# B3 - C++ Pool

B-CPP-300

# Day 08

The Trade Federation

1.15

# Day 08

| | |
|---|---|
| **binary name:** | no binary |
| **group size:** | 1 |
| **repository name:** | cpp_d08 |
| **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,...).

All your exercises will be compiled with `g++` **and the** `-W -Wall -Wextra -Werror` **flags**, unless specified otherwise.

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

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. It will include your header files.

For each exercise, the files must be turned-in in a separate directory called **exXX** where XX is the exercise number (for instance `ex01`), unless specified otherwise.

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

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

The `*alloc`, `free`, `*printf`, `open` and `fopen` functions, as well as the `using namespace` keyword, are forbidden in C++.
By the way, `friend` is forbidden too, as well as any library except the standard one.

## 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 functions you turn in, create a file in that directory named `tests-FUNCTION_NAME.c` containing all the tests needed to cover all of the exercise's possible cases (regular or irregular).

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

```
#include <criterion/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 - Droids

**Turn in**: `Droid.hpp`, `Droid.cpp`

Hey you! Yes… you, over there.
From now on, you are a lead designer. What for?
Well, you are now chief engineer designer for my upcoming **Droid army**!
Why you? Just because you were there.
Stop babbling now, and get to work.

Start by creating a cheap Droid with the following specifications:

- The `Droid` takes as a parameter its serial number, which is an `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 replication, as well as an assignment operator for replacement.
  This is the easiest solution to damaged `Droids`.
- The `Droid` also has the following properties:

  `Id`, the `Droid`'s serial number, stored as an `std::string`
  `Energy`, the remaining energy before the `Droid`'s batteries need to be changed, stored as a `size_t`
  `Attack`, the `Droid`'s attack power, stored as a `const size_t`
  `Toughness`, the `Droid`'s resistance, stored as a `const size_t`
  `Status`, the `Droid`'s current status, stored as an `std::string *`

  Upon construction, `Energy`, `Attack`, `Toughness` and `Status` are respectively set to 50, 25, 15 and "Standing by".

  Each of these attributes is private. They therefore 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 whether two `Droids` are identical or not, thanks to the `==` and `!=` operators.
  Be careful: we don't care whether we are comparing the **same** `Droid`.
  Two `Droids` are considered identical if they have the same characteristics.
- Overload the `<<` operator to reload the `Droid`.
  A `Droid` can't have more than 100 nor less than 0 `Energy`.
  It substracts the value it requires to reload its batteries from the other operand.
  It must be possible to chain calls.

- This just in: the `Droid` can talk.
  Upon creation, it prints (including the single quotes):

```
Droid 'serial' Activated
```

  When replicated:

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

  When destroyed, it prints:

```
Droid 'serial' Destroyed
```

- Whenever it is directed to `std::cout`, a `Droid` prints:

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

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

