B3- C++ Pool

B-PAV-242

SKL

C++ Pool - Rush 1

v1.0

SKL

C++ Pool - Rush 1

repository name:	Check the Intranet
repository rights:	ramassage-tek
language:	C
group size:	Check the Intranet

- 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).

Introduction

And Inspiration

Read the introduction carefully, each instruction is very important.

0.1 Inspiration

This rush is inspired 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 much fun as the author!

0.2 Unfolding of the rush

The rush is divided into three main parts.

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

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

Part 2 (implementation) is the main point of this rush. Using the object implementation introduced in the first part, we'll ask you to implement a certain number of basic types, containers and object oriented (and generic) algorithms. In this part, all exercises are independant.

0.3 Defense

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

- The progression of your rush. When you're done with part 1, you can unleash your imagination in part 2.
- The quality of your thinking. It will be the most important point for us. The goal of this rush is to get you to think about several new concepts, which will be more clearly explained during the next two weeks of your pool. Read documentation, question your thoughts, and find a way to answer them!

0.4 Turn-in

We ask you to strictly respect the names of every folder in which you'll turn-in your 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 right name or isn't located at the right place, you won't be corrected! No derogation will be given, you've been warned.

Your repository will be closed for writing 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 suggest you email the authors of the subject.

I advise you to read this part once again.

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

Part 0

Some reading

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

Google and Wikipedia are your best friends!

0.5 Encapsulation

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

So, for each different type, you'll have to systematically code all the functions which handle the **objects** of your program. For example, for the **player_t** type, you'll prefix all the functions by **player_**. Hiding a type representation from the user and letting them use that type through functions is called **encapsulation**.

This technique guarantees to the user a total security as long as they only handle these objects with the provided functions.

0.6 Types and objects

When you declare a type in C, you must systematically do two things. First of all, allocate a memory area for your instance, and then initialize it. We call this process the **construction** of an **object** or **instanciation**. The first step is exactly the same for every type, as long as the size of the type is known. Only the second step, called initialization, is different 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 (which act on this type) is called a **class**. During this rush, we'll describe one of various possible ways to do **object oriented programming** in C.

0.7 C language implementation

Whatever the author mentioned above may think, one has to admit that C is not a fundamentally object oriented language. In this part, we'll introduce you to the main ideas underlying the conception we're offering. Your job is to design all the missing pieces of part 1!

0.8 object.h

object.h contains the **Class** type. This is the type of a type (inception). It contains the name of the class, its size (for use with malloc), and function pointers, like the constructor and the destructor.

0.9 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 a string that indicates the error type as parameter. It has for effect to display this message on the error output and exit the program.

0.10 point.h

point.h contains 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 on 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**.

0.11 point.c

This file contains the **implementation** of the **Point** class. Ultimately, the constructor and destructor will be coded here. You'll also find the infamous **Point** variable, which contains, as expected, the function pointers for constructors and destructors, which will be used in a transparent manner by **new** and **delete**.

0.12 new.h

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

Part 1

Simple objects

- 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 build and 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, and to then 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)

- ex_02: Creation / destruction of objects, the come back

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 let us pass parameters to the constructor. You therefore have to provide a new version of **new()** using a **va_list**. In addition, in order to fullfill other needs, a version of **new()** named **va_new()** is required. The prototypes of these functions are therefore:

```
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 this way:

```
#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** called **__str__**, which returns a string. The **str** macro located in **object.h** calls this member function.

You must ensure that the following source code:

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

produces this output:

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

stdarg(3) snprintf(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 to the **Class** structure to be able to add or subtract objects. Therefore, you will have to adapt your previous **point.c** and **vertex.c** files so that they can implement the **__add__** and **__sub__** member functions.
We could have, for example, a code like this one:

```
#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 output:

```
<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)>
```

- 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 once again extend the base class and rewrite some native C types. The new types you'll have to implement are **Int**, **Float** and **Char**. Once again, it must be possible to add and subtract objects of the same type, but also to be able to compare them. Comparison operators **==** (**__eq__**), **<** (**__lt__**) and **>** (**__gt__**) should 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 required but could be a bonus :)
For example, an object of type **Int**, **Float** or **Char** could be used like this:

```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 should be defined to facilitate calling member functions.

Part 2

Generic containers

In the previous part, we created simple types. This section is going to focus on creating containers. A container is an object capable of containing any other type of object deriving from the base class. We'll add an **intermediate class**, in order to only add member functions to containers. Indeed, it would be too cumbersome to add all member functions to each class, like we have until now. An intermediate class is simply a structure containing a **Class** variable and is defined thusly:

```
#include "object.h"

typedef struct Container_s Container;

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

We have added a function pointer named **newMethod** into the **Container** class. Notice how this class **contains** the base class: it must therefore implement all of its member functions.
To define a container, you can proceed this way in the **.c** file:

```
#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 */
```

Here, we have defined a **MyContainer** class, **derived** from **Container**, itself derived 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 lets us walk through a container, like an index in an array. Up until now, when you ran through a C array, you did something like this:

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

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

The issue was that when you decided to change the container, for example switching to lists, you had to change all the iterating code, because the technique for walking 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:

```
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);
}
```

Notice that the philosophy is still the same, but the implementation of the container is completely hidden. The **incr** member function is identical to the incrementation of the **i** variable in the previous example: it lets us **progress** to the next element in the table. The definition of the **Iterator** class follows the same reasoning as for the **Container**. We define it as an intermediate class, used to define the other containers.

- 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 a partial implementation of the **Array** class, simulating the behavior of a standard table. You therefore have to fill in the functions contained in the file **array.c** in order to obtain the following behavior:

The constructor of an **Array** takes as arguments the size of the table (**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** objects, initialized with the 0.0f value.

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

The setitem member function 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 a **Float** with the 42.042f value at the index 3 in **tab**.

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

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

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

make use of va_new()

- 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 inspiration from the **Array** class, you'll create a **List** class, letting us easily manipulate lists. You can choose the behaviors you want for the operators (for example add), but you must defend 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 in your turned-in main showing the working behavior of your class. These tests must be numerous and of quality, and will also be evaluated during the defense.

In this part, you are allowed to modify **container.h**.

- ex_07: Bonus!

Feel free to impress us by:

- Coding additional containers (**String** again?)
- Adding intermediate classes for **Array** and for **List**, 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 it possible to perform operations between different types:
  - **3.0f + 2 = 5.0f**
  - **3 * ['a'] = ['a', 'a', 'a']**
  - **2 * "hello" = "hellohello"**
  - etc.
- Look totally fresh after this rush ;)