



C++ Pool - d13

Abstract: This document is the subject of d13

Contents

I	GENERAL RULES	2
II	Exercise 0	4
III	Exercise 1	7
IV	Exercise 2	8
V	Exercise 3	9
VI	Exercise 4	11
VII	Exercise 5	13
VIII	Exercise 6	16
	VIII.1 The tellMeAStory behavior	17

Chapter I

GENERAL RULES

- READ THE GENERAL RULES CAREFULLY !!
 - You will have no possible excuse if you end up with a 0 because you didn't follow one of the general rules
- GENERAL RULES :
 - 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.
 - Every function implemented in a header, or unprotected header, means 0 to the exercise.
 - Every class MUST have a constructor and a destructor.
 - Every output goes to the standard output, and will be ended by a newline, unless specified otherwise.
 - The imposed filenames must be followed TO THE LETTER, as well as class, functions and method names.
 - 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
* open, fopen, etc ...
```

* `using` keyword

- Files associated with a class will be `CLASS_NAME.hh` and `CLASS_NAME.cpp` (If applicable), unless specified otherwise.
 - Turn-in directories are `ex00`, `ex01`, ..., `exN`
 - Any use of `friend` will be punished by a -42, no questions asked.
 - Read the examples CAREFULLY. They might require things the subject doesn't say ...
 - These exercises require that you create lots of classes, but most of them are VERY short. So, don't be lazy !
 - Read ENTIRELY the subject of an exercise before you start it !
 - THINK. Please.
 - THINK. By Odin !
 - T.H.I.N.K ! For Pony !
- COMPILATION OF THE EXERCISES :
 - The Koalinette compiles your code with the following flags : `-W -Wall -Werror -Wextra -std=c++03`
 - To avoid compilation problems with the Koalinette, include every required headers in your headers.
 - Note that none of your files must contain a `main` function. We will use our own to compile and test your code.
 - This subject may be modified up to 4h before turn-in time. Refresh it regularly !
 - Turn in only required files.
 - The turn-in dirs are `(cpp_d13/exN)` , N being the exercise number

Chapter II

Exercise 0

	Exercise : 00	points : 4
Encapsulation		
Turn-in directory: <code>cpp_d13/ex00</code>		
Compiler: <code>g++</code>	Compilation flags: <code>-W -Wall -Werror</code>	
Makefile: No	Rules: <code>n/a</code>	
Files to turn in : <code>Picture.h</code> , <code>Picture.cpp</code> , <code>Toy.h</code> , <code>Toy.cpp</code>		
Remarks : <code>n/a</code>		
Forbidden functions : <code>None</code>		

We are going to create basic toys, each toy will be pictured. We will see in the following exercises more of these toys' functionalities.

First, create a class named `Picture` which will be our toy's illustration.

The class will contain:

- Publicly:
 - `std::string data;` // Our toy's ASCII art
 - `bool getPictureFromFile(const std::string& file);`
Extracts the file given which name will be given as parameter, and sets the `data` property to the file content. If an error occurs, the `data` property must be set to "ERROR" and the method must return `false` ; it returns `true` if successful.
 - `Picture(const std::string& file);` Creates a `Picture` object and loads the content of the file `file` as picture container. If an error occurs, the `data` property must be set to "ERROR" (same error handling as `getPictureFromFile`).

If we create a `Picture` without filename as parameter, `data` will be an empty string.

Now create a class named `Toy` .
This class will contain an enum named `ToyType` with two fields: `BASIC_TOY` and `ALIEN` .

Your `Toy` class will contain a type, a name and a picture. Moreover this class will have the following member functions:

- `getType`
A getter for the toy's type (there is no setter since the toy's type will never change).
- `getName` A getter for the toy's name.
- `setName` A setter for the toy's name.
- `setAscii` Takes the filename as parameter and sets the toy's picture with. Returns `true` if successful, `false` otherwise.
- `getAscii` returns the toy's picture, as a string.
- A constructor that does not take any argument, sets the toy's type to `BASIC_TOY` , its name to "toy", and creates a picture with an empty string.
- A constructor that takes 3 parameters: the `ToyType` , a string containing the toy's name, and a string containing the picture's file name.

