B4 - Object-Oriented Programming

B-OOP-400

OOP in C

Simulate OOP

1.0

{EPITECH.}

OOP in C

language: C
compilation: via Makefile, including re, clean and fclean rules

- Your repository must contain the totality of your source files, but no useless files (binary, temp files, obj files,...).
- All the bonus files (including a potential specific Makefile) should be in a directory named *bonus*.
- Error messages have to be written on the error output, and the program should then exit with the 84 error code (0 if there is no error).

"OOP" stands for **Object-Oriented-Programming**.
It's a programming paradigm based on the concept of **objects**, which can contain **data**, in the form of **fields** (often known as attributes or properties), and **code**, in the form of **procedures** (often know as methods). An object in OOP is a symbolic and autonomous container that contains informations and mechanisms in relation to the subject. (A car, a button, a dog, a sound, everything that interacts with something else …)
The goal of this workshop is to let you discover how to do OOP in C, which isn't the goto language to do OOP.

Topics :

- Union and enums :
  It will ease the implementation of new objects in the game without touching at the calling sequence.
  We'll also cover "structure padding".
- Function pointers :
  It will allow us to imitate callbacks from higher level languages.

As the best way to learn about these topics is to practice, you'll find exercices at the end of this subject.

Unions and enums

An union is a special data type that allows you to store different data types in the same memory location. You can define a union with many members, but only one can be set at given time.
Unions provide an efficient way of using the same memory location for multiple purposes. It can look like that :
You should add an object_type_t to every object you create to help identify it.

```
typedef union game_object
{
    enemy_t *enemy;
    player_t *player;
} game_object_t;
```

If you had objects in an array; and wanted to access to specific common field, you would do:

```
game_object_array[index]->enemy->common_field_x = 0;
```

You can now access to every common field in the array without knowing about what type of data you're dealing with. Because every object have is own field of this type at the same location .

The padding of a structure is the alignement of data in memory. In a C structure, fields have a constant padding, that means that when you use a union, and when you access "union->enemy->the_common_field" you will try to acces everything at the memory location of enemy->the_common_field.
Even if it's something else. (That can be bad.)

You should therefore split your objects in two parts, a struct that contains the common fields (types and update/destroy function) and the specific fields next to it (life, size, sprite, ect).

If you have the same common fields for every object you create, there will be no memory padding problems. You can then call any specific functions by checking an object type.

Enums allows you to set constants linked to members of enums, in the following example, enemy_type is equal to zero and player_type is equal to one.

```
typedef enum object_type
{
    enemy_type,
    player_type
} object_type_t;

object_type_t type_of_the_object = enemy_type;

if (game_object_array[index]->enemy->type == player_type)
    game_object_array[index]->player->hairstyle = swap_hairstyle();
```

FUNCTION POINTERS

Let's make a basic function pointer (fonction pointer funct_ptr).
First of all, you'll have to create a struct prototype that contains a funct_ptr, it will be composed of 3 elements.

```
void *(*funct_ptr_name)(param_type param);
```

Here for an example update function

```
int (*update)(game_object_t *object);
```

To initialize a funct_ptr: funct_ptr_name = &function_name;
(Note that the **&** isn't necessary but advised.)

Do not call it like that :

```
callback_return = update(object);
```

But like that :

```
callback_return = object->update(object);
```

Because the function update() is not the same for every object.
(Note that we send the adcress of the object to itself so that it can change its own contents.)

"Ok, but what's the point of using a pointer and not directly the real function ?"
Reusability ! You can create multiples objects with the same callbacks and they will behave the same way. for instance multiple button objects, multiples enemies, …

```
typedef struct enemy enemy_t;
struct enemy
{
    object_type_t type;
    void(*update)(enemy_t *enemy, object_list_t *list);
    void(*render)(render_window *window, enemy_t *enemy);
    void(*destroy)(enemy_t *enemy); //free/destroy for program sanity's
    //you can add other propriety : life, hairstyle, or anything that "is" your "
        enemy"**
};
```

Here is an idea of an enemy structure from a graphical project. It shares its type and callbacks attributes with all our other structures.

Forced Labour

Exercice 1: Create a “game_object_t” union of simple structures, and a type enumeration associated to it.

Exercice 2: Write a function to create a game_object_t, and one to destroy it (free all allocated memory inside it, if any)

Exercice 3: Create a functions that adds an object to an array (or a list)

Exercice 4: Add a display function to your structures that displays its contents

Exercice 5: Add an update function to your structure that changes its contents when its called

Exercice 6: insert those function calls in a loops

Conclusion and tips

Now you can imitate OOP in C.
You may use linked lists in your projects, it's far more flexible than arrays to add and remove quickly : particles, enemies, …
With this program architecture, you can easily add new object types to your program.

Last tip, it's common that games are split in scenes.
To handle that you can do with multiple lists that represent your scenes :

```
object_list_t **init_scenes(void)
{
    object_list_t **scenes_list = malloc(sizeof(object_list_t *) * NB_SCENES);

    scenes_list[menu_scene_idx] = init_menu_scene();
    scenes_list[game_scene_idx] = init_game_scene();
    scenes_list[fight_scene_idx] = init_fight_scene();
    return (scenes_list);
}

int game(void)
{
    object_list_t **scenes = init_scenes();
    status_t *prog_stat = init_program_status(); //that contains a scene_index, a
        window, a clock and the running status

    while (prog_stat->running) {
        analyze_events(prog_stat, scene[prog_stat->scene_index]);
        update(prog_stat, scene[prog_stat->scene_index]);
        render(prog_stat, scene[prog_stat->scene_index]);
    }
    return (destroy(prog_stat, scenes) == SUCCESS ? SUCCESS : ERROR);
}
```

With an array of object_list_t * you can easily swap between the scenes, by modifying the prog_stat->scene_index. **They are totally independent.** By default. All the objects of all scenes are handled by this simple loop.

To go further:
- If you used arrays for your game objects until now, consider using linked lists !
- Try using objects for a projects, graphical projects are a pretty common use of OOP

By Carcenac Sautron Izaac and Dommel-Prioux Iona and Anne Richard