



CoE 164

Computing Platforms

Assessments Week 02

Academic Period: 2nd Semester AY 2023-2024

Workload: 3 hours

Synopsis: Rust compound data structures

SE Week 02A

This assessment will help you become familiar with mutating `structs` and `enums` in Rust.

This is worth 40% of your grade for this week

Problem Statement

Platform games (platformers) have an objective to move a player character between points in an environment. One of the old ways to implement platformers is by adding side-scrolling backgrounds and making the player character move from left to right (or move the background from right to left) on the screen. The most popular and iconic one involves an Italian plumber going through gigantic pipes.



Our goal is to be able to emulate a side-scrolling platform game. The player character has a name and can optionally carry at most one item with a certain quantity. At the start of the game, the player is located at position 0. The player can then do one of the following three moves: go left, go right, or use an item. Going left subtracts one to their position and going right adds one to their position. On the other hand, when an item is used, the existence of the item is checked first. If it does, then its quantity is subtracted by one.

The catch is that player information and state should be stored and manipulated using constructs in Rust, a programming language that you currently study. More specifically, a player is represented as a `struct` that holds its name, current item, and current position. The state of the current item is then represented by another `struct`, whose quantity on hand is then represented as an `enum`. Representing it in terms of "native" types enables faster and more convenient processing of data than if we were to store the information as a pile of different unconnected variables.

Input

The input starts with a number T on a single line denoting the number of players. T blocks of lines then follow. The first line of a block starts with three space-separated values `p_name ui_name ui_qty` denoting the name of the player, name of their item, and the number of that item that they initially have on hand. The next line contains a number `n_cmd` denoting the number of player moves that will follow. `n_cmd` lines then each follow, with each line C_i being only from one of the following values: `left`, `right`, `uitem`.

Output

The output consists of T blocks of lines, with each block denoting the corresponding player in the input. Each block should start with a line containing the text `Player #t:` where t is the serial of the player starting from 1. The next line then contains the text `Player:` `<p_name>` where `<p_name>` is player name. The next line then contains information about the item the player initially has, which can be from one of the following

- Item: `<ui_qty> <ui_name>`
 - If `ui_qty` is greater than 0, then this is printed with the name of item `ui_qty` and `ui_qty` the number of initial items
- Item: NONE
 - If `ui_qty` is zero

The next line then contains the string `-----LOG-----` (string `LOG` surrounded by ten dashes each) that separates player information from movement information.

Then, the next `n_cmd` lines correspond to the respective player moves in the input. For each move, the text should be only from one of the following:

- For `left` move
 - New position: `<pos-1>`
- For `right` move
 - New position: `<pos+1>`
- For `uitem` move
 - If current `ui_qty` is zero
 - Cannot use item as player does not have one.
 - Otherwise, if current `ui_qty-1` is zero
 - Player used `<<ui_name>>`. It is now gone.
 - Otherwise, if `ui_qty-1` is greater than zero
 - Player used `<<ui_name>>`. `<ui_qty-1>x` of `<<ui_name>>` remains.

Please see the sample output for more details.

Constraints

Input Constraints

$T \leq 10$

$ui_qty \in \mathbb{Z}^+ \cup \{0\}; ui_qty \geq 0$

$n_cmd \in \mathbb{Z}^+ \cup \{0\}; n_cmd \leq 50$

$|p_name|, |ui_name| \leq 50$

$C_i \in \{\text{left, right, uitem}\}$

Characters in `p_name` and `ui_name` are always from within the set `[a-zA-Z0-9_]`.

You can assume that all of the inputs are well-formed and are always provided within these constraints. You are not required to handle any errors.

Functional Constraints

You are **required** to create and use the following `structs` and `enums`:

- `Player` - struct representing information about a player
 - `name: String` - player name
 - `pos: i64` - current position of player
 - `item: Option <PlayerItem>` - current item of player
- `PlayerItem` - struct representing information about an item
 - `name: String` - item name
 - `item_type: PlayerItemQtyType` - number of this item on hand
- `PlayerItemQtyType` - enum representing quantity of an item
 - `Once` - the item can be used once
 - `Consumable(u64)` - the item has at least more than one copy
 - First argument - current number of items

You are **required** to have the following function signatures, their arguments in order, and their return values:

- `main()` - entry point to the program
 - Arguments
 - None
 - Return value
 - None
 - Additional constraints
 - Input and output parsing should be done here

You are required to create a variable of type `Player` and manipulate it when dealing with player moves.