```

1 #include <iostream>
2 #include "Toy.h"
3
4 int main()
5 {
6     Toy toto;
7     Toy ET(Toy::ALIEN, "green", "./alien.txt");
8
9     toto.setName("TOTO !");
10
11     if (toto.getType() == Toy::BASIC_TOY)
12         std::cout << "basic toy: " << toto.getName() << std::endl
13             << toto.getAscii() << std::endl;
14     if (ET.getType() == Toy::ALIEN)
15         std::cout << "this alien is: " << ET.getName() << std::endl
16             << ET.getAscii() << std::endl;
17     return 1337;
18 }

```

Output :


```

1 $>./a.out
2 basic toy: TOTO !
3
4 this alien is: green
5
6      _|_
7    ,_.-._' _' _-._,
8    \ (.) (.) (.) /
9    _ , '\_ -===- _/' , _
10 > |-----" " " "-----| <
11 "' "' --/_ _-@-\-- "' "'
12    |===L_I===|
13    \ /
14    _\__|__/_
15    ' " " " ' " " " '
16 $>

```

Chapter III


Exercise 1

 KOALA	Exercise : 01	points : 1
Canonical form		
Turn-in directory: <code>cpp_d13/ex01</code>		
Compiler: <code>g++</code>		Compilation flags: <code>-W -Wall -Werror -Wextra -std=c++03</code>
Makefile: No		Rules: <code>n/a</code>
Files to turn in : <code>Picture.h</code> , <code>Picture.cpp</code> , <code>Toy.h</code> , <code>Toy.cpp</code>		
Remarks : <code>n/a</code>		
Forbidden functions : <code>None</code>		

Take back the two classes made in the previous exercise and change them in order to make them compliant to the canonical form (think about what it involves). If you are doubtful about it, see your lesson or ask a Koala. Warning: ALL the attributes will be copied.

Chapter IV

Exercise 2

	Exercise : 02	points : 2
Simple inheritance		
Turn-in directory: <code>cpp_d13/ex02</code>		
Compiler: <code>g++</code>	Compilation flags: <code>-W -Wall -Werror -Wextra -std=c++03</code>	
Makefile: No	Rules: <code>n/a</code>	
Files to turn in : <code>Picture.h</code> , <code>Picture.cpp</code> , <code>Toy.h</code> , <code>Toy.cpp</code> , <code>Buzz.h</code> , <code>Buzz.cpp</code> , <code>Woody.h</code> , <code>Woody.cpp</code>		
Remarks : <code>n/a</code>		
Forbidden functions : <code>None</code>		

Add to the `ToyType` enum two new toys: `BUZZ` and `WOODY` . Then create two new classes: `Buzz` and `Woody` .


These two classes are `Toy` specialized subclasses, they will inherit from `Toy` . Each class will set the `Toy` attributes to the correct value at the object construction:

- `type` : respectively `BUZZ` and `WOODY`
- `name` : will be given as parameter
- `ascii` can be optionally be given as parameter ; if no filename is given, then the objects will respectively load their picture from the files “`buzz.txt`” and “`woody.txt`” in the working directory.

We should not be able to create a `Buzz` or a `Woody` object without giving any name.

Chapter V

Exercise 3

	Exercise : 03	points : 2
Polymorphism by inheritance		
Turn-in directory: <code>cpp_d13/ex03</code>		
Compiler: <code>g++</code>	Compilation flags: <code>-W -Wall -Werror -Wextra -std=c++03</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> , <code>Buzz.h</code> , <code>Buzz.cpp</code> , <code>Woody.h</code> , <code>Woody.cpp</code>		
Remarks : n/a		
Forbidden functions : None		

We now want to make our toys able to speak, so you will add a method named `speak` to the `Toy` class, that will take in parameter the statement to say.



beware `const`, `ref`, `lunar cycle`, `white rabbit`, and so on.

This method will display the toy's name followed by a space, the statement taken in parameter and a carriage return.

