SKL

C++ Pool - Rush 1

Koalab koala@epitech.eu

Abstract: Subject of the first rush of your C++ pool / Standard Kreog Library

Contents

Chapter I

Introduction

Read carefully the introduction, each instruction is very important. This rush requires a lot of rigor.

I.1 Inspiration

This rush is inspirated by the work of Axel-Tobias Schreiner, and mainly by his book *Objekt-orientierte Programmierung mit ANSI-C* (available at the following address: `http://www.cs.rit.edu/~ats/books/ooc.pdf`). Of course, we encourage you to take a look at it.

In the preface, the author tells us about his amusement while discovering that the C language is an oriented-object language in its own right. Obviously, it has stirred up our curiosity and the result is this rush. Of course we hope you'll have as fun as the author!

I.2 Unfolding of the rush

The rush is divided into three main parts.

The part 0 (said reading) will introduce you some concepts that you'll need to master before keep going the rush. This part will also be evaluated during the defense, do not neglect it!

The part 1 (said designing) is mandatory, and the exercises are linear (ie. you must do them following the given order). In this first part, we'll present you an "object" design in C language. This design does not claim to be perfect, but in this part, we ask you to understand it and to implement it.

The part 2 (said implementation) is the apogee of this rush. Using the object implementation which has been introduced to you in the first part, we'll ask you to implement a certain number of basic types, containers and object oriented (and generic) algorythms. In this part, all exercises are independant.

I.3 Defense

During the defense, you'll be evaluated on two main points:

- The progression of your rush. When you're done with the part 1, you can unleash your imagination in the part 2.
- The quality of your thinking. It will be the most important point for us. The goal of this rush is to drive you to think about several new concepts for you, and which will be precised to you during the next two weeks of your pool. Document you, ask questions to yourselves, and find a way to answer it!

I.4 Turn-in

We ask you to strictly respect the names of every folders in which you'll turn-in each exercise: ex_01, ex_02, ex_03, ex_04, ex_05, ex_06 and bonus. These folders must be located at the root folder of your turn-in directory. None of these folders must contain a sub-folder. If your folder doesn't have the good name or isn't located at the right place, you won't be corrected! No derogation will be given, you are warned.

Your repository will be closed in writting on sunday morning at 10am. Only the files present on the repository during the defense will be corrected. No derogation will be given.

If you have any question, we invite you to send a message by email to the authors of the subject.

I advise you to read this part once again.

⚠ `Your files will be compiled with the flags -std=gnu99 -Wall -Wextra -Werror`

Chapter II

Part 0 : Some reading

In this part, we'll introduce you some concepts of the oriented-object programming.

`Google and Wikipedia are your friends!`

II.1 Encapsulation

The problem when creating a type in C language is that the user is totally free in the manner of using it. Your only way to protect the user is to give him a set of functions to use this type. For example, if you define a `player_t` structure, you will have to provide the functions `player_init(player_t*)`, `player_destroy(player_t*)` and so on.

So, for each different type, you'll have to systematicaly code all the functions which handle the *objects* of your program. For example, for the type `player_t`, you'll prefix all the functions by `player_`. The way to hide to the user the representation in memory of a type, to order every function which apply to it together is called the *encapsulation*.

This way of protecting guarantees to the user a total security while he only handles these objects with the provided functions.

II.2 Types and objects

When you declare a type in C language, you must, to use it, systematically do two things. First of all, allocate a memory area for your instance, and then to initialize it. We call this process the *construction* of an *object* or *instance*. The first step is done exactly in the same way for every type, if only we know the size of this type. Only the second step, said initialization, is typical for each type. We call the function that initializes an object a *constructor*, and the one that destroys it a *destructor*.

The set of a type and all its member functions (who act on this type) is called a *class*. We'll describe through this rush one of some possible way to do *oriented object programming* in C language.

II.3 C language implementation

Whatever the author mentioned above may think, one has to admit that C language is not a fundamentally object oriented language. How can we manage that? In this part, we'll introduce you the main ideas underlying to the conception we'll propose you. That's your job to design all the missing pieces of the part 1!

II.3.1 *object.h*

object.h contains the `Class` type. This is therefore the type of a type (inception), it contains the name of the class, its size (for the malloc), and function pointers, like the constructor and the destructor.

II.3.2 *raise.h*

raise.h contains the `raise()` macro that you *must* use to handle all error cases (for example, if `malloc()` returns `NULL`). It takes as parameter a chain of caracters that indicates the error type. For example:

```c
if ((ptr = malloc(sizeof(*ptr))) == NULL)
    raise("Out of memory");
```

And has for effect to display this message on the error output and exit the program.

