DPST1091 / CPTG1391

Introduction to Programming

Week 5 – Lecture 2

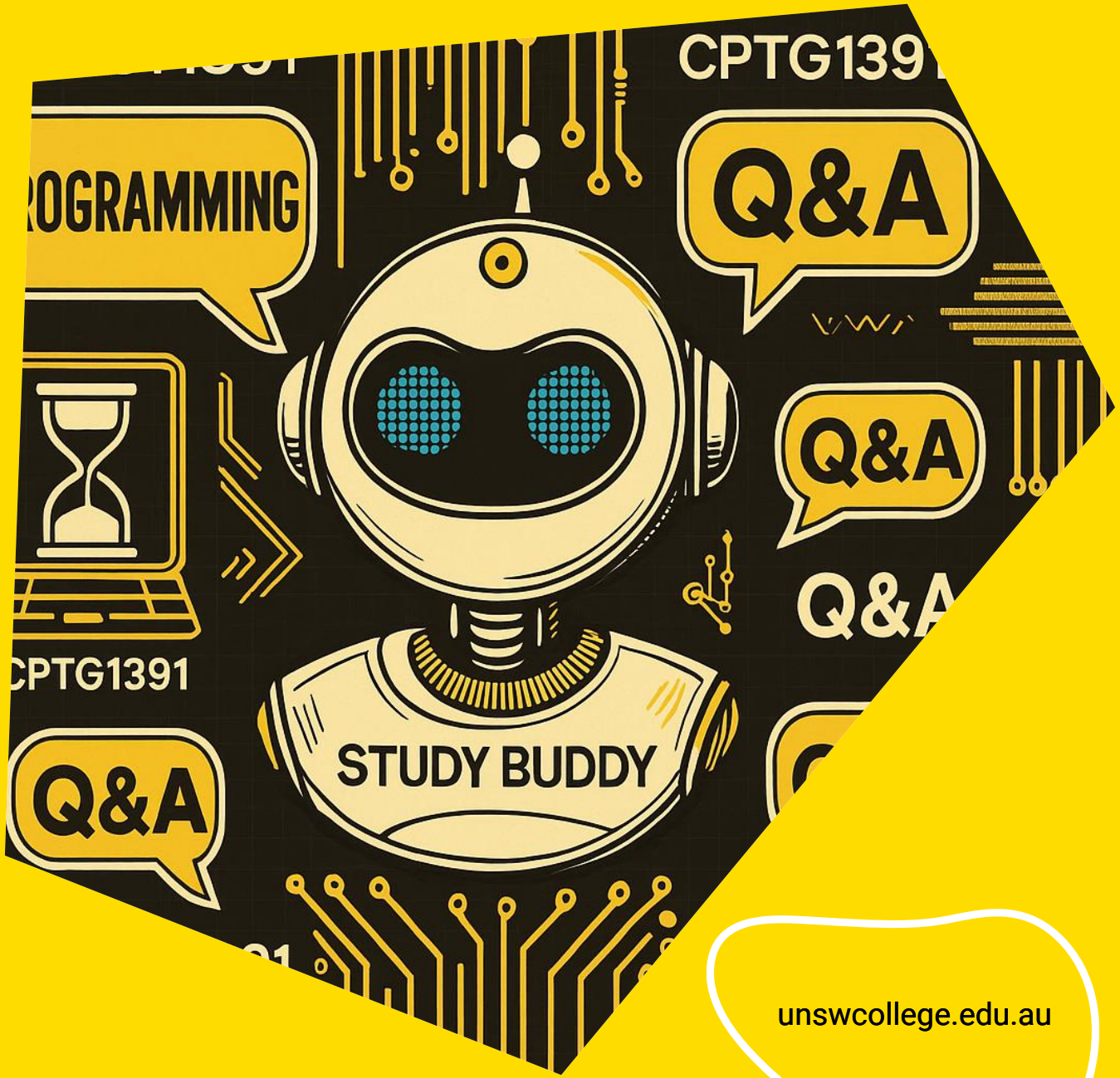
Lecturer and Course Convener:

Dr Pantea Aria

UNSW College

Command Line Arguments

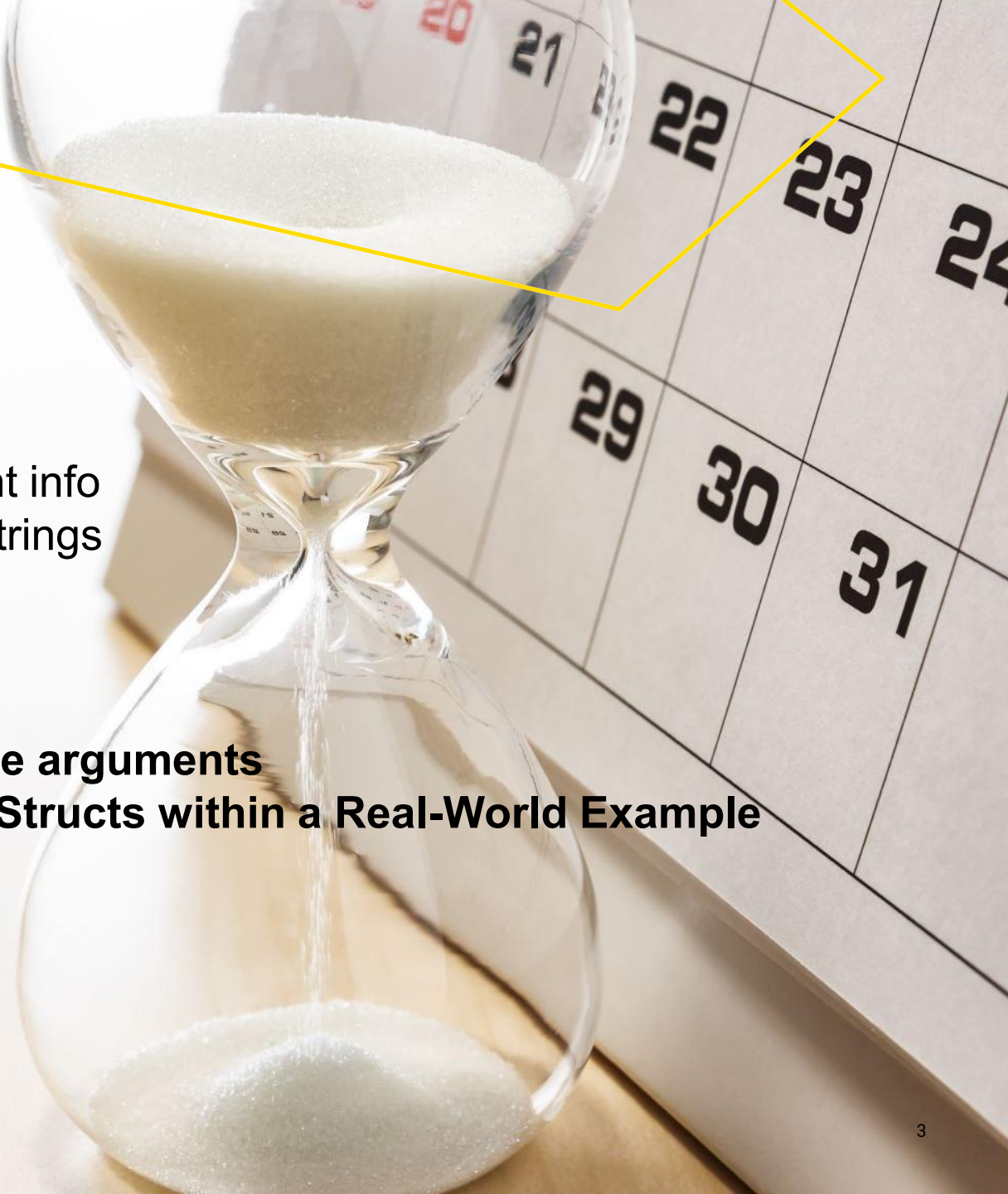
2D Arrays of Structs



unswcollege.edu.au

Agenda

- **Last lecture**
  - ⟶ Assignment info
  - ⟶Arrays of Strings
- **Today**
  - **Command line arguments**
  - **2D Arrays of Structs within a Real-World Example**



Strings recap

Strings represent **sequences of characters**

In C, a string is defined as: an **array** of characters (**char**)

terminated by the **null character '\0'** stored in consecutive memory locations

Example

```c
 // Declare an array to store a string
char pet[MAX_LENGTH] = "Oreo";

// Copy a new string into the array
// Make sure the array is large enough to avoid overflow
strcpy(pet, "Buddy");

printf("%s\n", pet);

// Find the length of the string (does NOT count '\0')
int length = strlen(pet);
printf("%s has length %d\n", pet, length);
```

