C++ Pool - d08

Abstract: *This document is the subject for d08.*

Contents

Chapter I

BASIC RULES

- BASIC RULES:
  - If you do half the exercises because you have comprehension problems, it's okay, it can happen. But if you do half the exercises because you are 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.
  - Each 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, function 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 question asked:
    - `*alloc`
    - `*printf`
    - `free`
    - `using keyword`
  - Generally, files associated to a class will always be `CLASS_NAME.h` and `CLASS_NAME.cpp` (class name can be in lower case if applicable.)
  - Turn in directories are ex00, ex01, ..., exN

- Any use of `"friend"` will result in the grade of -42, `no questions asked` .
- You must read the examples thoroughly. They can contain requirements that are not obvious in the exercises description.
- You are asked to turn in an important number of classes. However, these classes are pretty short. Slackers not accepted!
- Read each exercise FULLY before starting it!
- USE YOUR BRAIN, Please!

- EXERCISES COMPILATION:
  - The Koalinette compiles your code with the following flags: `-W -Wall -Werror -Wextra -std=c++03`
  - In order to avoid compilation errors, please include the required files in your include files (*.hh).
  - Please note that none of your files must contain the `main` function except if it is explicitly asked. We will use our own `main` function to compile and test your code.
  - Remember that you are coding in C++ now. Thus, the compiler if g++!
  - The exercise description can be modified until 4h before the final turn in time! Look at it regularly!
  - Turn in only required files.
  - The turn in repository is: `(cpp_d08)/exN` (N being the exercise number).

Chapter II

Exercise 0

	Exercise : 00	points : 4
You, Droid lead designer		
Turn-in directory: `(cpp_d08)/ex00`		
Compiler: `g++`	Compilation flags: `-W -Wall -Werror -Wextra -std=c++03`	
Makefile: `No`	Rules: `n/a`	
Files to turn in : `droid.hh, droid.cpp`		
Remarks : `n/a`		
Forbidden functions : `None`		

Hey you, yes you, over there. From now, on you are a lead designer. What is it for? Well, you are now chief engineer designer for my coming Droid army! Why you? Just because you were there. Stop babbling now and get to work. As a start we need a cheap Droid.

These are the specifications:

- This Droid accepts as a parameter its serial number which is a `std::string` . The Droid can be constructed without this serial number. In this case, the serial number is an empty string.
- The Droid has a copy constructor for the replication and an affectation operator for the replacement. This is the easiest way for damaged Droids.
- The Droid must also possess the following properties:
  - `Id : std::string` // the serial as indicated earlier.
  - `Energy : size_t` // the remaining energy before changing the batteries
  - `Attack : size_t const` // Attack power

- `Toughness : size_t const` // Droid resistance
- `Status : std::string *` // Current status of the Droid.

When constructed, `Energy` , `Attack` , `Toughness` and `Status` are in respective order equal to 50, 25, 15 and "Standing by".
Each of these attributes is private. Thus they have a getter the form of which is `get[Property]` and a setter the form of which is `set[Property]` . `const` values have no setter, obviously.

- The Droid is in charge of its `Status` and takes ownership of it. The Droid is in charge of its destruction.
- It is necessary to know if two Droids are identical or different thanks to the following operators: `==` and `!=` . Warning: We are not interested in knowing if it is the `same` Droid. Two Droids are equivalent if they have the same characteristics.
- You have to overload the operator `«` , to reload the Droid.
  A Droid can't have more than 100 nor less than 0 of energy. It subtracts the value that he requires to reload its batteries. It must be possible to chain calls.
- Last minute information: The Droid can talk. It's more convenient and less boring for a scrap pile. When a Droid is created, it displays:

```
1   Droid 'serial' Activated
```

(with ' around the serial). When replicated:

```
1   Droid 'serial' Activated, Memory Dumped
```

When destructed, it displays:

```
1   Droid 'serial' Destroyed
```

- More important. For every visit to the great `std::cout` , it must display:

```
1   Droid 'serial', Status, Energy
```


EPITECH. L'ÉCOLE DE L'INNOVATION ET DE L'EXPERTISE INFORMATIQUE

```
int main()
{
    Droid d;
    Droid d1(''Avenger'');
    size_t Durasel = 200;

    std::cout << d << std::endl;
    std::cout << d1 << std::endl;
    d = d1;
    d.setStatus(new std::string(''Kill Kill Kill!''));
    d << Durasel;
    std::cout << d << ''--'' << Durasel << std::endl;
    Droid d2 = d;
    d.setId(''Rex'');
    std::cout << (d2 != d) << std::endl;
    return (0);
}
```