II.3.3 *point.h*

point.h presents the `Point` *class*. This description is intriguing, but *point.c* should enlighten you. The `Point` *class* is actually the external declaration of a static variable that describes the *class*. You can now stop reading for a while, and meditate this sentence using the provided code.

Actually, as `Point` is a `Class` type variable, it describes a type. We'll use this type in order to build and destroy point *instances*.

II.3.4 *point.c*

This file presents the *implementation* of the `Point` class. Ultimately, this is here that the constructor and the destructor will be coded. You'll also find the much talked-about `Point` variable, which contains as expected the function pointers on constructos and destructors, which will be used in a transparent manner by `new` and `delete`.

II.3.5 *new.h*

Finally, *new.h* contains the prototypes of `new()` and `delete()` functions, which allows you to build and destroy objects. Your turn now!

Chapter III

Part 1: Simple objects

III.1 ex_01: Creation / destruction of objects

Provided files:

Name	Description
`raise.h`	Contains the `raise` macro
`new.h`	Prototypes for `new` and `delete`
`object.h`	Definition of the `Class` structure
`point.h`	Declaration of the `Point` variable
`point.c`	Implementation of the `Point` class
`ex_01.c`	`main` example

Files to turn-in:

Name	Description
`new.c`	Implementation of `new` and `delete` functions

The purpose of this exercise is the implementation of `new()` and `delete()` functions. These will have for effect to build and to destroy `Point` type *objects*. We want to be able to write a code like this one:

```c
#include "new.h"
#include "point.h"

int main()
{
    Object * point = new(Point);
    delete(point);

    return 0;
}
```

The task of this first version of `new()` is to allocate memory depending on the class passed as parameter, then to call the *constructor* of the *class*, if it is available. In the same way, the `delete()` function should call the destructor if it is present, then free the memory.

`malloc(3) memcpy(3)`

III.2 ex_02: Creation / destruction of objects, the return

Provided files:

Name	Description
raise.h	Contains the raise macro
new.h	Prototypes for new and delete
object.h	Definition of the Class structure
point.h	Declaration of the Point variable
vertex.h	Declaration of the Vertex variable
ex_02.c	main example

Files to turn-in:

Name	Description
new.c	Implementation of new and delete functions
point.c	Implementation of the Point class
vertex.c	Implementation of the Vertex class

The previous version of the `new()` function didn't permit to pass parameters to the constructor. You therefore have to provide a new version of this one using the `va_list`. In addition, to be able to answer to other needs, a version of `new()` named `va_new()` is asked. Prototypes of asked functions are therefore:

```c
Object* new(Class* class, ...);
Object* va_new(Class* class, va_list* ap);
void delete(Object* ptr);
```

Constructors and destructors will no longer display messages, for this exercise and for the next one.

Thus, it will be possible to use the functions in this way:

```c
#include "new.h"
#include "point.h"
#include "vertex.h"

int main()
{
    Object* point = new(Point, 42, -42);
    Object* vertex = new(Vertex, 0, 1, 2);

    delete(point);
    delete(vertex);
    return 0;
}
```

You have to create a new `Vertex` class taking inspiration from the `Point` class. The `Class` structure now contains a new *member function* `__str__`, which returns a chain of caracters. The `str` macro present in `object.h` manages to call this member function. You must ensure that the following source code:

```c
printf("point = %s\n", str(point));
printf("vertex = %s\n", str(vertex));
```

produces this display:

```
point = <Point (42, -42)>
vertex = <Vertex (0, 1, 2)>
```

`stdarg(3) snprintf(3)`

III.3 ex_03: Add / subtract objects

Provided files:

Name	Description
`raise.h`	Contains the `raise` macro
`new.h`	Prototypes for `new` and `delete`
`object.h`	Definition of the `Class` structure
`point.h`	Declaration of the `Point` variable
`vertex.h`	Declaration of the `Vertex` variable
`ex_03.c`	`main` example

Files to turn-in:

Nom	Description
`new.c`	Implementation of `new` and `delete` functions
`point.c`	Implementation of the `Point` class
`vertex.c`	Implementation of the `Vertex` class