Example

```c
#include <stdio.h>
#include <string.h>

int main(void) {
    char name1[] = "pantea";
    char name2[] = "Pantea";
    char name3[] = "sara";

    // Case 1: strings are equal
    if (strcmp(name1, "pantea") == 0) {
        printf("name1 is equal to \"pantea\"\n");
    }

    // Case 2: first string comes before second (negative result)
    if (strcmp(name1, name3) < 0) {
        printf("\"%s\" comes before \"%s\"\n", name1, name3);
    }

    // Case 3: first string comes after second (positive result)
    if (strcmp(name1, name2) > 0) {
        printf("\"%s\" comes after \"%s\"\n", name1, name2);
    }

    return 0;
}
```

Array of Strings recap

An **array of strings** is a **two-dimensional array of characters**

Rows = individual strings

Columns = characters within each string

Example

```
char names[3][10] = {"pantea", "sara", "alex"};
```

“Pantea”	“sara”	“alex”
0	1	2

- names can **store 3 strings**
- Each string can store **up to 9 characters plus the null terminator ‘\0’**

```
names[0] → "pantea"
names[1] → "sara"
names[2] → "alex"
```

	col 0	col 1	col 2	col 3	col 4	col 5	col 6	col 7	col 8	col 9
row 0	‘P’	‘a’	‘n’	‘t’	‘e’	‘a’	‘\0’			
row 1	‘s’	‘a’	‘r’	‘a’	‘\0’					
row 2	‘a’	‘l’	‘e’	‘x’	‘\0’					

Another example

```c
char names[3][10] = {"pantea", "sara", "alex"};

// Using one index selects a whole string (a row)
// This prints "sara"
printf("%s\n", names[1]);
```

```c
char names[3][10] = {"pantea", "sara", "alex"};

// Using two indexes selects a single character
// This prints the 'e' from "alex"
printf("%c\n", names[2][2]);
```

Command Line Arguments

Giving Input When the Program Starts

So far, our programs have only received input after they start running.
We usually do this using functions like **scanf()** or **fgets().**
Because of this, our **main** function has always looked like this: **int main(void);**

Command line arguments let us provide **input at the moment we run the program**, instead of waiting for user input inside the program.

For example, when we run a program from the terminal:

```
$ dcc program.c -o program
$ ./program 42 apple
```

The values `42` and `apple` are **passed directly into the program** as **command line arguments**.

Changing the main Function

To use command line arguments, we must change the **main** function header to:

```
int main(int argc, char *argv[])
```

`argc`

- Short for ***argument count***
- Stores **how many** command line arguments were provided
- **Includes the program name itself**

`argv`

- Short for ***argument vector***
- An array that stores all the command line arguments
- Each argument is stored as a **string** (an array of **`char`**)

Example

If we run this: `$ ./program 42 apple "Pantea Aria"`

What Does the Program Receive?

- **`argc`** will be **4**
- **`argv`** will look like this:

Index	Value
argv[0]	"./program"
argv[1]	“42"
argv[2]	“apple"
argv[3]	“Pantea Aria"

`argv array`

"./program"	“42"	“apple"	“Pantea Aria"
0	1	2	3

Example

```c
#include <stdio.h>

int main(int argc, char *argv[]) {

    printf("Number of arguments: %d\n", argc);

    for (int i = 0; i < argc; i++) {
        printf("argv[%d] = %s\n", i, argv[i]);
    }

    return 0;
}
```

```
Week5 $ ./command_line_args 42 apple "Pantea Aria" -5
Number of arguments: 5
argv[0] = ./command_line_args
argv[1] = 42
argv[2] = apple
argv[3] = Pantea Aria
argv[4] = -5
Week5 $
```

Converting Strings to Integers with *atoi*

Command line arguments are often used in calculations
However, **all command line arguments** are received as **strings**
To work with **numbers**, we must **convert strings to integers**

The `atoi()` Function

- `atoi()` converts a string into an integer
- It is provided in the standard library `<stdlib.h>`

```
int x = atoi("952");
```

- After this line runs, `x` will store the integer value **952**

Using function *atoi* to convert command line args to integers

```c
#include <stdio.h>
#include <stdlib.h>

int main(int argc, char *argv[]) {
    int sum = 0;

    for (int i = 1; i < argc; i++) {
        sum = sum + atoi(argv[i]);
    }

    printf("%d is the sum of all command line arguments\n", sum);

    return 0;
}
```