The expected output:

```
tipiak@ender:~/workbox/d09/ex_0$ ./proggie | cat -e
Droid '' Activated$
Droid 'Avenger' Activated$
Droid '', Standing by, 50$
Droid 'Avenger', Standing by, 50$
Droid 'Avenger', Kill Kill Kill!, 100--150$
Droid 'Avenger' Activated, Memory Dumped$
1$
Droid 'Avenger' Destroyed$
Droid 'Avenger' Destroyed$
Droid 'Rex' Destroyed$
tipiak@ender:~/workbox/d09/ex_0$
```

Chapter III

Exercise 1

	Exercise : 01	points : 2
DroidMemory		
Turn-in directory: `(cpp_d08)/ex01`		
Compiler: `g++`	Compilation flags: `-W -Wall -Werror -Wextra -std=c++03`	
Makefile: `No`	Rules: `n/a`	
Files to turn in : `droid.hh, droid.cpp, droidmemory.hh, droidmemory.cpp`		
Remarks : `n/a`		
Forbidden functions : `malloc, free, *printf`		

Ok, That's not too bad so far. It seems you have some skills.

- Now it is necessary to improve the procedures to deploy and replace our Droids. First things first, Droid must not be able to be constructed without parameters anymore. It's up to you to solve this issue. It generates too much trouble and identity crisis for Droids. Droids losing their mind is never good for an army.

- Secondly, the comparison between two Droids must only be possible based on their `Status` . A Droid is a Droid, and knowing if they perform the same task is good enough.

- Thirdly, you must add a dedicated recording memory for the battle field information. This memory is called `DroidMemory` . This class contains the following properties:

  - `FingerPrint` : `size_t` // an id of the `DroidMemory` .
  - `Exp` : `size_t` // Value of acquired experience.

  With their own getter/setter.

Well, you do understand that in reality, Droid memory is more complex than that, but this is a good approximation of what we are interested in.
There are several possible interactions with this memory:

- operator « : Add the experience of the right member to the left one, then do `xor` the `FingerPrint` of the right to the one on the left. Operator « can be chained.
- operator » : Same as « but the other way around.
- operator += :
  - If the right operand is a `DroidMemory` , does the same as «.
  - If the parameter is a `size_t` then add to `Exp` , then do a xor on the fingerprint with the very same parameter `size_t` . The operator += can be chained.
- operator + : Does the same as += but return a new `DroidMemory` it can be chained. Obviously operands `MUST NOT` be modified.

`Although the actions of « and of += are identical, these two operators don't have the same associativity. Therefore, intrinsically, they won't have the same behavior when chained. even if they perform the same action. For those of you who don't understand the previous sentence, it is normal that a chaining of « and a chaining of += even with the same input does not result in the same output.`

The construction of a new `DroidMemory` initializes `Exp` to 0 and `Fingerprint` to a random value thanks to a call to the `random` function. No need to call the `srandom` , the lead Designer will do it for you.

Passing by `std::cout` will display with ' :

```
1 DroidMemory '[FingerPrint]', [Exp]
```

You have to add the following property to the Droid class:
- BattleData : DroidMemory * // Droid memory.
With its own getter/setter. Obviously, the `BattleData` is built during the construction of a `Droid` . A Droid with no memory is pretty much a waste.

```
int main()
{
    DroidMemory dm;
    DroidMemory dn;
    DroidMemory dg;
    dm += 42;
    DroidMemory dn1 = dm;
    std::cout << dm << std::endl;
    dn << dm;
    dn >> dm;
    dn << dm;
    std::cout << dn << std::endl;
    std::cout << dm << std::endl;
    dg = dm + dn1;
}
```

```
[tipiak@Calysto ex01]$ ./proggie | cat -e
DroidMemory '1804289357', 42$
DroidMemory '1804289357', 126$
DroidMemory '846930886', 84$
[tipiak@Calysto ex01]
```


EPITECH. L'ECOLE DE L'INNOVATION ET DE L'EXPERTISE INFORMATIQUE

Chapter IV

Exercise 2

	Exercise : 02	points : 2
Roger Roger		
Turn-in directory: `(cpp_d08)/ex02`		
Compiler: `g++`	Compilation flags: `-W -Wall -Werror -Wextra -std=c++03`	
Makefile: `No`	Rules: `n/a`	
Files to turn in : `droid.hh, droid.cpp, droidmemory.hh, droidmemory.cpp`		
Remarks : `n/a`		
Forbidden functions : `None`		

Obviously, it is possible to copy/overwrite a `DroidMemory` by another one. You need to overload == and != to be able to compare `DroidMemory` on the values of `Exp` and of `FingerPrint` .

