SEMINAR 2

Contents

Observation: All implementations are illustrative. In order for the programs to function correctly, more function implementations are necessary.

1. Objectives

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

2. Problem statement

NASA's exoplanet exploration website (https://exoplanets.nasa.gov/) shows there are 3705 confirmed exoplanets so far and the Kepler space telescope is still "searching" for new ones. Among these, there are 6 potential Earths – planets 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;
```

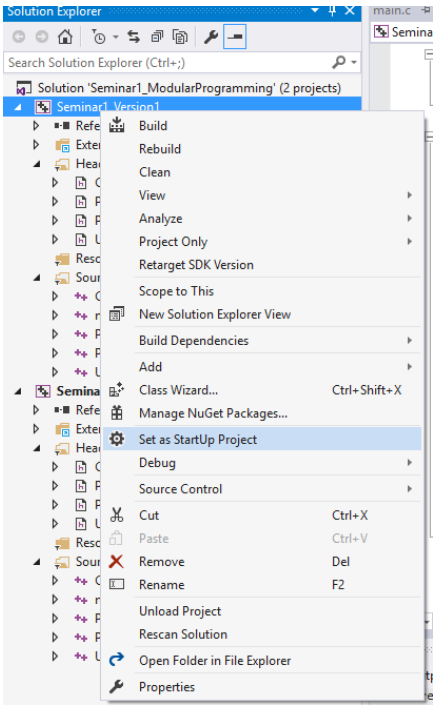
- All objects in the application are statically allocated.

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

	return 0;
}
```

Please see Seminar2_Week1_illustrative_implementation.zip → project Seminar1_Version1.

Obs.: In a Visual Studio solution, one can have several projects. If you want to run a certain project, right click on the project and choose “Set as StartUp Project”.



5. Dynamic allocation – Version 1

- 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;
}
```

Please see Seminar2_Week1_illustrative_implementation.zip → project Seminar1_Version2.

6. DYNAMIC ALLOCATION – VERSION 2

- Memory is allocated when we need it.
- We are **responsible** with de-allocating it, once we no longer need it.
- All objects that are responsible with other objects (store pointers to others) must destroy these as soon as they are no longer needed.
- We use a dynamic array of generic elements, with a maximum capacity and a size. The array's size may increase or decrease, depending on the number of elements that need to be stored.

```
typedef struct
{
    TElement* elems;
    int length;          // actual length of the array
    int capacity;        // maximum capacity of the array
} DynamicArray;
```

- In this version of the application, the dynamic array will store *pointers to objects* (of type Planet).
- For allocation and de-allocation, we have several options:
    - Objects are destroyed when they are no longer used. E.g.: The repository will store pointers to objects created in the Service. Then in the function creating objects in the Service, the created object will not be destroyed and a pointer to this object is passed to the Repository. The repository does not need to copy the object, but only to store the pointer. This approach can be used for objects in the repository and Service, as well as in

the *main* function, where elements repository, Service and ui are created and they are passed as pointers to one another (no copied of these are made). In this case, the ui will destroy everything and in the *main* function, the *create* functions will not have an associated *destroy* function explicitly called.

```
PlanetRepo* repo = createRepo();
OperationsStack* operationsStack = createStack();
Service* serv = createService(repo, operationsStack);

// … …

UI* ui = createUI(serv);
startUI(ui);
destroyUI(ui);
```

- Destroy an object in the same function in which it was created. In such cases, if another object must work with it, we must make sure to make a copy of the initial object. E.g.: we create a Planet in the Service and we want to destroy it in the same function where it was created, in order to have a pair of allocation/de-allocation function calls. In this case, the repository will be responsible with making a copy of the object and with destroying the objects that it is responsible with. This is *Option 2* presented in the application. This approach requires more memory, as a lot of copies are being made.

```
int addPlanetServ(Service* c, char* name, char* type, double distanceFromEarth)
{
	Planet* p = createPlanet(name, type, distanceFromEarth);

	int res = addPlanet(c->repo, p);

	if (res == 1) // if the planet was successfully added - register the operation
	{
		Operation* o = createOperation(p, "add");
		push(c->undoStack, o);
		// once added, the operation can be destroyed (a copy of the operation was
added)
		destroyOperation(o);
	}

	// destroy the planet that was just created, as the repository stored a copy
	destroyPlanet(p);

	return res;
}
```

7. UNDO/REDO

- A stack data structure is created to store the operations that are being made within the application. This structure will be used for multiple undo. Another stack can be created for multiple redo.

```
typedef struct
{
	Planet* planet;
	char* operationType;
} Operation;


typedef struct
{
	Operation* operations[100];
	int length;
} OperationsStack;
```

Please see Seminar2_Week1_illustrative_implementation.zip → project Seminar1_Version3.

- Another approach for the undo/redo functionality is the "list of lists" approach. This refers to memorizing the current list of elements before each modification that influences the list (add/remove). An undo operation would mean replacing the current list with the list on the last position in the "list of lists".
- For this approach you can use the generic DynamicArray, which contains objects of type TElement, where TElement is defined as void*. Thus, to memorise the previous "lists", a dynamic array containing pointers to other dynamic arrays can be used. For the generic implementation of the DynamicArray, please see some details below.

8. Working with a Dynamic Array of generic elements

- Working with generic elements in C is achieved via a **void pointer**.
- A void pointer can hold an address of any type and can be casted to any type.
- To dereference a void pointer, one must first cast it to its associated pointer type.
- The dynamic array will contain void pointers. Thus, we will be able to create dynamic arrays of any types (pointers to planets, pointers to operations, pointers to dynamic arrays).
- The dynamic array no longer uses a specific pointer type. The question is now: how will the elements stored in the array be destroyed/copied?
- This will be achieved using some generic functions (function pointers) stored in the dynamic array.
- Then, according to the type of elements the dynamic array contains, the function pointers will point to the correct associated functions.
- In the example below, the type `DestroyElementFunctionType` represents the function pointer that deals with the deallocation of elements contained in the dynamic array.

```
typedef void* TElement;
typedef void (*DestroyElementFunctionType)(void*);

typedef struct
{
```

```
    TElement* elems;
    int length;                // actual length of the array
    int capacity;        // maximum capacity of the array
    DestroyElementFunctionType destroyElemFct; // function pointer to the function
responsible with deallocating the elements stored in the array

} DynamicArray;

/// <summary>
/// Creates a dynamic array of generic elements, with a given capacity.
/// </summary>
/// <param name="capacity">Integer, maximum capacity for the dynamic array.</param>
/// <param name="destroyElemFct">Function pointer, of type DestroyElementFunctionType.
The function that deals with deallocating the elements of the array.</param>
/// <returns>A pointer the the created dynamic array.</returns>
DynamicArray* createDynamicArray(int capacity, DestroyElementFunctionType
destroyElemFct);
```

- The function pointer will point to the correct deallocation function, according to the stored elements. A dynamic array containing pointers to planets will be created as follows:

```
DynamicArray* arrayOfPlanets = createDynamicArray(2, &destroyPlanet);
```

Please see Seminar2_Week1_illustrative_implementation.zip → project Seminar1_Version4.