



# B3 - C++ Pool

---

B-PAV-242

## Rush 1

---

SKL



**KOALA**

42.0



# Rush 1

binary name: no binary  
group size: 1  
repository name: cpp\_rush1  
repository rights: ramassage-tek  
language: C



- Your repository must contain the totality of your source files, but no useless files (binary, temp files, obj files,...).



## INTRODUCTION

---

Please read this introduction carefully, as every point is very important.

### + INSPIRATION

---

This rush is inspired by the work of **Axel-Tobias Schreiner**, and more specifically his **Objekt-orientierte Programmierung mit ANSI-C**. Obviously, you are more than encouraged to take a look at it.

In the preface of the book, the author talks about his amusement when discovering that the C language can be an **object-oriented language** in its own right. Of course this made us curious, leading to the dawn of this rush. Here's hoping you have as much fun as the author!

### + UNFOLDING OF THE RUSH

---

The rush is split into three main parts:

#### 1.2.1 Part 0 - Reading

This part will introduce some concepts you'll need to master before you go any further. This part will also be evaluated during the defense, so do not neglect it!

#### 1.2.2 Part 1 - Design

This part will introduce an "object" design for the C language. This design does not claim to be perfect, but you will be required to understand and implement it.

#### 1.2.3 Part 2 - Implementation

Using the object implementation introduced in part 1, you will implement a number of basic types, containers and object-oriented (and generic) algorithms. All exercises in this part are independent.

### + DEFENSE

---

Two main points will be evaluated during the defense:

- The progress made during the rush
- The quality of your thinking process. This will be the most important part to us. The goal of the rush is to get you thinking of several concepts that are new to you, which you'll come to dive into more deeply in the coming weeks



## + TURN-IN

You **MUST** turn-in your work in a separate directory for each exercise: `ex_01`, `ex_02`, `ex_03`, `ex_04`, `ex_05`, `ex_06` and `bonus`. These directories must be located at the root of your turn-in directory. None of these folders must contain a sub-folder.



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



## PART 0 - SOME READING

---

In this part, we will introduce you to some basic concepts of **Object-Oriented Programming**.



Google and Wikipedia are your friends!

### + ENCAPSULATION

---

The problem when creating a type in C is that its usage is completely up to the user, they can do anything they want with it. The only way to protect the user is to give them a set of functions to use the type. For example, if you define a `player_t` structure, you will have to provide `player_init(player_t *)` and `player_destroy(player_t *)` functions.

This means that for each different type, you'll systematically be coding all the functions which handle the **objects** of your program. For example, for the `player_t` type, you'll prefix all the functions with "`player_`". Finding a way to hide the memory representation of a type from the user and group together all the functions that deal with that type is called **encapsulation**.

This protection guarantees total security to the user, as long as they only use the provided functions to modify the objects.

### + TYPES AND OBJECTS

---

When declaring a type in C, you must systematically do two things. The first is to allocate a memory area for the instance and initialize it. This process is called the **construction** of an **object** or **instance**.

The first step, allocating a memory area, is the same for every type, as long as its size is known.

The second step, initialization, differs for each type. A function, called a **constructor**, must be called to initialize the object. The function used to destroy an object is called a **destructor**.

The set of a type and its member functions (the functions that act on that type) is called a **class**. Throughout this rush, we'll explore one of several ways to do **Object-Oriented Programming** in C.



## + C IMPLEMENTATION

---

Whatever the author above may think, one has to admit that the C language is not fundamentally object-oriented. How can we deal with that? In this part, we'll introduce you to the core concepts underlying the design we're suggesting. Your job is to design all the missing pieces in part II!

### 2.3.1 object.h

The `object.h` file contains the `Class` type. This is the **type of a type**. A `Class` object contains the class' name, its size (used when allocating the memory), and function pointers such as the constructor and destructor.

### 2.3.2 raise.h

The `raise.h` file contains the `raise()` macro, which you **MUST** use to handle any error cases (for instance, if `malloc` returns a null pointer). It takes as a single parameter a string indicating the error type. For instance:

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

All it does is print out the message on the error output and exit the program.

### 2.3.3 point.h

The `point.h` file contains the `Point` **class**. This description is a bit intriguing, but `point.c` should enlighten you. The `Point` **class** is simply the external declaration of a static variable that describes the class. You may want to stop reading for a second and read that sentence again while taking a long look at the code.

`Point` is actually a `Class` object, as it describes a type. This type will be used in order to build and destroy point **instances**.

### 2.3.4 point.c

The `point.c` file contains the **implementation** of the `Point` class. This is where the constructor and destructor will be coded. You'll also find the `Point` variable we've been talking about so much. As expected, it contains function pointers for the constructor and destructor which will be used in a transparent manner by `new` and `delete`.