You also need to overload <, >, <= and >= only on the `Exp` value, acquired by the `DroidMemory` . The comparison is done with a `DroidMemory` or directly with a `size_t` .

Something more: when a Droid is assigned, or when the Droid is constructed by copy, the Droid doesn't copy its energy. Just its `Id` , its `Status` and its `BattleData` . When needed, the energy is initialized at 50.

The King of the Guild for the Enforcement of Enforcable Laws just caught me and charged me for violating the Energy Saving Act.

You have to add the possibility to assign a task to a Droid thanks to the operator(). This operator accepts two parameters: a `std::string const *` which represents the task, and a `size_t` which represents the required experience to perform this task.

Each task assignation costs 10 of energy. If energy falls to 0 or is not sufficient enough, the assignation returns `false` and the `Status` is consequently updated. In case there is not enough energy, the remaining energy is consumed anyway.

During the energy check, if the Droid possesses enough experience to achieve the task, it returns `true` , updates its status (see below) and increases its `Exp` by half the re-

quired experience. If not, it returns `false` and increases its experience by the total value of the required experience (this is what is called learning from your failure!)

After a task is assigned to a Droid, it is necessary to update its `Status` , as follows:

- "task - Completed!" When task succesful
- "task - Failed!" When task failed
- "Battery Low" When there is no more energy.

```
int main()
{
    DroidMemory dm;
    DroidMemory dn;
    DroidMemory dg;
    dm += 42;
    DroidMemory dn1 = dm;
    std::cout << dm << std::endl;
    dn << dm;
    dn >> dm;
    dn << dm;
    std::cout << dn << std::endl;
    std::cout << dm << std::endl;
    dg = dm + dn1;

    Droid d(``rudolf'');
    Droid d2(``gaston'');
    size_t DuraSell = 40;
    d << DuraSell;
    d.setStatus(new std::string(``having some reset''));
    d2.setStatus(new std::string(``having some reset''));
    if (d2 != d && !(d == d2))
    std::cout << ``a droid is a droid, all its matter is what it's doing'' <<
    std::endl;
    d(new std::string(``take a coffee''), 20);
    std::cout << d << std::endl;
    while (d(new std::string(``Patrol around''), 20))
    {
        if (!d(new std::string(``Shoot some ennemies''), 50))
            d(new std::string(``Run Away''), 20);
        std::cout << d << std::endl;
    }
    std::cout << d << std::endl;
    return (0);
}
```

```
[tipiak@Calysto ex01]$ ./proggie | cat -e
DroidMemory '1804289357', 42$
DroidMemory '1804289357', 126$
DroidMemory '846930886', 84$
Droid 'rudolf' Activated$
Droid 'gaston' Activated$
Droid 'rudolf', take a coffee - Failed!, 80$
Droid 'rudolf', Run Away - Completed!, 50$
Droid 'rudolf', Shoot some ennemies - Completed!, 30$
Droid 'rudolf', Shoot some ennemies - Completed!, 10$
Droid 'rudolf', Battery Low, 0$
Droid 'gaston' Destroyed$
```

```
Droid 'rudolf' Destroyed$
[tipiak@Calysto ex02]$
```

Droid are good enough by now. Let's move on to the Carrier. What do you say? Getting tired? Tssss....

Chapter V

Exercise 3

	Exercise : 03	points : 4
Carrier		
Turn-in directory: `(cpp_d08)/ex03`		
Compiler: `g++`	Compilation flags: `-W -Wall -Werror -Wextra -std=c++03`	
Makefile: `No`	Rules: `n/a`	
Files to turn in : `droid.hh, droid.cpp, carrier.hh, carrier.cpp, droidmemory.hh, droidmemory.cpp`		
Remarks : `n/a`		
Forbidden functions : `None`		

We were talking about the Carrier. Each Carrier can carry up to 5 Droids. A Carrier has the following properties:

- `Id : std::string;`
  Because its more convenient.
- `Energy : size_t;`
  Battery works too.
- `Attack : size_t const;`
  Good guess, the Carrier carries weapon, we are at war!
- `Toughness : size_t const;` // Its resistance.
- `Speed : size_t;` // Its speed.
- `Droids : Droid*[5];` // Droid slots.

