# SEMINAR 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

The **Stargate Program** needs an application to help keeping track of the planets and alien races that have been discovered so far. Each **Planet** has a unique symbol composed of exactly 7 symbols, a name, the Solar System it belongs to and the distance to Earth (measured in thousands light-years). The Stargate team needs this application to help them in the following ways:

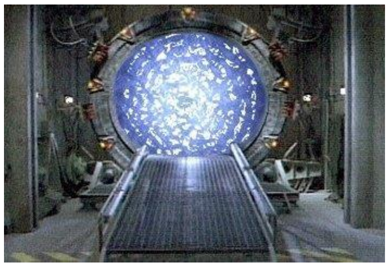


*Image source: https://www.pinterest.com/natbackstrom/sg-1/*

a. The application must allow adding and deleting planets.
b. The application should offer the possibility to display all the planets whose symbols contain a given combination as a substring (if the combination is empty, all the planets should be considered).
c. The application should allow displaying all the planets in a given Solar System (if the Solar System is empty, all planets should be considered), whose distances to Earth are less than a given value, sorted ascending by distance.
d. The application must provide the option to undo and redo the last change.

## 3. 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 symbols[8];
	char name[50];
	char solarSystem[50];
	double distanceToEarth;
} Planet;
```

- The vector of planets is statically allocated.

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

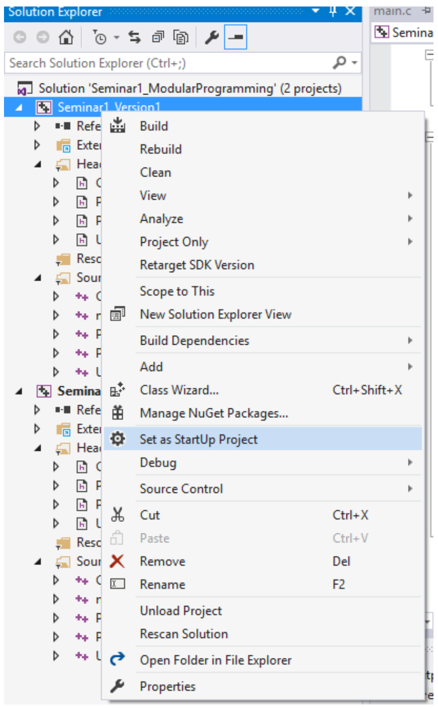
- All objects in the application are statically allocated.

```
int main()
{
	PlanetRepo repo = createRepo();
	Controller ctrl = createController(&repo);
	UI ui = createUI(&ctrl);
	// …

	return 0;
}
```

**Please see Seminar1_ModularProgramming.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”.



## 4. 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* symbols, char* name, char* solarSystem, double
distanceToEarth)
{
	Planet* p = (Planet*)malloc(sizeof(Planet));
	p->symbols = (char*)malloc(strlen(symbols) + 1);
	strcpy(p->symbols, symbols);
	p->name = (char*)malloc(strlen(name) + 1);
	strcpy(p->name, name);
	p->solarSystem = (char*)malloc(strlen(solarSystem) + 1);
	strcpy(p->solarSystem, solarSystem);
	p->distanceToEarth = distanceToEarth;

	return p;
}

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

	// 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();
        Controller* ctrl = createController(repo);
        UI* ui = createUI(ctrl);
        // …

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

**Please see Seminar1_ModularProgramming.zip → project Seminar1_Version2.**

## 5. 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:
    - 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 Controller 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. At the moment, the application uses this approach for objects in the repository and in the undo stack.

```
int addPlanetCtrl(Controller* c, char* symbols, char* name, char* solarSystem,
double distanceToEarth)
{
	Planet* p = createPlanet(symbols, name, solarSystem, distanceToEarth);

	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;
}
```

- Objects are destroyed when they are no longer used. E.g.: The repository will store pointers to objects created in the Controller. Then in the function creating objects in the Controller, 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 is **not** used for objects in the repository and controller, however it is used in the *main* function, where elements repository, controller 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();
Controller* ctrl = createController(repo, operationsStack);

// … …

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

- 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 Seminar1_ModularProgramming.zip → project Seminar1_Version3.**