```
1 name "statement"
```

You will overload this method in the `Buzz` and `Woody` classes in order to display respectively:

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

and

```
1 WOODY: name ‘‘statement’’
```

In the three cases `name` is the toy's name and `statement` the string given in parameter. As displayed, you have to print the double quotes.

Unlike the usual way, the `speak` method must not be `const`, we'll see why in the following exercises.


```
1
2 #include <iostream>
3 #include ‘‘Toy.h’’
4 #include ‘‘Buzz.h’’
5 #include ‘‘Woody.h’’
6
7 int main()
8 {
9     Toy *b = new Buzz(‘‘buzziiii’’);
10    Toy *w = new Woody(‘‘wood’’);
11    Toy *t = new Toy(Toy::ALIEN, ‘‘ET’’, ‘‘alien.txt’’);
12
13    b->speak(‘‘To the code, and beyond !!!!!!!!!’’);
14    w->speak(‘‘There’s a snake in my boot.’’);
15    t->speak(‘‘the claaaaaaw’’);
16 }
```

Output :

```
1 $>./a.out
2 BUZZ: buzziiii ‘‘To the code, and beyond !!!!!!!!!’’
3 WOODY: wood ‘‘There’s a snake in my boot.’’
4 ET ‘‘the claaaaaaw’’
5 $>
```

Chapter VI

Exercise 4

	Exercise : 04	points : 3
Operator overloading		
Turn-in directory: <code>cpp_d13/ex04</code>		
Compiler: <code>g++</code>	Compilation flags: <code>-W -Wall -Werror -Wextra -std=c++03</code>	
Makefile: No	Rules: <code>n/a</code>	
Files to turn in : <code>Picture.h</code> , <code>Picture.cpp</code> , <code>Toy.h</code> , <code>Toy.cpp</code> , <code>Buzz.h</code> , <code>Buzz.cpp</code> , <code>Woody.h</code> , <code>Woody.cpp</code>		
Remarks : <code>n/a</code>		
Forbidden functions : <code>None</code>		

We will now set up two operator overloads.

A first overload of the operator `<<` between an `ostream` and a `Toy` :

This will display on the standard output the toy's name followed by its picture. The name and the picture will have to be followed by a carriage return.

A second overload of the operator `<<` between a `Toy` and a `string` :

This will replace the toy's picture with the `string` .



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

int main()
{
    Toy a(Toy::BASIC_TOY, "REX", "rex.txt");


    std::cout << a;
    a << "\\o/";
    std::cout << a;
}
```

```
$> ./a.out
REX
```

REX
 \o/
 \$>

Chapter VII

Exercise 5

	Exercise : 05	points : 4
Nested Classes		
Turn-in directory: <code>cpp_d13/ex05</code>		
Compiler: <code>g++</code>	Compilation flags: <code>-W -Wall -Werror -Wextra -std=c++03</code>	
Makefile: No	Rules: <code>n/a</code>	
Files to turn in : <code>Picture.h</code> , <code>Picture.cpp</code> , <code>Toy.h</code> , <code>Toy.cpp</code> , <code>Buzz.h</code> , <code>Buzz.cpp</code> , <code>Woody.h</code> , <code>Woody.cpp</code>		
Remarks : <code>n/a</code>		
Forbidden functions : <code>None</code>		

We know that some toys have several modes: for example our Buzz Lightyear can speak spanish!

So we will add for him a method to the class `Toy` , named `speak_es` , which will have the same signature than `speak` .

In the `Buzz` class this method will have the same behavior than `speak` but will add “senorita” before and after the statement:

```
1 BUZZ: name senorita “statement” senorita
```

But all the 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 of them return `false` in the event an error occurred.

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

- `what`
will return the error message:
 - “bad new illustration” for an error that happened in `setAscii`
 - “wrong mode” for an error that happened in `speak_es`
- `where` will return the function name where the error occurred.
- `type` will contain the error type.

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

- UNKNOWN
- PICTURE
- SPEAK

Then, you will add to the `Toy` class a member function named `getLastError` that will return an `Error` instance 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 for type `UNKNOWN`.

