SEMINAR 1 – Week 2

Contents

1. OBJECTIVES

- Solve a problem using modular programming in C.
- Discuss memory management in C and implement various data structures (static and dynamic).

2. PROBLEM STATEMENT

NASA's exoplanet exploration website (https://exoplanets.nasa.gov/) shows there are 5587 confirmed exoplanets so far and the Kepler space telescope is still "searching" for new ones. The habitable exoplanets are those that are closest in size to Earth and located within the habitable zone of a star, where the temperature is right for liquid water to exist on the surface.

Create an application to keep track of the planets that have been discovered so far. Each **Planet** has a name, a type (Neptune-like, gas giant, terrestrial, super-Earth, unknown) and a distance from Earth (in light years). The application will allow one to:

a. Add a planet. There can be no two planets with the same name.
b. Display all planets of a given type. If the type is empty, all planets will be displayed.
c. Display all planets within a given distance from Earth.
d. Undo and redo the last change.

3. MODULAR PROGRAMMING

- Separate the interface from the implementation
- Hide implementation details
- Header files and source code files
- Protecting against multiple inclusion (#ifndef, #define, #pragma once)

4. STATIC ALLOCATION

- There is no need for explicit memory allocation, this happens automatically, when variables are declared.
- All fields of the *Planet* structure are statically allocated.

```
typedef struct
{
	char name[50];
	char type[50];
	double distanceFromEarth;
} Planet;
```

- The vector of planets is statically allocated.

```
typedef struct
{
	Planet planets[100];
	int length;
} PlanetRepo;
```

- All objects in the application are statically allocated.

```
int main()
{
	PlanetRepo repo = createRepo();
	Service serv = createService (&repo);
	UI ui = createUI(&serv);
	// …

	return 0;
}
```

5. DYNAMIC ALLOCATION

- Memory is allocated when we need it.
- We are **responsible** with de-allocating it, once we no longer need it.
- Necessary functions: **malloc**, **free** (header *stdlib.h*).
- The objects we are working with will have to provide functions for *creation and destruction*.
- E.g. Creating and destroying a Planet:

```
Planet* createPlanet(char* name, char* type, double distanceFromEarth)
{
    Planet* p = (Planet*)malloc(sizeof(Planet));
    p->name = (char*)malloc(sizeof(char) * (strlen(name) + 1));
    strcpy(p->name, name);
    p->type = (char*)malloc(sizeof(char) * (strlen(type) + 1));
    strcpy(p->type, type);
    p->type = type;

    return p;
```

```
}

void destroyPlanet(Planet* p)
{
    // free the memory which was allocated for the component fields
    free(p->name);
    free(p->type);

    // free the memory which was allocated for the planet structure
    free(p);
}
```

- The vector of planets will contain pointers, not objects.

```
typedef struct
{
	Planet* planets[100];
	int length;
} PlanetRepo;
```

- All objects in the application are dynamically allocated. Then they must also be destroyed.

```
int main()
{
	PlanetRepo* repo = createRepo();
	Service* serv = createService(repo);
	UI* ui = createUI(serv);
	// …

	destroyUI(ui);
	return 0;
}
```