Demo

→steps.c

Live lecture code is written for teaching, not perfection.
It may include extra comments and may not always follow ideal coding style

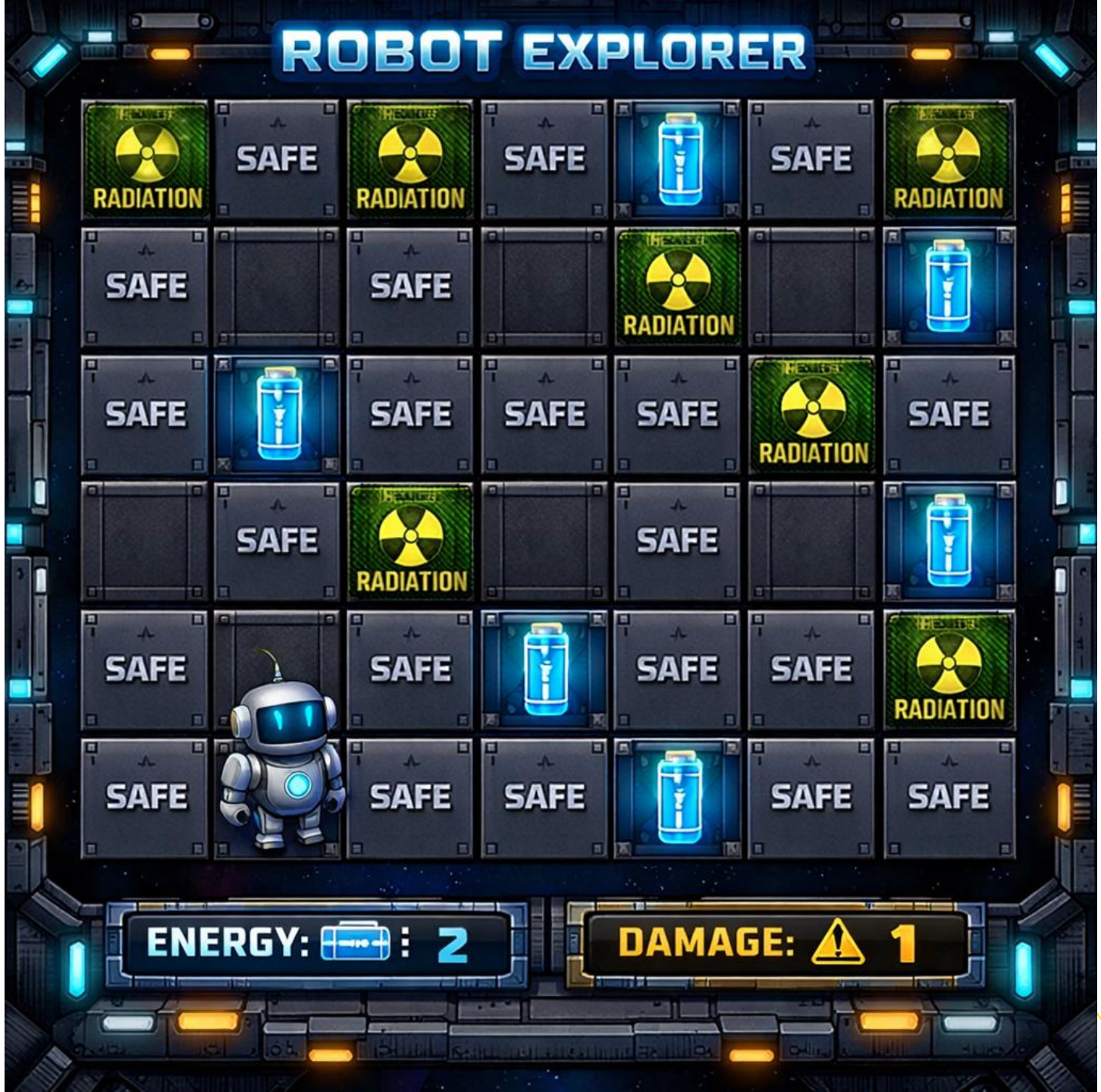
UNSW College


IT'S BREAK TIME!
#include <stdio.h>
#define ON_BREAK 1
int main(){
// Time for a 10 minute break! Switch to PARTY_MODE
#define PARTY_MODE ON_BREAK
if {PARTY_MODE == ON_BREAK) ;
print("Program will resume in 10 minutes...");
sleep(600,); // Take a break
exit(0);
}
10 MINUTES BREAK!
BRB
CHIPS
Relax... We'll be back soon!