### 2.3.5 new.h

The `new.h` file contains the prototypes for the `new()` and `delete()` functions which let you build and destroy objects.



## PART 1 - SIMPLE OBJECTS

### + EX\_01 - OBJECT CONSTRUCTION / DESTRUCTION

Provided files:

Name	Description
raise.h	raise macro
new.h	new and delete prototypes
object.h	Class structure definition
point.h	Point variable declaration
point.c	Point class implementation
ex_01.c	Sample main function

Files to turn-in:

Name	Description
new.c	new and delete implementations

The purpose of this exercise is to implement the `new()` and `delete()` functions. These will build and destroy `Point` objects. We want to be able to write code such as the following:

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

int main()
{
    Object *point = new(Point);
    delete(point);
    return 0;
}
```

This first draft of `new()` simply has to allocate memory according to the class passed as parameter, and then call the class' **constructor** if there is one. Similarly, `delete()` must call the **destructor** if there is one, and then free the memory.



`malloc(3)` `memcpy(3)`



## + EX\_02 - OBJECT CONSTRUCTION / DESTRUCTION, 2.0

Provided files:

Name	Description
raise.h	raise macro
new.h	new and delete prototypes
object.h	Class structure definition
point.h	Point variable declaration
vertex.h	Vertex variable declaration
ex_02.c	Sample main function

Files to turn-in:

Name	Description
new.c	new and delete implementations
point.c	Point class implementation
vertex.c	Vertex class implementation

The previous version of `new()` didn't let us pass parameters to the constructor. To fix this issue, you must provide a new version of `new()` making use of the C language's support for **variadic arguments**.



stdarg(3)

In addition, to fulfill other needs, you must provide an alternate version of `new()` called `va_new()`. Here are the prototypes of the functions to turn-in:

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



Constructors and destructors will no longer print out messages





It must be possible to use the functions like so:

```
#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 must create a new `Vertex` class, similar to `Point`.

The `Class` structure now contains a new *member function* called `__str__`, which returns a string. The `str` macro in `object.h` calls this member function.



`snprintf(3)`



You must ensure that the following source code:

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

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

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

    delete(point);
    delete(vertex);

    return 0;
}
```

Produces the following output:

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



## + EX\_03 - OBJECT ADDITION / SUBTRACTION

Provided files:

Name	Description
raise.h	raise macro
new.h	new and delete prototypes
object.h	Class structure definition
point.h	Point variable declaration
vertex.h	Vertex variable declaration
ex_03.c	Sample main function

Files to turn-in:

Name	Description
new.c	new and delete implementations
point.c	Point class implementation
vertex.c	Vertex class implementation

In this exercise, we'll simply add two new member functions to the `Class` structure, which will let users add and subtract objects. To comply with these changes, you must adapt your previous `point.c` and `vertex.c` files to implement the `__add__` and `__sub__` functions.

The following code:

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

int main()
{
    Object *p1 = new(Point, 12, 13);
    Object *p2 = new(Point, 2, 2);

    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)));

    Object *v1 = new(Vertex, 1, 2, 3);
    Object *v2 = new(Vertex, 4, 5, 6);

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

    return 0;
}
```



Should produce 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 - OBJECT ADDITION / SUBTRACTION

Provided files:

Name	Description
raise.h	raise macro
bool.h	bool type definition and true and false macros
new.h	new and delete prototypes
object.h	Class structure definition
int.h	Int variable declaration
float.h	Float variable declaration
char.h	Char variable declaration
ex_04.c	Sample main function

Files to turn-in:

Name	Description
new.c	new and delete implementations
int.c	Int class implementation
float.c	Float class implementation
char.c	Char class implementation

This exercise will once again extend the base class, in addition to re-writing some native C types. The new types you must implement are `Int`, `Float` and `Char`. Like in previous exercises, it must be possible to add and subtract objects of the same type, but we also want to be able to compare them.

Add the following comparison operators to the base class:

```
* == (__eq__)  
* < (__lt__)  
* > (__gt__)
```

These three classes will be used extensively in the next part. Operations and comparisons between objects of different types are not required, but could be a bonus :)



The following code must compile and run as expected:

```
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 make calling member functions easier.



## PART 2 - GENERIC CONTAINERS

In the previous part, we created simple types. This section will focus on writing containers. A container is an object that contains any type of object deriving from the base class.

Let's start by adding an *intermediate class*, in order to only add member functions to containers. Adding every member function to every class, like we've been doing so far, would be a lot of work. An intermediate class is simply a structure containing a `Class` variable, defined like so:

```
#include "object.h"

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

We have simply added a `newMethod` function pointer to the `Container` class. Notice this class *contains* the base class, and must therefore implement all its member functions.