```
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;
}
```

Terminal

```
~/B-CPP-300> ./a.out | 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$
```

## Exercise 1 - DroidMemory

**Turn in**: `Droid.hpp/cpp`, `DroidMemory.hpp/cpp`

Ok, so far so good.
It seems you may have some skills.

- It is now time to improve the `Droid` deployment and replacement procedures.
  First things first, it shouldn't be possible to construct a `Droid` without parameters anymore.
  It's up to you to make it so.
  It simply created too much trouble and identity crisis for `Droids`.
  `Droids` losing their minds is never good for an army.
- `Droid` comparisons should only take into account their `Status`.
  A `Droid` is a `Droid`, and knowing if they are working on the same task is good enough.
- Add a dedicated recording memory for battlefield information to `Droids`.
  This memory will be called `DroidMemory`, and have the following properties:
  - `FingerPrint`, the `DroidMemory`'s id, stored as a `size_t`
  - `Exp`, the experience acquired, stored as a `size_t`

Each of these properties will have their own getter and setter.

Of course you understand that droid memory is much more complex in reality, but this is a good approximation of what we are interested in.
It is possible to interact with this memory in several ways:

- `operator<<`: adds the experience of the right-hand-side operand to the left, and then performs a `xor` of the `FingerPrint` of the right-hand-side operand on that of the left-hand-side one. `operator<<` can be chained.
- `operator>>`: same as `operator<<`, but the other way around.
- `operator+=`:
  - if the right-hand-side operand is a `DroidMemory`, does the same as `operator<<`
  - if the right-hand-side operand is a `size_t`, adds it to `Exp`, then performs a `xor` of that `size_t` on the `FingerPrint`.
    `operator+=` can be chained.
- `operator+`: Does the same as `+=` but returns a new `DroidMemory` instead. `operator+` can be chained. Operands MUST NOT be modified, obviously.

Although the actions of << and += 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 setence, it is normal that a chaining of << does not result in the same output as one of +=.

Upon construction, a `DroidMemory`'s `Exp` is set to 0 and its `FingerPrint` to a random value, thanks to a call to the `random` function.
No need to call the `srandom` function, the lead Designer will do it for you.

Sending the `DroidMemory` through `std::cout` will print the following, including the single quotes:

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

Add the following property to the `Droid` class:

```
DroidMemory *BattleData;
```

with its own getter and setter.
Obviously, the `BattleData` is created during a `Droid`'s construction.
A `Droid` with no memory is pretty much a waste.

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

```
int main()
{
        DroidMemory mem1;
        mem1 += 42;

        DroidMemory mem2 = mem1;
        std::cout << mem1 << std::endl;

        DroidMemory mem3;
        mem3 << mem1;
        mem3 >> mem1;
        mem3 << mem1;

        std::cout << mem3 << std::endl;
        std::cout << mem1 << std::endl;
}
```

Terminal

```
~/B-CPP-300> ./a.out | cat -e
DroidMemory '1804289357', 42$
DroidMemory '1804289357', 126$
DroidMemory '846930886', 84$
```

## EXERCISE 2 - ROGER ROGER

**Turn in**: `Droid.hpp/cpp`, `DroidMemory.hpp/cpp`

Obviously, it is possible to copy a `DroidMemory` into another.
Overload the `==` and `!=` operators so that they compare the `Exp` and `FingerPrint` values of the `DroidMemories`.
Overload the `<`, `>`, `<=` and `>=` operators so that they compare the `Exp` values.
`DroidMemories` can be compared to other `DroidMemories` or to `size_t`s directly.

When a `Droid` is assigned to, or copy-constructed, it doesn't copy its energy but only its `Id`, `Status` and `BattleData`.
When needed, the energy is set to 50.
The **King of the Guild for the Enforcement of Enforcable Laws** just prosecuted me for violating the Energy Saving Act.

Make it possible to assign tasks to `Droids` using `operator()`.
The operator will take two parameters: a `const std::string *` representing the task and a `size_t` representing the experience required to perform the task.

Each task assignment costs 10 energy points.
If the `Droid`'s energy reaches 0 or is not sufficient, the assignment returns `false` and the `Status` is consequently updated.
If there is not enough energy, what little remains is consumed anyway.

When checking the energy level, if the `Droid` has enough experience to achieve the task, the assignment returns `true`, updates its status (see below) and increases its `Exp` by half the required experience points.
If not, it returns `false` and increases its experience by the total amount of required experience points (this is called learning from your mistakes).

Once a task is assigned to a `Droid`, its `Status` must be updated as follows:

- *"task - Completed!"* when the task is successful
- *"task - Failed!"* when the task failed
- *"Battery Low"* when there is not enough energy

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

```
static void testMemory()
{
        DroidMemory mem1;
        mem1 += 42;
        std::cout << mem1 << std::endl;