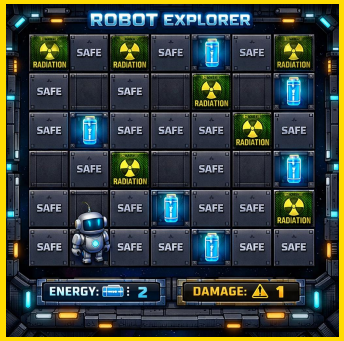
2D Arrays of structs

A Practical example
2D Arrays, Enums, Structs

Robot Explorer

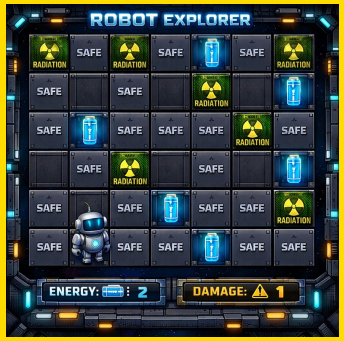


Story / Scenario:



1. A **robot** is exploring an **abandoned space station**.
2. The station is represented as an **8×8 grid**.
3. Each location can contain:
   - **Floor type**: safe floor or radiation
   - **Item**: empty or energy cell
4. The robot moves around the station collecting **energy cells**.
5. Stepping on **radiation** damages the robot.

Game Rules



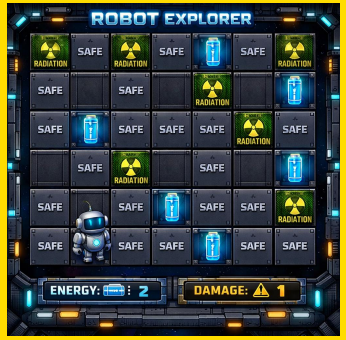
Map

- The map is an **8×8 2D array of structs**
- Each cell stores:
  - the **floor type**
  - the **item type**

Player Actions

- The player controls the robot using:
  - `w` → up
  - `a` → left
  - `s` → down
  - `d` → right

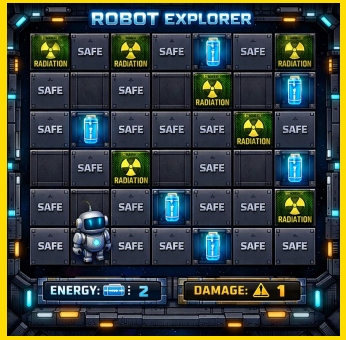
What Happens When the Robot Moves?



- If the robot lands on an **energy cell**:
  - Energy count increases
  - The cell becomes empty

- If the robot lands on **radiation**:
  - Damage count increases

Game End Conditions



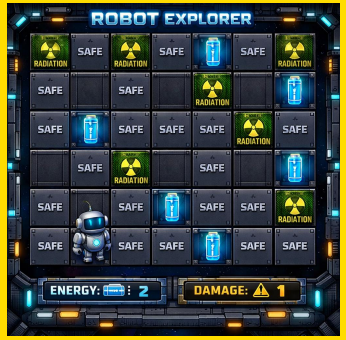
The game ends when **one** of the following happens:

✅ **Win**: The robot collects all energy cells

❌ **Lose**: The robot takes radiation damage **3 times**

⛔ **Exit**: The player presses **Ctrl-D**

Given Types and Constants



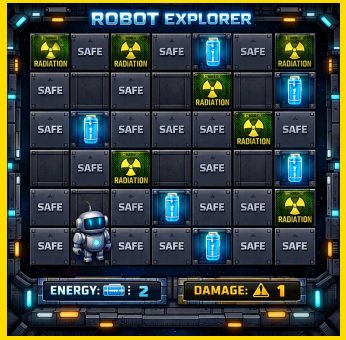

```c
#define MAP_ROWS 8
#define MAP_COLUMNS 8

enum floor_type {
    SAFE,
    RADIATION
};

enum item_type {
    EMPTY,
    ENERGY
};

struct location {
    enum floor_type floor;
    enum item_type item;
};
```

The Map Representation




```
struct location station[MAP_ROWS][MAP_COLUMNS];
```

Conceptually:

Each cell is **one** `struct location`.