Failure to follow these functional constraints will result in a score of zero in this assessment.

Sample Input/Output

Sample Input 1:

```
1
MyPlayer_02 potion 3
10
left
left
right
right
right
right
uitem
right
uitem
left
```

Sample Output 1:

```
Player #1:
Name: MyPlayer_02
Item: 3x potion
-----LOG-----
New position: -1
New position: -2
New position: -1
New position: 0
New position: 1
New position: 2
Player used <potion>. 2x of <potion> remains.
New position: 3
Player used <potion>. 1x of <potion> remains.
New position: 2
```

Sample Input 2:

```
2
MyPlayer bullets 5
6
left
uitem
uitem
uitem
uitem
uitem
MyPlayer item 0
0
```

Sample Output 2:

```
Player #1:
Name: MyPlayer
Item: 5x bullets
-----LOG-----
```

```
New position: -1
Player used <bullets>. 4x of <bullets> remains.
Player used <bullets>. 3x of <bullets> remains.
Player used <bullets>. 2x of <bullets> remains.
Player used <bullets>. 1x of <bullets> remains.
Player used <bullets>. It is now gone.
Player #2:
Name: MyPlayer
Item: NONE
-----LOG-----
```

Steps

1. Write your program in Rust. Compile and make sure that there are no syntax errors.
2. Make sure to accept input via standard input and print your output via standard output. For example, you can write your inputs into a text file named `in_pub.txt` and the expected and correct outputs into another text file named `out_pub_ans.txt`. If the compiled program is named `wa`, and you want the printed output to be saved into a file named `out_pub.txt`, you can execute the following command from the following terminals to run it:

Windows (Powershell): `cat in_pub.txt | ./wa.exe | Out-File out_pub.txt`

Linux/macOS (bash, zsh): `./wa < in_pub.txt > out_pub.txt`

Then, compare the program output with the reference output by executing the following commands:

Windows (Powershell): `Compare-Object (gc out_pub.txt) (gc out_pub_ans.txt)`

Linux/macOS (bash, zsh): `diff out_pub.txt out_pub_ans.txt`

3. Submit a copy of the source code to the [Week 02A submission bin](#). Make sure that you attach one (1) file in the bin containing the Rust source code with a `.rs` extension (preferably named `w02a.rs`). **Please do not send compressed files!**

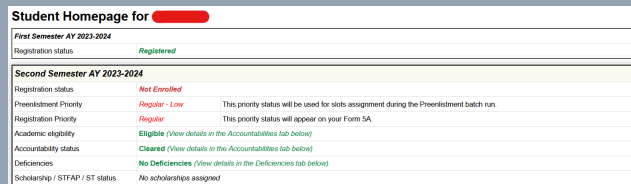
SE Week 02B

This assessment will help you become familiar with `structs` in Rust.

This is worth 30% of your grade for this week.

Problem Statement

The Computerized Registration System (CRS) is the official registration system of UP Diliman. When you log in to the system via your browser, you will be greeted with a summary of your



Student Homepage for [REDACTED]	
First Semester AY 2023-2024	
Registration status	Registered
Second Semester AY 2023-2024	
Registration status	Not Enrolled
Pre-enrollment Priority	Regular - Low This priority status will be used for slots assignment during the Pre-enrollment batch run
Registration Priority	Regular This priority status will appear on your Form SA
Academic eligibility	Eligible (View details in the Accountabilities tab below)
Accountability status	Cleared (View details in the Accountabilities tab below)
Delinquencies	No Delinquencies (View details in the Delinquencies tab below)
Scholarship / STAP / ST status	No scholarships assigned

registration details for the current semester. Two of the important information you need to check before every enrollment period is eligibility and accountability since you can encounter hindrances in your enrollment, and hence registration, if you are ineligible or have accountabilities.

