



B3 - C++ Pool

B-PAV-242

Day 13

A Game of Toys



KOALA

42.0



Day 13

binary name: no binary
group size: 1
repository name: cpp_d13
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,...).



GENERAL SETPOINTS

READ THESE CAREFULLY

You will have no possible excuse if you end up with a 0 because you didn't follow one of these.



If you do half the exercises because you have comprehension problems, it's okay, it happens. But if you do half the exercises because you're lazy, and leave at 2PM, you **WILL** have problems. Do not tempt the devil.



Read the examples **CAREFULLY**. They might require things that weren't mentioned in the subject...



All output goes to the standard output and must be ended with a newline character, unless specified otherwise.



Remember: you're coding in C++ now, and not in C. Therefore, the following functions are **FORBIDDEN** and their use will be punished by a -42, no questions asked:

*alloc
*printf
free



Any use of the `friend` keyword will result in a -42



You are not allowed to use any library other than the C++ standard library.



It must be possible to include each of your header files independently from the others. Headers must include all their dependencies.



All your header files will be included in the correction `main`.



None of your files must contain a `main` function



THINK. Please.



THINK



T.H.I.N.K.! For Pony!



To avoid compilation problems during automated tests, please include all necessary files within your headers.

Please note that none of your files must contain a `main` function, unless specified otherwise. We will use our own `main` functions to compile and test your code.



This subject may be modified up to one hour before turn-in time!



UNIT TESTS

It is highly recommended to test your functions as you implement them. It is common practice to create and use what are called **unit tests**.

From now on, we expect you to write unit tests for your functions (when possible). To do so, please follow the instructions in the “**How to write Unit Tests**” document on the intranet, available [here](#).

Create a directory named `tests`. For each of the classes you turn in, create a file in that directory named `tests-CLASS-NAME.cpp` containing all the tests needed to cover all of the class' possible cases (regular or irregular).

Here is a sample set of unit tests for the **string** class:

```
#include <riterion/criterion.h>
```

```
Test(string, default_value)
{
    std::string s;
    cr_assert_eq(s, "");
}
```

```
Test(string, assign)
{
    std::string s;


    s = "test";
    cr_assert_eq(s, "test");
}
```

```
Test(string, append)
{
    std::string s("test");

    s += "ing";
    cr_assert_eq(s, "testing");
}
```



EXERCISE 0 - ENCAPSULATION

	Exercise: 00	points : 4
Encapsulation		
Turn-in directory: cpp_d13/ex00		
Compiler: g++	Compilation flags: -W -Wall -Wextra -Werror -std=c++14	
Makefile: No	Rules: n/a	
Files to turn in: Picture.h, Picture.cpp, Toy.h, Toy.cpp		
Notes: None		
Forbidden functions: 'using namespace' keyword		

We are going to create some basic toys for you to play with, each with a picture (so you can know what it looks like!). More features will be added to these toys in the following exercises.

Start by creating a `Picture` class to represent our toys' illustrations.

The class will contain:

- Publicly
 - `std::string data; // Our toy's ASCII art`
 - `bool getPictureFromFile(const std::string &file);`
Sets `data`'s value to the content of `file`. If an error occurs, `data` must be set to `"ERROR"` and the function must return `false`. Otherwise, it returns `true`.
 - `Picture(const std::string &file);`
Creates a `Picture` object by loading the content of `file`. If an error occurs, `data` must be set to `"ERROR"`.

Creating a `Picture` without a filename as parameter sets `data` to an empty string.

Now, create a `Toy` class. It must contain a `ToyType` enumeration with two fields: `BASIC_TOY` and `ALIEN`.

The `Toy` class will contain a type, a name and a picture, as well as the following member functions:

- `getType`
A getter for the toy's type (there is no setter, as the type will never change).
- `getName`
- `setName`
- `setAscii`
Takes a filename as parameter and sets the toy's picture to the file's content. Returns `true` if it succeeds, `false` otherwise.
- `getAscii` Returns the toy's picture as a string.
- A constructor taking no parameter, setting the toy's type to `BASIC_TOY`, its name to `"toy"` and its picture to an empty string.
- A constructor taking three parameters: the `ToyType`, a string containing the toy's name, and a string containing the picture's filename.




main.cpp

6



EXERCISE 1 - CANONICAL FORM

	Exercise: 01	points : 1
Canonical form		
Turn-in directory: <code>cpp_d13/ex01</code>		
Compiler: <code>g++</code>	Compilation flags: <code>-W -Wall -Wextra -Werror -std=c++14</code>	
Makefile: No	Rules: n/a	
Files to turn in: <code>Picture.h</code> , <code>Picture.cpp</code> , <code>Toy.h</code> , <code>Toy.cpp</code>		
Notes: None		
Forbidden functions: <code>'using namespace'</code> keyword		


Re-use the two classes from the previous exercise and make them comply with the canonical form.