A `Carrier` is built by passing to it its `Id` . The default constructor initialize `Id` properties, `Energy` , `Attack` , `Toughness` to the respective values: ", 300, 100, 90.

Speed is 0 if there are no Droid on board. Otherwise speed value is 100 as soon as there is a Droid (A Carrier needs a pilot ...) However the speed decrease by 10 for every Droid on board, including the first one. (These piles of junk are pretty heavy.)

Boarding a Droid is done thanks to the operator `«` . Disembarking is done thanks to `»` . If 5 Droids are already on board or if there are no more Droids to disembark, then nothing happens. A slot is designated as empty if its pointer is `NULL` .

When a Droid board the Carrier, its pointer must be set to `NULL` so that it is not possible to have a Droid in two different slots. It is not possible to copy a `Carrier` . When destructed it will also destruct every Droid on board.

It is possible to access any slot thanks to the operator []. You can replace any post that you want without considering the consequences. Remember you can also call it if the `Carrier` is `const` ...

The operator `~` allows to run a complete check-up of the `Carrier` , This is convenient to re evaluate the speed in case of a free rider.

The `Carrier` moves thanks to the operator (), by giving it `X` and `Y` coordinates of the `int` type. The energy cost of moving is calculated as follows:

```
(abs(X) + abs(Y)) * (10 + (NbDroid))
```

Where `NbDroid` is the number of Droids on board of the `Carrier` . A `Carrier` can't move if its speed is 0, or if the energy is too low. In these cases, the operation returns `false` . If the move succeed, then it returns `true` .

It is possible to recharge a `Carrier` the same way as for a `Droid` , thanks to the operator `«` . Follow the same guidelines that for the `Droid` , except that the maximum `Energy` of a `Carrier` is 600.

Don't forget the `std::cout` .The format is available in the example. By now, you should be used to it and you should be able to find it by yourself.

```
int main()
{
    Carrier c(``HellExpress'');
    Droid *d1= new Droid(``Commander'');
    Droid *d2 = new Droid(``Sergent'');
    Droid *d3 = new Droid(``Troufiont'');
    Droid *d4 = new Droid(``Groupie'');
    Droid *d5 = new Droid(``BeerHolder'');

    c << d1 << d2 << d3 << d4 << d5;
    std::cout << c.getSpeed() << d1 << std::endl;
    c >> d1 >> d2 >> d3;
    std::cout << c.getSpeed() << std::endl;
    c[0] = d1;
    std::cout << (~c).getSpeed() << std::endl;
    c(4, 2);
    std::cout << c << std::endl;
    c(-15, 4);
    std::cout << c << std::endl;
    c[3] = 0;
    c[4] = 0;
    (~c)(-15, 4);
    std::cout << c << std::endl;
    return (0);
}
```

```
tipiak@ender:~/workbox/d09/ex03$ ./proggie | cat -e
Droid 'Commander' Activated$
Droid 'Sergent' Activated$
Droid 'Troufiont' Activated$
Droid 'Groupie' Activated$
Droid 'BeerHolder' Activated$
500$
80$
70$
Carrier 'HellExpress' Droid(s) on-board:$
[0] : Droid 'Commander', Standing by, 50$
[1] : Free$
[2] : Free$
[3] : Droid 'Groupie', Standing by, 50$
[4] : Droid 'BeerHolder', Standing by, 50$
Speed : 70, Energy 222$
Carrier 'HellExpress' Droid(s) on-board:$
[0] : Droid 'Commander', Standing by, 50$
[1] : Free$
[2] : Free$
[3] : Droid 'Groupie', Standing by, 50$
[4] : Droid 'BeerHolder', Standing by, 50$
```

```
Speed : 70, Energy 222$
Carrier 'HellExpress' Droid(s) on-board:$
[0] : Droid 'Commander', Standing by, 50$
[1] : Free$
[2] : Free$
[3] : Free$
[4] : Free$
Speed : 90, Energy 13$
Droid 'Commander' Destroyed$
tipiak@ender:~/workbox/d09/ex03/$
```