As an all-in-one system, degree completion is also recorded through the CRS, which includes you being able to view the grades for all of your courses that you have taken during your stay in the university. Under the "Grades Viewing" module, you can check whether you have satisfied the required general education (GE) courses for your degree program. As of this current time of writing, students are required to take the following GE courses. Note that some courses can be taken as substitute for another one denoted by a slash:

- ARTS 1
- Fil 40
- Kas 1
- Philo 1
- Eng 13/Speech 30
- STS 1/DRMAPS
- Soc Sci 1/Soc Sci 2

Your current goal is to be able to replicate a command line version of the CRS frontpage. You have surmised that initial tests of the CRS include having a program that accepts commands that change the state of a student. For example, a command can edit student information. A command can also set the eligibility and accountability of a student. Finally, there is also probably a command that can set the courses already completed or passed by the student.

Input

The input starts with a number T on a single line denoting the number of students. T blocks of lines then follow. The first line of a block starts with two space-separated numbers

n_cmd sn denoting the number of commands that will follow and the student number of the student. n_cmd lines then each follow, with each line of the general format cmd arg with cmd being the command and arg the argument for the relevant command. cmd shall be from one of the three following formats:

- e $\langle y/n \rangle$ - set eligibility of student; y if they are and n otherwise
- a $\langle y/n \rangle$ - set accountability of student; y if they are and n otherwise
- c $\langle course \rangle$ - notes that student passed or satisfied $\langle course \rangle$ course

Output

The output consists of T blocks of lines, with each block denoting the corresponding student in the input. Each block should start with a line containing the text *Student #t*: where t is the serial of the student starting from 1. The next line then contains the text *Record for SN $\langle sn \rangle$* where $\langle sn \rangle$ is the student number of the corresponding student as a 9-digit number. Then, the next four lines shall be indented four spaces from the leftmost part and should display the following in order.

- *Is eligible? $\langle YES/NO \rangle$*
 - Replace $\langle YES/NO \rangle$ with *YES* if the student is eligible and *NO* otherwise
- *Is accountable? $\langle YES/NO \rangle$*
 - Replace $\langle YES/NO \rangle$ with *YES* if the student is accountable and *NO* otherwise
- *Unsatisfied GE2017 Courses:*
- $\langle courses \rangle$
 - Space-separated list of core GE 2017 courses that the student has *not yet* passed, or *NONE* if everything is satisfied
 - The exact list of courses assuming that the student has not yet passed *any* course GE 2017 course is as follows:
 - ARTS1 FIL40 KAS1 PHILO1 ENG13/SPEECH30 STS1/DRMAPS SOCSCI2/SOCSCI1
 - Relevant courses are deleted from the above list the student has passed those
 - Courses are written in all uppercase letters

Please see the sample output for more details.

Constraints

Input Constraints

$$T \leq 10$$

$$sn \in \mathbb{Z}; sn \in (0, 1\,0000\,00000]; |sn| = 9$$

$$n_cmd \leq 50$$

$$cmd \in \{d, a, c\}$$

`arg` is always a string within the set `[a-zA-Z0-9_]`. This includes the `<course>` and `<y/n>` arguments in a command.

You can assume that all of the inputs are well-formed and are always provided within these constraints. You are not required to handle any errors.

Functional Constraints

You are **required** to create and use the following structs:

- `StudentEnrollInfo` - struct representing an enrollment information of a student
 - `sn: u64` - 9-digit student number
 - `is_eligible: bool` - whether the student is eligible
 - `has_accountables: bool` - whether the student has accountables
 - `has_taken_ge2017: [bool; 10]` - each element represents a GE 2017 course; `true` if the student has passed or satisfied the corresponding course
 - Each element corresponds to the following courses in the following order: ARTS1 FIL40 KAS1 PHILO1 ENG13 SPEECH30 STS1 DRMAPS SOCSCI2 SOCSCI1

You are **required** to have the following function signatures, their arguments in order, and their return values:

- `StudentEnrollInfo::new()` - initialize a new `StudentEnrollInfo` struct with default values
 - Arguments
 - None
 - Return value - `StudentEnrollInfo` representing a "default" student that is not eligible and has accountabilities, and has taken *no* GE 2017 course
- `StudentEnrollInfo::check_ge2017()` - check whether the entered satisfied course is part of GE 2017
 - Arguments
 - `&mut self: &mut self` - the "self" instance of the struct
 - `course: String` - the course to check that has been satisfied; it can not be part of GE 2017
 - Return value - `bool` representing whether the checklist of GE 2017 courses of the student should be updated
 - Additional constraints
 - `self.has_taken_ge2017` should be updated if `ge` is a GE 2017 course
- `StudentEnrollInfo::print_unsatisfied_ge2017()` - print the list of unsatisfied GE 2017 courses to the standard output
 - Arguments
 - `&self: &self` - the "self" instance of the struct

- Return value - `bool` representing whether the GE 2017 course checklist is satisfied
- Additional constraints
 - Does not print a newline after this function ends
- `main()` - entry point to the program
 - Arguments
 - None
 - Return value
 - None
 - Additional constraints
 - Input and output parsing should be done here

Failure to follow these functional constraints will result in a score of zero in this assessment.

Sample Input/Output

Sample Input 1:

```
1
4 200345938
e y
a y
c drmaps
c eng13
```

Sample Output 1:

```
Student #1:
Record for SN 200345938
  Is eligible? YES
  Is accountable? YES
  Unsatisfied GE2017 Courses:
  ARTS1 FIL40 KAS1 PHILO1 SOCSCI2/SOCSCI1
```

Sample Input 2:

```
2
5 200530495
c kas1
e n
a y
c philo1
a n
0 201134162
```

Sample Output 2:

```
Student #1:
Record for SN 200530495
  Is eligible? NO
  Is accountable? NO
  Unsatisfied GE2017 Courses:
  ARTS1 FIL40 ENG13/SPEECH30 STS1/DRMAPS SOCSCI2/SOCSCI1
```

```
Student #2:
Record for SN 201134162
  Is eligible? NO
  Is accountable? YES
Unsatisfied GE2017 Courses:
ARTS1 FIL40 KAS1 PHILO1 ENG13/SPEECH30 STS1/DRMAPS
SOCSCI2/SOCSCI1
```

Steps

1. Write your program in Rust. Compile and make sure that there are no syntax errors.
2. Make sure to accept input via standard input and print your output via standard output. For example, you can write your inputs into a text file named `in_pub.txt` and the expected and correct outputs into another text file named `out_pub_ans.txt`. If the compiled program is named `wa`, and you want the printed output to be saved into a file named `out_pub.txt`, you can execute the following command from the following terminals to run it:

Windows (Powershell): `cat in_pub.txt | ./wa.exe | Out-File out_pub.txt`

Linux/macOS (bash, zsh): `./wa < in_pub.txt > out_pub.txt`

Then, compare the program output with the reference output by executing the following commands:

Windows (Powershell): `Compare-Object (gc out_pub.txt) (gc out_pub_ans.txt)`

Linux/macOS (bash, zsh): `diff out_pub.txt out_pub_ans.txt`

3. Submit a copy of the source code to the Week 02B [submission bin](#). Make sure that you attach one (1) file in the bin containing the Rust source code with a `.rs` extension (preferably named `w02b.rs`). **Please do not send compressed files!**

SE Week 02C

This assessment will let you be familiar with structs, enums, impl, and error handling in Rust.

This is worth 30% of your grade for this week

Problem Statement

Tara shoppEEEng!

An intelligent grocery store, "grocerEEE store," has just opened in your neighborhood, and they are hiring a Rust programmer to develop their smart cart system. Curious about the system, you take a visit to the store and interview their development team.