This may imply more than meets the eye...



EXERCISE 2 - SIMPLE INHERITANCE

	Exercise: 02	points : 2
Simple inheritance		
Turn-in directory: cpp_d13/ex02		
Compiler: g++	Compilation flags: -W -Wall -Wextra -Werror -std=c++14	
Makefile: No	Rules: n/a	
Files to turn in: Picture.h, Picture.cpp, Toy.h, Toy.cpp, Buzz.h, Buzz.cpp, Woody.h, Woody.cpp		
Notes: None		
Forbidden functions: 'using namespace' keyword		

Add two values to the `ToyType` enumeration: `BUZZ` and `WOODY`, and create two new `Buzz` and `Woody` classes.


These two classes inherit from `Toy`, and will set their parent's attributes to the corresponding values upon construction:

- `type`: `BUZZ` and `WOODY`, respectively
- `name`: passed as parameter
- `ascii`: optionally passed as parameter; if no filename is provided, the objects will respectively load their picture from the `"buzz.txt"` and `"woody.txt"` files

It shouldn't be possible to create `Buzz` or `Woody` objects without a name.



EXERCISE 3 - PONYMORPHISM

	Exercise: 03	points : 2
Inheritance polymorphism		
Turn-in directory: cpp_d13/ex03		
Compiler: g++	Compilation flags: -W -Wall -Wextra -Werror -std=c++14	
Makefile: No	Rules: n/a	
Files to turn in: Picture.h, Picture.cpp, Toy.h, Toy.cpp, Buzz.h, Buzz.cpp, Woody.h, Woody.cpp		
Notes: None		
Forbidden functions: 'using namespace' keyword		

We'd like our toys to be able to speak. Add a `speak` method to the `Toy` class, taking the statement to say as a parameter.

This method will display the toy's name, followed by a space and the statement passed as parameter.

```
name "statement"
```

Overload this method in the `Buzz` and `Woody` classes in order to display, respectively:

```
BUZZ: name "statement"
```

and

```
WOODY: name "statement"
```

In all three cases, `name` is to be replaced with the toy's name and `statement` with the `string` passed as parameter. The double quotes in the examples must be printed.

The `speak` method must **not** be `const`. You'll understand why in the following exercises.



Here is a sample `main` function and its expected output:

```
#include <iostream>
#include "Toy.h"
#include "Buzz.h"
#include "Woody.h"

int main()
{
    std::unique_ptr<Toy> b(new Buzz("buzziiii"));
    std::unique_ptr<Toy> w(new Woody("wood"));
    std::unique_ptr<Toy> t(new Toy(Toy::ALIEN, "ET", "alien.txt"));


    b->speak("To the code, and beyond !!!!!!!");
    w->speak("There's a snake in my boot.");
    t->speak("the claaaaaaw");
}
```

main.cpp

```
~/B-PAV-242> ./a.out
BUZZ: buzziiii "To the code, and beyond !!!!!!!"
WOODY: wood "There's a snake in my boot."
ET "the claaaaaaw"
```



EXERCISE 4 - OPERATORS

	Exercise: 04	points : 3
Operator overloading		
Turn-in directory: cpp_d13/ex04		
Compiler: g++	Compilation flags: -W -Wall -Wextra -Werror -std=c++14	
Makefile: No	Rules: n/a	
Files to turn in: Picture.h, Picture.cpp, Toy.h, Toy.cpp, Buzz.h, Buzz.cpp, Woody.h, Woody.cpp		
Notes: None		
Forbidden functions: 'using namespace' keyword		

We will now add two operator overloads.

A first overload of the << operator, between an `ostream` and a `Toy`. This operator will print the toy's name, followed by its picture, on the standard output. The name and picture will have to be followed by a newline.

A second overload the << operator, between a `Toy` and a `string`. This operator will replace the `Toy`'s picture with the `string`.




main.cpp

REX

0/



EXERCISE 5 - NESTING

	Exercise: 05	points : 4
Nested Classes		
Turn-in directory: cpp_d13/ex05		
Compiler: g++	Compilation flags: -W -Wall -Wextra -Werror -std=c++14	
Makefile: No	Rules: n/a	
Files to turn in: Picture.h, Picture.cpp, Toy.h, Toy.cpp, Buzz.h, Buzz.cpp, Woody.h, Woody.cpp		
Notes: None		
Forbidden functions: 'using namespace' keyword		

We know some toys have several options: for example, our Buzz Lightyear toy can speak spanish! To illustrate this, add a `speak_es` method to the `Toy` class, with the same signature as `speak`. In the `Buzz` class, this method will have the same behavior as `speak` but will add *“senorita”* before and after the statement:

```
BUZZ: name senorita "statement" senorita
```

However, some toys don't speak spanish, so we have to handle this case. For every toy that can't speak spanish, the `speak_es` method won't display anything and will return `false`.