In this exercise, we'll simply add two member functions in the `Class` structure to be able to add or subtract objects. Therefore, you will have to adapt your previous files `point.c` and `vertex.c` so that they can implement the member functions `__add__` and `__sub__`. We could have, for example, a code like this one:

```c
#include "new.h"
#include "point.h"
#include "vertex.h"

int main()
{
    Object* p1 = new(Point, 12, 13),
           * p2 = new(Point, 2, 2),
           * v1 = new(Vertex, 1, 2, 3),
           * v2 = new(Vertex, 4, 5, 6);

    printf("%s + %s = %s\n", str(p1), str(p2), str(add(p1, p2)));
    printf("%s - %s = %s\n", str(p1), str(p2), str(sub(p1, p2)));
    printf("%s + %s = %s\n", str(v1), str(v2), str(add(v1, v2)));
    printf("%s - %s = %s\n", str(v1), str(v2), str(sub(v1, v2)));

    return 0;
}
```

And we'll have the following display:

```
<Point (12, 13)> + <Point (2, 2)> = <Point (14, 15)>
<Point (12, 13)> - <Point (2, 2)> = <Point (10, 11)>
<Vertex (1, 2, 3)> + <Vertex (4, 5, 6)> = <Vertex (5, 7, 9)>
<Vertex (1, 2, 3)> - <Vertex (4, 5, 6)> = <Vertex (-3, -3, -3)>
```

III.4 ex_04: Base types

Provided files:

Nom	Description
`raise.h`	Contains the `raise` macro
`bool.h`	Definition of the `bool` type and `true` and `false` macros
`new.h`	Prototypes for `new` and `delete`
`object.h`	Definition of the `Class` structure
`float.h`	Declaration of the `Float` variable
`int.h`	Declaration of the `Int` variable
`char.h`	Declaration of the `Char` variable
`ex_04.c`	`main` example

Files to turn-in:

Nom	Description
`new.c`	Implementation of `new` and `delete` functions
`float.c`	Implementation of the `Float` class
`int.c`	Implementation of the `Int` class
`char.c`	Implementation of the `Char` class

We'll extend one time again the base class and rewrite some native types of C language. The new types you'll have to implement are `Int`, `Float` and `Char`. It will be like previously, we have to be able to add and subtract objects of the same type, but also to be able to compare them. Comparison operators `==` (`__eq__`), `<` (`__lt__`) and `>` (`__gt__`) make their appearance in the base class. These three classes will be intensively used in the next section. Operations and comparisons between objects of different types is not asked but could be a bonus :)
For example, an object of type `Int`, `Float` or `Char` could be manipulated like that:

```c
void compareAndDivide(Object* a, Object* b)
{
    if (gt(a, b))
        printf("a > b\n");
    else if (lt(a, b))
        printf("a < b\n");
    else
        printf("a == b\n");

    printf("b / a = %s\n", str(div(b, a)));
}
```

As usual, macros are defined to facilitate the call of member functions.

Chapter IV

Part 2: Generic containers

In the previous part, we have created simple types. This section is going to focus onto the writting of containers. A container is an object capable to contain any type of object deriving from the base class. We'll add here an *intermediate class*, in order to add only member functions to containers. Indeed, it will be too heavy to add all member functions for every classes, that's what we've done until now. An intermediate class is simply a structure containing a variable of type `Class` and is defined thus:

```c
#include "object.h"

typedef struct Container_s Container;

struct Container_s
{
    Class base;
    void (*newMethod)(Container* self);
};
```

We have simply added a function pointer named `newMethod` into the class `Container`. You have to notice that this class *contains* the base class, it must therefore implement all its member functions.

To define a container, you can proceed in this way in the `.c` file:

```c

#include "container.h"

typedef struct MyContainerClass
{
    Container base;
    int _val;
};

static MyContainerClass _descr = {
    { /* Container struct */
        { /* Class struct */
            sizeof(MyContainerClass),
            "MyContainer",
            /* All Class functions here */
        },
        &MyContainer_newMethod, /* the new method from class Container */
    },
    0, /* members of MyContainer */
};

Class* MyContainer = (Class*) &_descr; /* as usual */
```

We have defined here a class `MyContainer`, *derived* from `Container`, derived itself from `Class`. This is why the `_descr` variable includes the declaration of three imbricated structures.