Chapter VI

Exercise 4

	Exercise : 04	points : 4
The Robot factory - First part		
Turn-in directory: `(cpp_d08)/ex04`		
Compiler: `g++`	Compilation flags: `-W -Wall -Werror -Wextra -std=c++03`	
Makefile: `No`	Rules: `n/a`	
Files to turn in : `supply.hh, supply.cpp, droid.hh, droid.cpp, droidmemory.hh, droidmemory.cpp`		
Remarks : `n/a`		
Forbidden functions : `None`		

It's pretty nice to create prototypes, but to win a war, it is necessary to produce heavily and quickly. You now have to build a brand new robot factory. The class is named `DroidFactory` . We'll come back later on that. A factory requires resources. In this case they are Iron and Silicon. For practical purpose we are considering a standard container of these resources.

You have to create a `Supply` class, which is the container of resources. This class contains:

- Type : `Types` // An enum to declare in the class.
- Amount : `size_t` // The quantity of the given resource.
- Wrecks : `Droid**` // An array of `Droid*` to recycle. This is trendy.

The enum `Types` (to declare in the class `Supply` , as stated earlier...) must contain the following values:

- Iron = 1
- Silicon = 2

- Wreck = 3

In case that `Type` is `Iron` or `Silicon` (exclusively), then `Amount` represents the quantity of the container. In case that `Type` is `Wreck` , then `Amount` represents the number of Droids in the array `Wrecks` . Droids in the container are arranged on a rotary rack. Like for the barrel of a revolver. The operator * allows to access to a pointer on `Droid` . Therefore, the operator -> allows direct access to its members. To scroll through the `Droids` you can use `++` (prefix) on the container, or $--$ (prefix). Please note that it's a cyclical process!!!

The access to the value `Amount` is done through the overload of the implicit cast operator on `size_t` (think `const` ... or not!). You can add a classical getter if you wish, but keep in mind that factories are working this way.)

The class `Supply` must not be "copy-constructable". And must be constructible in two different ways:

- The first one is to pass as parameters the type and the quantity of resources.
- The second one is the same that the first one, but with an added `Droid**` , representing Droids to recycle.

The operator ! allows to purge the container, meaning that `Amount` is set to zero, and all the `Droids` in the container are deleted.

The operator == allows to test the type of resource in the container. Don't forget to implement the != operator too !

The `Supply` class destructs the `Droid*` that belongs to it, if their value is different than `NULL` when the destructor is called. As your Mum taught you, avoid the waste. So make sure it doesn't happen.

This is it for `Supply` , and don't forget the `std::cout` .

```
int main()
{
    Droid **w;
    w = new Droid*[10];
    char c = '0';
    for (int i = 0; i < 3; ++i)
    {
        w[i] = new Droid(std::string(``wreck: ``) + (char)(c + i));
    }
    Supply s1(Supply::Silicon, 42);
    Supply s2(Supply::Iron, 70);
    Supply s3(Supply::Wreck, 3, w);

    std::cout << s3 << std::endl;
    size_t s = s2;
    std::cout << s << std::endl;
    std::cout << *(*(--s3)) << std::endl;
    std::cout << *(++s3)->getStatus() << std::endl;
    ++s3;
    *s3 = 0;
    std::cout << *s3 << std::endl;
    std::cout << s2 << std::endl;
    std::cout << !s3 << std::endl;
    return 0;
}
```

```
tipiak@ender:~/workbox/d09/ex04$ ./proggie | cat -e
Droid 'wreck: 0' Activated$
Droid 'wreck: 1' Activated$
Droid 'wreck: 2' Activated$
Supply : 3, Wreck$
Droid 'wreck: 0', Standing by, 50$
Droid 'wreck: 1', Standing by, 50$
Droid 'wreck: 2', Standing by, 50$
70$
Droid 'wreck: 2', Standing by, 50$
Standing by$
0$
Supply : 70, Iron$
Droid 'wreck: 0' Destroyed$
Droid 'wreck: 2' Destroyed$
Supply : 0, Wreck$
tipiak@ender:~/workbox/d09/ex04$
tipiak@ender:~/workbox/d09/ex04$
```

Chapter VII

Exercise 5

	Exercise : 05	points : 4
The Robot factory - Part two		
Turn-in directory: `(cpp_d08)/ex05`		
Compiler: `g++`	Compilation flags: `-W -Wall -Werror -Wextra -std=c++03`	
Makefile: `No`	Rules: `n/a`	
Files to turn in : `droidfactory.hh, droidfactory.cpp, droid.hh, droid.cpp, supply.hh, supply.cpp, droidmemory.hh, droidmemory.cpp`		
Remarks : `n/a`		
Forbidden functions : `None`		

Now that the resource system is up and running, it is time to create your first factory. As mentioned earlier, the class is called `DroidFactory` . As for any factory, resources are required. These resources are provided by the `Supply` container as described earlier.