A short main with its required output will shed light on it:


```
1 #include <iostream>
2 #include 'Toy.h'
3 #include 'Buzz.h'
4 #include 'Woody.h'
5
6 int main()
7 {
8     Woody w('wood');
9
10    if (w.setAscii('file_who_does_not_exist.txt') == false)
11    {
12        Toy::Error e = w.getLastError();
13        if (e.type == Toy::Error::PICTURE)
14        {
15            std::cout << 'Error in ' << e.where()
16                      << ': ' << e.what() << std::endl;
17        }
18    }
19
20    if (w.speak_es('Woody does not have spanish mode') == false)
21    {
22        Toy::Error e = w.getLastError();
23        if (e.type == Toy::Error::SPEAK)
24        {
25            std::cout << 'Error in ' << e.where()
26                      << ': ' << e.what() << std::endl;
27        }
28    }
29
30    if (w.speak_es('Woody does not have spanish mode') == false)
31    {
32        Toy::Error e = w.getLastError();
33        if (e.type == Toy::Error::SPEAK)
34        {
35            std::cout << 'Error in ' << e.where()
36                      << ': ' << e.what() << std::endl;
37        }
38    }
39 }
```

Output :

```
1 $>./a.out
2 Error in setAscii: bad new illustration
3 Error in speak_es: wrong mode
4 Error in speak_es: wrong mode
5 $>
```


Chapter VIII

Exercice 6

	Exercise : 06	points : 4
Pointers to members		
Turn-in directory: <code>cpp_d13/ex06</code>		
Compiler: <code>g++</code>	Compilation flags: <code>-W -Wall -Werror -Wextra -std=c++03</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> , <code>Buzz.h</code> , <code>Buzz.cpp</code> , <code>Woody.h</code> , <code>Woody.cpp</code> , <code>ToyStory.h</code> , <code>ToyStory.cpp</code>		
Remarks : n/a		
Forbidden functions : None		

You will create a `ToyStory` class, which will tell stories about two toys.

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

- a filename containing the story
- the first `Toy` , let's say `toy1`
- a method pointer of the class `Toy` taking a `string` in parameter and returning a boolean, let's say `func1`
- the second `Toy` , let's say `toy2`
- a method pointer of the class `Toy` taking a `string` in parameter and returning a boolean, let's say `func2`

The received `Toy` instances and method pointers are respectively related. `toy1` is related with `func1` and `toy2` is related to `func2` .

VIII.1 The tellMeAStory behavior

It begins with the two toys pictures, both of them followed by a carriage return.

`tellMeAStory` will read the file given as parameter, and for each line in it, the function will call 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 one to `func2` on `toy2`
- The third one to `func1` on `toy1`
- And so on until there is no more line to read in the file.

If the line is beginning by “picture:”, then it will change the picture of the toy which was supposed to be called. The new toy’s picture is the content of the specified file following the “picture:” mention. We then display the new toy’s picture.

For example, with the following file:

```
1 $>cat story.txt
2 salut
3 picture:ham.txt
4 coucou
5 a+
6 $>
```

The actions will be the following:

- Displays the `toy1` ’s picture followed by a carriage return
- Displays the `toy2` ’s picture followed by a carriage return
- `func1` is called on `toy1` , with “salut”.
- We change the `toy2` ’s picture with the content of the file “ham.txt”
- We display the new `toy2` ’s picture
- `func2` is called on `toy2` with “coucou”
- `func1` is called on `toy1` with “a+”

`tellMeAStory` stops at the first error encountered (if, for example it cannot change the toy's picture). Your “speak*” functions will also return a `bool`. If an error occurs, you will have to print information about this error using the following format:

```
1 where: what
```

With “where” replaced with the error's `where` property, and “what” replaced by the error's `what` property.

One last thing, if the file given as parameter cannot be read and opened, you have to print “Bad Story” on the standard output.

Here is a sample `main` which should help you understand the `tellMeAStory` behavior:

```
1 #include <iostream>
2 #include 'Toy.h'
3 #include 'ToyStory.h'
4 #include 'Buzz.h'
5 #include 'Woody.h'
6
7 int main()
8 {
9     Buzz b('buzzi');
10    Woody w('wood');
11
12    ToyStory::tellMeAStory('superStory.txt', b, &Toy::speak_es, w,
13                           &Toy::speak);
14 }
```