The development team revealed that the system can automatically detect the shopper's action and make a log file. The system can also show the current items in the cart by pressing an on-board cart button; the shown items, however, are abstracted according to their item type. Furthermore, to enhance customer experience, they added a budget tracker feature to ensure they stay within their allotted allowance. The physical limitations of the cart were also considered in their system design. When an item is added, the system indicates if the current items exceed the cart's weight capacity, which is around 12.0 kg. Moreover, the cart can only accommodate up to 10 items to ensure the cart's space is not *completely* crammed.

With this, the development team deemed that their system should be memory-safe; however, none had sufficient proficiency in Rust. Luckily, you have the knowledge to code¹ in Rust and offer your services to them. As such, your task is to develop their system in Rust, which generates a log file based on the user's action. Furthermore, you noticed that sometimes the cart's sensor provides gibberish input to the program, i.e., the command or number is not interpreted correctly. With this, you will also implement error-handling schemes to make the system more robust.

¹ Hopefully... 😊

Input

The input starts with stating the user's allotted budget B . It will then be followed by a series of T lines as indicated by the user. In each line, the following commands can be executed:

- `add <grocery_item> <price> <weight>`
 - Adds the specified `grocery_item` to the cart. The added item will then be abstracted according to its `ItemType` as shown in the table below.

<code>grocery_item</code>	<code>ItemType</code>
coke, sprite, royal	Beverage
bleach, soap	Cleaners
battery, bulb	Electronics
banana, mango, strawberries	Fruits
beef, chicken, pork	Meat

- `remove <grocery_item_number>`
 - Removes the `grocery_item` at the specified index, `grocery_item_number`, starting at 1 and ending at 10.
- `show_info`
 - Generates the content of the grocery cart, its current value, and the total weight of the items.

If the input command generates an error to the system, the erroneous command is disregarded and should **not** be counted as part of the T commands.

Output

The output consists of R_i lines indicating the response of the system for each T_i commands. Depending on the command, the system will generate the following response:

- add <grocery_item> <price> <weight>
 - If successfully added: [SYSTEM] Item successfully added!
 - If unsuccessful:
 - If the added item will cause an overbudget: [SYSTEM] Maximum budget reached! Item unsuccessfully added.
 - If the added item exceeds the cart's weight limit: [SYSTEM] Maximum weight reached! Item unsuccessfully added.
 - If the added item will exceed 10 cart items: [SYSTEM] Maximum number of items reached! Item unsuccessfully added.
 - Otherwise: [ERROR] <system_error>!
 - where system_error is the first error encountered which could be one of the following:
 - Item not classified - if not found in grocery_item
 - Price error - if there is an error reading the price
 - Weight error - if there is an error reading the weight
- remove <grocery_item_number>
 - If successfully removed: [SYSTEM] Item removed!
 - If there is no item to remove: [SYSTEM] No item removed!
 - Otherwise: [ERROR] Index does not exist!
- show_info
 - A series of strings will be generated as shown:

```
-----GROCERY CART-----
1: <ItemType>
2: <ItemType>
3: <ItemType>
...
n: <ItemType>
Total price: Php <current_value>
Total weight: <current_weight> kg
-----
```
- Any other unlisted command: [ERROR] Command not found!

where n is the index of the last non-empty element in the cart, the `current_value` is current price of the cart rounded down to two decimal places, and `current_weight` is the current weight of the items rounded up to two decimal places. If there are no items in the cart, it will simply display the total price and weight of the cart.

Constraints

Input Constraints

$0.0 \leq B \leq 5000.00$, where $B \in \mathbb{R}^+$

$0 < T \leq 20$, where $T \in \mathbb{Z}$

$T_i = \{\text{add } I P W, \text{remove } N, \text{show_info}\}$

$I = \{\text{coke, sprite, royal, bleach, soap, battery, bulb, banana, mango, strawberries, beef, chicken, pork}\}$

$0 < P \leq 750$, where $P \in \mathbb{R}^+$

$0 < W \leq 10$, where $W \in \mathbb{R}^+$

$W_{\text{cart,max}} = 12.0 \text{ kg}$

$0 < N \leq 10$, where $N \in \mathbb{Z}$

You are **required** to handle errors regarding system commands. To reiterate, if the system encounters an error, the command is disregarded and not counted as part of the T commands. Note that B and T are always well-formed. You are not required to handle any other errors not outlined in this problem statement.