Supplying is done thanks to the operator «. Logistical requirements forbid the use of pointers. This would create too much trouble with referencing. You have to handle that by yourself. It is of course possible to chain the supplying with several containers one after the other. At the end of the supplying process, the container is empty.

Each container bring in a quantity of resources of the indicated type. When the Factory is recycling, then 80 units of `Iron` and 30 units of `Silicon` are extracted from each Droid. It is also necessary to consider the `BattleData` proper to each Droid, depending on the ratio indicated later.

100 units of `Iron` and 50 units of `Silicon` are required to build a Droid. The `BattleData` is distributed according to a factor proper to each factory.

Calling the operator » allows to create a new `Droid` and returns a `Droid*` or returns `NULL` if the conditions are not satisfied.

The ratio is passed as parameter when the factory is constructed. The ratio type is `size_t` and is initialized to 2 by default. Each construction must be explicit. I guess

you understand what I mean, don't you?

`DroidFactory` is canonical, but this is between you and me. Of course the new factory is identical to the original one.

The factory keeps its stock, but nobody needs to access it. The factory displays its status on the standard output (Our dear friend coût) Thanks to the operator « :

```
DroidFactory status report :
Iron : XX
Silicon : XX
Exp : XX
End of status report.
```

The ratio is used as described below:

- For each Droid creation, this one gets the quantity of `Exp` available in the factory, minus this total value of this `Exp` divided by the ratio.
- When a Droid is recycled, if the `Exp` of the Droid being recycled is superior to the `Exp` of the factory, then the `Exp` of the factory becomes the addition of its current `Exp` with: the absolute difference of the Droid `Exp` and the factory `Exp` , divided by the ratio.
- It is possible to change this ratio thanks to the operators ++ and −−, postfix and prefix (for the 2 of them.)

While I am thinking about it, It can be useful to use alternative roads. Therefore you have to overload the operator » for the transportation of the containers and on « for the creation of a Droid. It is forbidden to modify the `Supply` et `Droid` classes. These overloads must be turned in within the files `droidfactory` .hh.cpp as they are part of the interface of `DroidFactory` . See the main of the example.

```
int main()
{
    DroidFactory factory(3);
    Droid **w;
    Droid *newbie;
    w = new Droid*[10];
    char c = '0';
    for (int i = 0; i < 3; ++i)
    {
        w[i] = new Droid(std::string(``wreck: ``) + (char)(c + i));
        *(w[i]->getBattleData()) += (i * 100);
    }
    Supply s1(Supply::Silicon, 42);
    Supply s2(Supply::Iron, 70);
    Supply s3(Supply::Wreck, 3, w);

    factory >> newbie;

    std::cout << newbie << std::endl;

    factory << s1 << s2;
    std::cout << factory << std::endl;
    s3 >> factory >> newbie;
    std::cout << factory << std::endl;
    factory++ >> newbie;
    std::cout << *newbie->getBattleData() << std::endl;
    --factory >> newbie;
    std::cout << *newbie->getBattleData() << std::endl;
    return 0;
}
```

```
[tipiak@Calysto correction]$ ./proggie | cat -e
Droid 'wreck: 0' Activated$
Droid 'wreck: 1' Activated$
Droid 'wreck: 2' Activated$
0$
DroidFactory status report :$
Iron : 70$
Silicon : 42$
Exp : 0$
End of status report.$
Droid 'wreck: 0' Destroyed$
Droid 'wreck: 1' Destroyed$
Droid 'wreck: 2' Destroyed$
Droid '' Activated$
DroidFactory status report :$
Iron : 210$
Silicon : 82$
```


EPITECH. L'ECOLE DE L'INNOVATION ET DE L'EXPERTISE INFORMATIQUE

```
Exp : 88$
End of status report.$
Droid '' Activated$
DroidMemory '1957747793', 59$
Droid '' Activated$
DroidMemory '424238335', 59$
[tipiak@Calysto correction]$
```