The last concept introduced in this section is the concept of *iterators*. An iterator allows to run through a container, like an index in a table. Earlier, when you ran through a C language table, you were doing something like that:

```c
int tab[10];
unsigned int i;

i = 0;
while (i < 10)
{
    /* do stuff */
    i = i + 1;
}
```

The issue was, when you decided to change the container, for example switching to lists, you had to change all the code, because the way of going through a list is totally different. A standardized way to go through a container is to use iterators. Assuming the presence of a class `MyContainer`, we'll have:

```c
Container* tab = new(MyContainer);
Iterator* it;
Iterator* tab_end;

it = begin(tab);
tab_end = end(tab);
while (lt(it, tab_end)) /* it < tab_end */
```

```
{
    /* do stuff */
    incr(it);
}
```

You can therefore notice that the philosophy still the same, but the implementation of the container is completely hidden. The member function `incr` is exactly identical to the incrementation of the variable `i` in the previous example: it allows us to *progress* to the next element in the table. The definition of the `Iterator` class follows the same reasoning than for the `Container`. We define it as an intermediate class, serving to define the other containers, specific to each container.

IV.1 ex_05 : Array class

Provided files:

Nom	Description
raise.h	Contains the raise macro
bool.h	Definition of the bool type and true and false macros
new.h	Prototypes for new and delete
object.h	Definition of the Class structure
container.h	Definition of the Container structure
iterator.h	Definition of the Iterator structure
float.h	Declaration of the Float variable
int.h	Declaration of the Int variable
char.h	Declaration of the Char variable
array.h	Declaration of the Array variable
array.c	Partial implementation of the Array class
ex_05.c	main example

Files to turn-in:

Nom	Description
new.c	Implementation of new and delete functions
float.c	Implementation of the Float class
int.c	Implementation of the Int class
char.c	Implementation of the Char class
array.c	Implementation of the Array class

In this exercise, we give you the partial implementation of the Array class, simulating the operation of a standard table. You therefore have to fill the functions contained in the file array.c in order to obtain the following behavior:

The constructor of an Array takes respectively as argument the size (size_t), the type contained in the table (Class*) and the arguments for the constructor of this type. We'll have for example:

```
Object* tab = new(Array, 10, Float, 0.0f);
```

Here, tab is a table of 10 Float initialized at the value 0.0f.

The member function getitem takes an index (size_t) as argument and returns an object (Object*).

The member function setitem takes an index (size_t) as argument and all arguments to build an instance of the contained type. For example:

```
setitem(tab, 3, 42.042f);
```

We put in tab a Float whose value is 42.042f at the index 3.

The member function `setval` of the table iterator works the same way:

```
Object *start = begin(tab);
setval(start, 3.14159265f);
```

We put the value 3.14159265f into the first square of the table.

`make use of va_new()`

IV.2 ex_06: List class

Provided files:

Nom	Description
`raise.h`	Contains the `raise` macroc
`bool.h`	Definition of the `bool` type and `true` and `false` macros
`new.h`	Prototypes for `new` and `delete`
`object.h`	Definition of the `Class` structure
`container.h`	Definition of the `Container` structure
`iterator.h`	Definition of the `Iterator` structure
`float.h`	Declaration of the `Float` variable
`int.h`	Declaration of the `Int` variable
`char.h`	Declaration of the `Char` variable

Files to turn-in:

Nom	Description
`new.c`	Implementation of `new` and `delete` functions
`float.c`	Implementation of the `Float` class
`int.c`	Implementation of the `Int` class
`char.c`	Implementation of the `Char` class
`list.h`	Declaration of the `List` variable
`list.c`	Implementation of the `List` class
`ex_06.c`	`main` example

Taking example on the `Array` class, you'll create a `List` class, allowing to easily manipulate lists. You can choose the behaviors you want for the operators (for example add), but you must justify your choices during the defense (is it relevant to be able to multiply two lists?). You are free to add methods if you use a list-specific intermediate class. We'll have to be able to use your lists and the `Array` class together.

A set of tests must be present into your turned-in main showing the good working of your class. These tests must be numerous and of quality, and will also be evaluated during the defense.

In this part, it is permitted to modify *container.h*.

IV.3 ex_07: Bonus!

Feel free to impress us by:

- Coding additional containers (`String` again?)
- Adding intermediate classes for `Array` and for `List`, and which define specific member functions to one and the other of these classes:
  - Array.resize
  - Array.push_back
  - List.push_front and List.push_back
  - List.pop_front and List.pop_back
  - List.front and List.back
- Making the previous containers compatible with C language native types (int, float, char).
- Making the use safer (no more macros? magic number at the beginning of a class? verify the types?)
- Modifying the design to bear the definition of member functions into the intermediate classes.
- Making possible to make operations between different types:
  - `3.0f + 2 = 5.0f`
  - `3 * ['a'] = ['a', 'a', 'a']`
  - `2 * "hello" = "hellohello"`
  - etc.
- Look totally fresh after this rush ;)