Functional Constraints

You are **required** to create the following structs, enums, and impl:

- enum `ItemType`, which contains the five main variants of an item: `Beverage`, `Cleaners`, `Electronics`, `Fruits`, and `Meat`. If an item does not belong to any of these variants, indicate it as `None`.
- struct `GroceryItem` with fields `item: ItemType`, `price: f64`, and `weight: f64`
- struct `SmartCart` with fields `items: [GroceryItem; 10]`, `max_budget: f64`, `max_weight: f64`, `current_value: f64`, `current_weight: f64`, `current_size: usize`.
- impl `SmartCart` with methods and parameters:
 - `fn new(max_budget: f64) -> SmartCart` - initializes the cart
 - `fn add_item(grocery_item: GroceryItem)` - adds item to the cart
 - `fn remove_item(index: usize)` - removes item from the cart
 - `fn show_info()` - displays the contents of the cart

Failure to follow these functional constraints will result in a score of zero in this assessment.

Sample Input/Output

Sample Input 1:

```
1000.00
2
addd coke 150 0.5
add cok3 150 0.S
add coke 150 0.S
add coke 150 0.S
add coke 150 0.5
show_information
show_info
```

Sample Output 1:

```
[ERROR] Command not found!
[ERROR] Item not classified!
[ERROR] Price error!
[ERROR] Weight error!
[SYSTEM] Item successfully added!
[ERROR] Command not found!
-----GROCERY CART-----
1: Beverage
Total price: Php 150.00
Total weight: 0.50 kg
-----
```

Sample Input 2:

```
750.50
11
show_info
remove 11
add pork 200 0.72
remov3 2
add battery 55 0.1
add sprite 70 B
add royal 70 0.1
add mango 400 2.0
show_info
remove 3
add soap 50 5.1
add calamansi 10 10
add banana 40.5 5
add banana 20.5 2.5
show_info
```

Sample Output 2:

```
-----GROCERY CART-----
Total price: Php 0.00
```

Total weight: 0.00 kg

[ERROR] Index does not exist!
[SYSTEM] Item successfully added!
[ERROR] Command not found!
[SYSTEM] Item successfully added!
[ERROR] Weight error!
[SYSTEM] Item successfully added!
[SYSTEM] Item successfully added!

-----GROCERY CART-----

1: Meat
2: Electronics
3: Beverage
4: Fruits
Total price: Php 725.00
Total weight: 2.92 kg

[SYSTEM] Item removed!
[SYSTEM] Item successfully added!
[ERROR] Item not classified!
[SYSTEM] Maximum weight reached! Item unsuccessfully added.
[SYSTEM] Item successfully added!

-----GROCERY CART-----

1: Meat
2: Electronics
3: Fruits
4: Cleaners
5: Fruits
Total price: Php 725.50
Total weight: 10.42 kg

Steps

1. Write your program in Rust. Compile and make sure that there are no syntax errors.
2. Make sure to accept input via standard input and print your output via standard output. For example, you can write your inputs into a text file named `in_pub.txt` and the expected and correct outputs into another text file named `out_pub_ans.txt`. If the compiled program is named `wa`, and you want the printed output to be saved into a file named `out_pub.txt`, you can execute the following command from the following terminals to run it:

Windows (Powershell): `cat in_pub.txt | ./wa.exe | Out-File out_pub.txt`

Linux/macOS (bash, zsh): `./wa < in_pub.txt > out_pub.txt`

Then, compare the program output with the reference output by executing the following commands:

Windows (Powershell): `Compare-Object (gc out_pub.txt) (gc out_pub_ans.txt)`

Linux/macOS (bash, zsh): `diff out_pub.txt out_pub_ans.txt`

3. Submit a copy of the source code to the Week 02C [submission bin](#). Make sure that you attach one (1) file in the bin containing the Rust source code with a `.rs` extension (preferably named `w02c.rs`). **Please do not send compressed files!**