	Col 0	Col 1	Col 2	Col 3	Col 4	Col 5	Col 6	Col 7
Row 0								
Row 1								
Row 2								
Row 3								
Row 4								
Row 5								
Row 6								
Row 7								

```c
#define MAP_ROWS 8
#define MAP_COLUMNS 8

enum floor_type {
    SAFE,
    RADIATION
};

enum item_type {
    EMPTY,
    ENERGY
};

struct location {
    enum floor_type floor;
    enum item_type item;
};
```

```c
struct location station[MAP_ROWS][MAP_COLUMNS];
```

We can initialise some of the cells.

	Col 0	Col 1	Col 2	Col 3	Col 4	Col 5	Col 6	Col 7
Row 0		RADIATION EMPTY	SAFE ENERGY	SAFE EMPTY	RADIATION ENERGY	SAFE EMPTY	SAFE ENERGY	RADIATION EMPTY
Row 1	SAFE EMPTY		RADIATION EMPTY		SAFE EMPTY	RADIATION ENERGY		
Row 2	RADIATION ENERGY	SAFE EMPTY	SAFE EMPTY	RADIATION EMPTY	SAFE ENERGY	SAFE EMPTY	RADIATION EMPTY	SAFE EMPTY
Row 3	SAFE ENERGY	RADIATION EMPTY	SAFE EMPTY	SAFE EMPTY				RADIATION EMPTY
Row 4			RADIATION EMPTY	SAFE EMPTY	SAFE EMPTY	RADIATION EMPTY	SAFE ENERGY	SAFE EMPTY
Row 5	RADIATION EMPTY	SAFE EMPTY	SAFE ENERGY	RADIATION ENERGY	SAFE EMPTY	SAFE EMPTY		SAFE ENERGY
Row 6		RADIATION ENERGY	SAFE EMPTY	SAFE ENERGY		SAFE EMPTY		RADIATION EMPTY
Row 7			RADIATION EMPTY	SAFE EMPTY	SAFE ENERGY	RADIATION EMPTY	SAFE EMPTY	RADIATION ENERGY

Station[5][3].floor_type = RADIATION;
Station[5][3].item_type = ENERGY;

Provided Function Prototypes

```c
void initialise_station(struct location station[MAP_ROWS][MAP_COLUMNS]);

void print_station(
    struct location station[MAP_ROWS][MAP_COLUMNS],
    int robot_row,
    int robot_col,
    int energy_count,
    int damage_count
);
```

starter code

1. Creates a **station variable**
2. Calls **initialise** on the station
3. Calls **print_station**, passing in INVALID_INDEX for robot_row and robot_col and 0 for energy_count (0) and damage_count (0)

Stage 1

1. Read the **starting row and column** from the user and **place the robot** at that position.
2. If the **coordinates are invalid**, place the **robot at (0, 0)** instead.
3. **Update the station** and relevant counters:
    1. Increase the energy count and remove any energy cell that the robot collects.
    2. Update the floor type as the robot moves through radiation.
4. Display the updated station.

Stage 2 – Movement Loop

1. In a loop, read **movement** commands (`w, a, s, d`)
2. Move robot **if inside bounds**
3. Increase the energy count and remove the energy cell from the map when it is collected.
4. Increase the damage count if the robot steps on radiation (do not implement radiation spreading yet).
5. Print the map after each movement.
6. Note: You may assume the user always enters valid input. If invalid input is provided, the program behaviour is undefined.

Stage 3 – Game Logic

1. Implement:
   - Winning condition (all energy collected)
   - Losing condition (3 radiation hits)

2. Optional extension:
   Radiation spreads to the next cell the robot moves into

Demo

→Robot_explorer.c

Live lecture code is written for teaching, not perfection.
It may include extra comments and may not always follow ideal coding style

UNSW College

Assignment 1 Style Tips:

Focus Area	What to Check
Functions	Use clear structure and meaningful function names
Constants	Use #define for magic numbers and characters (e.g. w, a, s, d)
Comments	Write clear and helpful comments to explain your code
Line length	Keep lines short and readable
Style checker	Run the style checker to identify issues
Style guide	Follow the style guide on the course website
Tutor feedback	Ask your tutor/lab demo for feedback

Voice of the Student

Anonymous ongoing feedback
Anything you wanted to share with me

26T1 Voice of the Student – Fill out form

See you soon …

UNSW College