Let's make the most of our error handling in the `Toy` class. We currently have two possible error causes:

- `setAscii`
- `speak_es`

Both return `false` in the event an error occurred.

Create a nested `Error` class in `Toy` that will contain two methods and a public attribute:

- `what`: returns the error message:
 - *“bad new illustration”* if the error happened in `setAscii`
 - *“wrong mode”* if the error happened in `speak_es`
- `where`: returns the name of the function where the error occurred
- `type`: holds the error type



Moreover, `Error` will contain an `ErrorType` enum with the different error types:

- UNKNOWN
- PICTURE
- SPEAK

Add a `getLastError` to the `Toy` class that will return an `Error` object containing information about the last error that occurred. If no error happened, `getLastError` will return an `Error` instance with two empty strings for `what` and `where`, and will have `UNKNOWN` as its type.



Here is a sample `main` function and its expected output:

```
#include <iostream>
#include "Toy.h"
#include "Buzz.h"
#include "Woody.h"

int main()
{
    Woody w("wood");

    if (w.setAscii("file_who_does_not_exist.txt") == false)
    {
        auto e = w.getLastError();
        if (e.type == Toy::Error::PICTURE)
        {
            std::cout << "Error in " << e.where()
                      << ": " << e.what() << std::endl;
        }
    }

    if (w.speak_es("Woody does not have spanish mode") == false)
    {
        auto e = w.getLastError();
        if (e.type == Toy::Error::SPEAK)
        {
            std::cout << "Error in " << e.where()
                      << ": " << e.what() << std::endl;
        }
    }


    if (w.speak_es("Woody does not have spanish mode") == false)
    {
        auto e = w.getLastError();
        if (e.type == Toy::Error::SPEAK)
        {
            std::cout << "Error in " << e.where()
                      << ": " << e.what() << std::endl;
        }
    }
}
```

main.cpp

```
Terminal
~/B-PAV-242> ./a.out
Error in setAscii: bad new illustration
Error in speak_es: wrong mode
Error in speak_es: wrong mode
```




EXERCISE 6 - A TOY STORY

	Exercise: O6	points : 4
Member pointers		
Turn-in directory: cpp_d13/exO6		
Compiler: g++	Compilation flags: -W -Wall -Wextra -Werror -std=c++14	
Makefile: No	Rules: n/a	
Files to turn in: Picture.h, Picture.cpp, Toy.h, Toy.cpp, Buzz.h, Buzz.cpp, Woody.h, Woody.cpp, ToyStory.h, ToyStory.cpp		
Notes: None		
Forbidden functions: 'using namespace' keyword		

Create a `ToyStory` class which will tell stories about two toys.

`ToyStory` will contain a class `tellMeAStory` that will take 5 parameters:

- a filename containing the story
- the first `Toy`, which we'll call `toy1`
- a `Toy` method pointer taking a `string` as parameter and returning a boolean, which we'll call `func1`
- the second `Toy`, which we'll call `toy2`
- a `Toy` method pointer taking a `string` as parameter and returning a boolean, which we'll call `func2`

These `Toy` instances and method pointers are respectively associated. `toy1` is associated with `func1` and `toy2` is associated with `func2`.

The `tellMeAStory` function starts by printing the two `Toys`' pictures, each followed by a newline. It then reads the file given as parameter, and for each line in it, calls the method pointer associated to the toy. The toys will be called on a rotating basis:

- the first line will be sent to `func1` on `toy1`
- the second to `func2` on `toy2`
- the third to `func1` on `toy1`
- ...

If the line starts with *"picture:"*, it changes the picture of the toy which was supposed to be called. The toy's new picture is then set to the content of the file specified after the *"picture:"* mention. The toy's picture is then displayed.



For instance, with the following file:

```
salut
picture:ham.txt
coucou
a+
```

story.txt

The actions will be the following:

- Print `toy1`'s picture followed by a newline
- Print `toy2`'s picture followed by a newline
- Call `func1` on `toy1` with "`salut`"
- Set `toy2`'s picture to the content of the "`ham.txt`" file
- Print `toy2`'s picture
- Call `func2` on `toy2` with "`coucou`"
- Call `func1` on `toy1` with "`a+`"

`tellMeAStory` stops as soon as it encounters an error (if it fails to change a toy's picture, for instance). If an error occurs, you must print information about it using the following format:

```
where: what
```

`where` must be replaced with the error's `where` property, and `what` must be replaced with the error's `what` property.

If the file passed as parameter cannot be opened or read, you must print "`Bad Story`" to the standard output.

Here is a sample `main` function:

```
#include <iostream>
#include "Toy.h"
#include "ToyStory.h"
#include "Buzz.h"
#include "Woody.h"

int main()
{
    Buzz    b("buzzi");
    Woody   w("wood");

    ToyStory::tellMeAStory("superStory.txt", b, &Toy::speak_es, w,
                          &Toy::speak);
}
```

main.cpp