        DroidMemory mem2;
        mem2 << mem1;
        mem2 >> mem1;
        mem2 << mem1;

        std::cout << mem2 << std::endl;
        std::cout << mem1 << std::endl;
```

```
        DroidMemory mem3 = mem1;
        DroidMemory mem4;
        mem4 = mem1 + mem3;
}

static void testDroids()
{
        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;
}

int main()
{
        testMemory();
        testDroids();
        return 0;
}
```

Terminal

```
~/B-CPP-300> ./a.out | 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$
```

## Exercise 3 - Carrier

**Turn in**: `Droid.hpp/cpp`, `DroidMemory.hpp/cpp`, `Carrier.hpp/cpp`

Each `Carrier` can carry up to 5 `Droids`.
A `Carrier` has the following properties:

- `Id`: `std::string`
- `Energy`: `size_t`
- `Attack`: `const size_t`
- `Toughness`: `const size_t`
- `Speed`: `size_t`
- `Droids`: `Droid*[5]`

A `Carrier` is built by providing its `Id`.
The default constructor sets `Id`, `Energy`, `Attack` and `Toughness` to "", 300, 100 and 90, respectively.

`Speed` is set to 0 if there are no `Droids` on board.
Once a `Droid` is in the `Carrier`, `Speed` is set to 100 (a `Carrier` needs a pilot).
However, the speed decreases by 10 for every `Droid` on board, including the first one.

Boarding a `Droid` is done through the << operator.
Disembarking is done through the >> operator.
If 5 `Droids` are already on board, or if there are no more `Droids` to disembark, nothing happens.
A slot is considered empty if its pointer is `nullptr`.

When a `Droid` boards the `Carrier`, its pointer must be set to `nullptr` so that it is not possible to have a `Droid` in two different slots.
It is not possible to copy `Carriers`.
Upon destruction, a `Carrier` destroys all `Droids` on board.

`Droid` slots can be accessed using the `[]` operator.
`Droids` can be replaced by others using this operator.
Keep in mind that it is also possible to use this operator on `const Carriers`.

The `\~` operator runs a complete check-up on the `Carrier`.
It provides a convenient way to re-evaluate the speed of the `Carrier` in case of a free-rider.

The `Carrier` can be moved using the parenthesis operator, by providing it with `x` and `y` coordinates (which are `int`s).
The energy cost is calculated as follows:

```
(abs(x) + abs(y)) * (10 * (NbDroid))
```

where `NbDroid` is the number of `Droids` on the `Carrier`.

A `Carrier` can't move if its `Speed` is 0, or if it doesn't have enough energy.
In these cases, the operation returns `false`.

If it can move, the operation returns `true`.

`Carriers` can be recharged like `Droids`, using the `<<` operator.
Follow the same guidelines as those for `Droids`, except the maximum `Energy` level of a `Carrier` is 600.

Don't forget that `Carriers` can be printed to `std::cout`.
The format can be found in the example.
By now

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