Here's one approach to defining a container in a `.c` file:

```
#include "container.h"

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

static MyContainerClass _descr = {
    { // Container struct
        { // Class struct
            sizeof(MyContainerClass)
            "MyContainer",
            // All Class functions here
        }
    },
    0, // Members of MyContainer
};
```

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



The last concept this section introduces is *iterators*. An iterator lets you walk through a container, like an index in a table. To walk through an array in C, you would do something like this:

```
int tab[10];
for (size_t i = 0; i < 10; ++i)
    doStuff();
```

The problem with this approach is that when you decide to change the type of the container, switching to a linked list for instance, you had to change all the code, since iterating through a linked list is done completely differently. *Iterators* are a standardized technique for iterating over any type of container. Here is the same sample code, using iterators:

```
Container *tab = new(MyContainer);
for (Iterator *it = begin(tab); !it(it, end(tab)), incr(it))
    doStuff();
```

Notice that the code is very similar, and the meaning behind it stays the same. However, the implementation of the container remains completely hidden. The `incr` member function is exactly identical to the incrementation of the `i` variable in the previous example: it lets us *move forward* to the next element in the table. The definition of the `Iterator` class uses the same reasoning as that of `Container`. It is defined as an intermediate class, used to define other iterators, specific to each container.





## + EX\_05 - ARRAY

Provided files:

Name	Description
raise.h	raise macro
bool.h	bool type definition and true and false macros
new.h	new and delete prototypes
object.h	Class structure definition
container.h	Container structure definition
iterator.h	Iterator structure definition
int.h	Int variable declaration
float.h	Float variable declaration
char.h	Char variable declaration
array.h	Array variable declaration
array.c	Partial Array class implementation
ex_05.c	Sample main function

Files to turn-in:

Name	Description
new.c	new and delete implementations
int.c	Int class implementation
float.c	Float class implementation
char.c	Char class implementation
array.c	Array class implementation

In this exercise, we provided a partial implementation of the Array class, simulating a standard array. You must fill in the functions in the `array.c` file in order to obtain the following behavior:

The **constructor** of an Array takes as parameters its size (`size_t`), the type contained in the array (`Class *`) and the arguments for the type's constructor. For instance:

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

In this example, `tab` is an array of 10 `Floats`, each initialized with `0.0f`.

The **getitem** member function takes an index (`size_t`) as parameter and returns the object at that index (`Object *`).

The **setitem** member function takes an index (`size_t`) as parameter, along with all the arguments to create a new instance of the contained type. For instance:

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



In this example, the object at index 3 of `tab` will be replaced by a new `Float`, initialized with `42.042f`.

The `setval` member function of the table iterator works similarly:

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

In this example, the first element in the table will be replaced by a new `Float`, initialized with `3.14159265f`.



`va_new()`



## + EX\_06 - LIST

Provided files:

Name	Description
<code>raise.h</code>	raise macro
<code>bool.h</code>	bool type definition and true and false macros
<code>new.h</code>	new and delete prototypes
<code>object.h</code>	Class structure definition
<code>container.h</code>	Container structure definition
<code>iterator.h</code>	Iterator structure definition
<code>int.h</code>	Int variable declaration
<code>float.h</code>	Float variable declaration
<code>char.h</code>	Char variable declaration

Files to turn-in:

Name	Description
<code>new.c</code>	new and delete implementations
<code>int.c</code>	Int class implementation
<code>float.c</code>	Float class implementation
<code>char.c</code>	Char class implementation
<code>list.h</code>	List variable declaration
<code>list.c</code>	List class implementation
<code>ex_06.c</code>	Sample main function

Based on the Array class example, you must create a List class that lets users easily manipulate linked lists. You are free to choose the behaviors you want for the operators (such as `add`), but will have to justify your choices during the defense. Is it really relevant to multiply lists?

Feel free to add methods if you use a list-specific intermediate class. It must be possible to use your lists in conjunction with Array objects.

A set of tests must be provided in your sample `main` function, and these will also be evaluated during the defense.

In this part, you are free to modify `container.h`.



## BONUS

---

Impress us!

- Code additional containers (String again?)
- Add intermediate classes for Array and List, defining specific member functions for each of these:
  - `Array.resize`
  - `Array.push_back`
  - `List.push_front`, `List.push_back`
  - `List.pop_front`, `List.pop_back`
  - `List.front`, `List.back`
- Make the previous containers compatible with native C types (`int`, `float`, `char`)
- Make using all these systems safer (no more macros? A magic the number at the beginning of each class? Type verification?)
- Change the design to make it possible to define member functions in an intermediate class
- Make operations between different types possible:
  - `3.0f + 2 = 5.0f`
  - `3 * ['a'] = ['a', 'a', 'a']`
  - `2 * "hello" = "hellohello"`
  - etc...