Cycles and Weak References in Rust:**

Rust usually ensures memory safety and automatic cleanup, but it doesn’t fully prevent memory leaks. If you use Rc<T> and RefCell<T> together incorrectly, you can accidentally create **reference cycles**. These cycles happen when two or more values hold Rc pointers to each other, preventing their reference counts from ever reaching zero. Since Rust uses reference counts to know when to drop values, these cycles result in leaking memory.

**Creating a Reference Cycle**

To demonstrate this, a custom List enum is defined where each Cons item holds an i32 and a RefCell<Rc<List>>. This allows modifying the second element of the list:

A computer code with text

AI-generated content may be incorrect.

In main(), a is created with Cons(5, Nil) and wrapped in a Rc. Then, b is created pointing to a. After that, we mutate a to point to b, creating a cycle:

A screenshot of a computer program

AI-generated content may be incorrect.

When you run this, the strong\_count of both a and b becomes 2. Even after they go out of scope, the reference counts never hit zero, so the memory isn't free. If you try to print the entire structure, the program will enter an infinite loop and eventually cause a stack overflow.

 To visualize this reference cycle, authors have created the diagram in:

A diagram of a flowchart

AI-generated content may be incorrect.

In this code:

* b is created with its next item pointing to a (b → a).
* Then, a is modified so its next item points to b (a → b).

This creates a reference cycle: a → b → a

**Preventing Cycles with Weak<T>**

To avoid this, Rust offers Weak<T>. It’s a non-owning reference created with Rc::downgrade(). Unlike Rc<T>, a Weak<T> doesn't increase the strong reference count, so it doesn’t prevent values from being dropped.

Consider a tree structure where a Node owns its children (Rc<T>) and each child can reference its parent (Weak<T>). This avoids cycles because the child doesn’t keep the parent alive:

A screenshot of a computer code

AI-generated content may be incorrect.

Thinking about the relationship another way, a parent node should own its children: if a parent node is dropped, its child nodes should be dropped as well. However, a child should not own its parent: if we drop a child node, the parent should still exist. This is a case for weak references!

In main(), we create a leaf node and a branch node that has leaf as a child. Then, we set leaf’s parent to point to branch using a Weak reference:

A screenshot of a computer code

AI-generated content may be incorrect.

Now leaf can access its parent safely using upgrade():

The use of Weak avoids cycles and ensures memory is cleaned up. If branch is dropped (e.g., goes out of scope), leaf.parent becomes None, and the memory for branch is free since only a weak reference remains. However, if we drop a leaf node, the branch should still exist.

**Tracking Reference Counts**

You can also inspect reference counts using Rc::strong\_count(&x) and Rc::weak\_count(&x). For example, right after creating leaf, it has a strong count of 1. After assigning it as a child of branch, the strong count increases to 2. When branch is dropped, leaf strong count returns to 1, and memory is cleaned up correctly because the weak reference doesn’t block deallocation.