```
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;
}
```

Terminal

```
~/B-CPP-300> ./a.out | 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$
```

## Exercise 4 - Factory

**Turn in**: `Droid.hpp/cpp`, `DroidMemory.hpp/cpp`, `Carrier.hpp/cpp`, `Supply.hpp/cpp`

Creating prototypes is fine, but we'll need to have a fast and heavy production line in order to win the war.
It is time for you to create a brand new robot factory: `DroidFactory`.
A factory requires resources.
In our case, these will be `Iron` and `Silicon`.
For practical purposes, we will implement a container for these resources.

Create a `Supply` class which will serve as a resource container.
It must have the following properties:

- `Type`: `Types` (an enum to declare in the class)
- `Amount`: `size_t`, the quantity of a given resource
- `Wrecks`: `Droid**`, an array of `Droid*` to recycle, I heard it was trendy

The `Types` enum, nested in the `Supply` class, must have the following values:

- `Iron` = 1
- `Silicon` = 2
- `Wreck` = 3

If `Type` is `Iron` or `Silicon`, `Amount` is the quantity of the resource held in the container.
If `Type` is `Wreck`, `Amount` is the number of `Droids` in the `Wrecks` array.
`Droids` in the container are stored in a rotating rack.

Think of it like the barrel of a revolver.

The `*` operator provides access to a `Droid` pointer.
Therefore, the `->` operator provides direct access to its members.
The prefix `++` and `--` operators make it possible to scroll through the `Droids`.

Keep in mind the process is cyclic.

The `Amount` value can be accessed through an implicit cast to a `size_t` (`const`? no `const`? Both?).
The `Supply` class must not be copy-constructible, and can be constructed in two different ways:

- passing the type and resource quantity as parameters,
- passing those two parameters, in addition to a `Droid**` set of `Droids` to recycle.

The container can be purged thanks to the `!` operator.
`Amount` is then set to 0, and all the `Droids` in the container are destroyed.
The `==` operator makes it possible to check the type of resource in the container.

Remember to also implement the `!=` operator.

The `Supply` class destroys the `Droids` that belong to it.
That's it for the `Supply` class. And don't forget `std::cout`.

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

```
int main()
{
        Droid **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;
}
```

Terminal

```
~/B-CPP-300> ./a.out | 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$
```

## EXERCISE 5 - FACTORY II

**Turn in**: `Droid.hpp/cpp`, `DroidMemory.hpp/cpp`, `Carrier.hpp/cpp`, `Supply.hpp/cpp`, `DroidFactory.hpp/cpp`

Now that the resource system is up and running, it is time to create our first factory.
As we mentioned in the previous exercise, the class is called `DroidFactory`.
As for any factory, resources are required, and will be provided by the `Supply` container described previously.

Supplying is done through the << operator.
Logistic requirements forbid the use of pointers, as those would lead to too much trouble with referencing.
You'll have to handle it yourself.
It is of course possible to chain the supplying with several containers one after the other.
At the end of the supply process, a container is emptied.

Each container provides a given quantity of resources of a given type.
When the `Factory` is recycling, 80 units of `Iron` and 30 units of `Silicon` are extracted from each `Droid`.
The `BattleData` held in each `Droid` also play a role, depending on the ratio indicated later in this instruction manual.

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

Calling the >> operator makes it possible to create a new `Droid` and returns a `Droid *`, or `nullptr` if the requirements are not met.

The ratio is passed as a parameter when constructing the factory.
It is a `size_t` and defaults to 2. Each construction must be explicit.
Of course, you know what I mean, don't you?

`DroidFactory` is canonical.
Of course, new factories are identical to their model.

The factory keeps its stock but nobody can access it.
It can print its status to the standard output through the << operator:

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

The ratio is used as described below:

- when creating a `Droid`, it gets the amount of `Exp` available in the factory, minus that amount divided by the ratio,
- when a `Droid` is recycled, if the `Exp` of the `Droid` being recycled is greater than that of the factory, the `Exp` of the factory becomes the sum of:
  1. its current `Exp`,
  2. the absolute difference of the `Droid`'s `Exp` and the factory's `Exp`, divided by the ratio,
- it is possible to change this ratio using the prefix or postfix `++` and `--` operators.

While we're at it, it can be useful to have alternative paths…
Overload the `>>` operator for loading containers, and the `>>` operator for creating a `Droid`. It is forbidden to modify the `Droid` and `Supply` classes.
These overloads must be turned in within the `DroidFactory.hpp` and `DroidFactory.cpp` files, as they are part of `DroidFactory`'s interface.

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

```
int main()
{
        DroidFactory factory(3);
        Droid **w = new Droid*[10];
        Droid *newbie;
        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;
}
```

Terminal

```
~/B-CPP-300> ./a.out | 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$
Exp :  88$
End of status report.$
Droid’‘Activated$
DroidMemory ’1957747793’, 59$
Droid ’‘Activated$
DroidMemory ’424238335’, 59$
```