Piscine C++ - d08

Alexandre "Tipiak" BOSSARD bossar_a@epitech.eu

Abstract: *Ce document est le sujet du d08*

Table des matières

Chapitre I

REMARQUES GÉNÉRALES

- REMARQUES GÉNÉRALES :
  - Si vous faites la moitié des exercices car vous avez du mal, c'est normal. Par contre, si vous faites la moitié des exercices par flemme et vous tirez à 14h, vous AUREZ des surprises. Ne tentez pas le diable.
  - Toute fonction implémentée dans un header ou header non protègé signifie 0 à l'exercice.
  - Toutes les classes doivent posséder un constructeur et un destructeur.
  - Toutes les sorties se font sur la sortie standard et sont terminées par un retour à la ligne sauf si le contraire est precisé explicitement.
  - Les noms de fichiers qui vous sont imposés doivent être respectés À LA LETTRE, de même que les noms de classes et de fonctions membres / méthodes.
  - Rappelez-vous que vous faites du C++ et non plus du C. Par conséquent, les fonctions suivantes sont INTERDITES, et leur utilisation sera sanctionnée par un -42 :
    - `*alloc`
    - `*printf`
    - `free`
  - De facon générale, les fichers associés à une classe seront toujours NOM_DE_LA_CLASSE.hh et NOM_DE_LA_CLASSE.cpp (s'il y a lieu).
  - Les repértoires de rendus sont ex00, ex01, ..., exN
  - Toute utilisation de `friend` se soldera par un -42, `no questions asked` .
  - Lisez attentivement les exemples, ils peuvent requérir des choses que le sujet ne dit pas...

- Ces exercices vous demandent de rendre beaucoup de classes, mais la plupart sont TRÈS courtes si vous faites ca intelligemment. Donc, halte à la flemme !
- Lisez ENTIÈREMENT le sujet d'un exercice avant de le commencer !
- REFLÉCHISSEZ. Par pitié.

- COMPILATION DES EXERCICES :
  - La moulinette compile votre code avec les flags : `-W -Wall -Werror`
  - Pour éviter les problèmes de compilation de la moulinette, incluez les fichiers nécéssaires dans vos fichiers include (*.hh).
  - Notez bien qu'aucun de vos fichiers ne doit contenir de fonction `main` . Nous utiliserons notre propre fonction `main` pour compiler et tester votre code.
  - Rappelez-vous, on fait du C++ maintenant, donc le compilateur est g++ !
  - Ce sujet peut être modifié jusqu'à 4h avant le rendu. Rafraichissez-le régulièrement !
  - Le repértoire de rendu est : `(DÉPOT SVN - piscine_cpp_d08-promo-login_x)/exN` (N étant bien sur le numéro de l'exercice).

Chapitre II

Exercice 0

	Exercice : 00	points : 4
Vous Grand concepteur, de Droid		
Répertoire de rendu: `(DÉPOT SVN - piscine_cpp_d08-promo-login_x)/ex00`		
Compilateur : `g++`	Flags de compilation: `-Wall -Wextra -Werror`	
Makefile : `Non`	Règles : `n/a`	
Fichiers a rendre : `droid.hh, droid.cpp`		
Remarques : `n/a`		
Fonctions Interdites : `Aucune`		

Vous, oui vous, là. À partir de maintenant je vous fais grand concepteur. À quoi ca vous sert ? Eh bien je viens de vous faire ingénieur/designeur en chef de ma future grande armée de droid ! Pourquoi vous ? Parce que vous étiez là.
Bon cessez de discuter et mettez vous au travail tout de suite. Pour commencer, il nous faudrait un droid. Bon marché, cela va de soit.

Voici le cahier des charges :

- Ce Droid devra prendre en paramètre son matricule de type `std::string` . Il peut être construit sans celui-ci, son matricule vaudra alors une chaîne vide.
- Il aura aussi un constructeur par copie pour la réplication et un opérateur d'affectation pour le remplacement. C'est la manière la plus simple pour les droids défectueux.
- Ce n'est pas tout, ce droid devra posséder ces propriétés :
  - `Id : std::string` // le matricule cité plus haut
  - `Energy : size_t` // l'énergie restante avant de changer les piles
  - `Attack : size_t const` // la puissance d'attaque
  - `Toughness : size_t const` // sa résistance

◦ `Status : std::string *` // le statut actuel du droid.

Lors d'une construction, `Energy` , `Attack` , `Toughness` et `Status` valent respectivement, 50, 25, 15 et "Standing by".
Chacun des attributs est évidemment privé. Donc ils ont tous un getter de la forme `get[Property]` et un setter de la forme `set[Property]` . Les valeurs `const` n'ont toutefois pas de setter, cela va de soit.

- Le Droid a la responsabilité de gèrer son `Status` et en prend la possession. Il a donc la charge d'en assurer sa destruction.
- C'est pas fini, On devra, pour chaque droid, savoir si ils sont identiques ou différents, grace aux opérateurs `==` et `!=` . Attention on se moque de savoir si c'est le `même` droid, deux droids sont égaux si ils ont les mêmes caractéristiques.
- Vous surchargerez aussi l'opérateur `«` , pour pouvoir recharger notre droid.
  Un droid ne peut pas avoir plus de 100 d'énergie et moins de zéro. Il soustraira la valeur dont il s'est servi pour regénerer ses batteries. On doit pouvoir chaîner les appels.
- Ah! oui, j'oubliais, le droid sera doté de la parole, c'est plus pratique et moins ennuyeux, pour un tas de ferraille. Donc quand le droid est créé il affichera :

```
1   Droid 'Matricule' Activated
```

(avec les ' autour du matricule). Et, dans le cas d'un réplication :

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

Lors de sa destruction il affichera

```
1   Droid 'Matricule' Destroyed
```

- Et le plus important, à chaque passage devant le grand `std::cout` , il devra afficher :

```
1   Droid 'Matricule', Status, Energy
```

```
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);
}
```

Et la sortie attendue :

```
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$
```

Chapitre III

Exercice 1

	Exercice : 01	points : 2
DroidMemory		
Répertoire de rendu: `(DÉPOT SVN - piscine_cpp_d08-promo-login_x)/ex01`		
Compilateur : `g++`	Flags de compilation: `-Wall -Wextra -Werror`	
Makefile : `Non`	Règles : `n/a`	
Fichiers a rendre : `droid.hh, droid.cpp, droidmemory.hh, droidmemory.cpp`		
Remarques : `n/a`		
Fonctions Interdites : `malloc, free, *printf`		

Bon, c'est pas du mauvais boulot, Le grand Géniteur n'a pas fait de vous un manchot.

- Il va falloir améliorer les procédures de déploiement et de remplacement de nos droids. Premièrement on ne doit plus pouvoir construire un Droid sans paramètre. Débrouillez-vous : ca pose trop de problèmes, et de crises d'identité chez les droids. Des droids qui perdent la tête dans une armée, c'est jamais bon.
- Deuxièmement, la comparaison d'un droid à un autre ne doit se faire que sur son `Status` . Un droid c'est un droid, ca nous est bien suffisant de savoir si ils font la même chose ou pas.
- Troisièmement, vous ajouterez une mémoire d'enregistrement des données de bataille, que vous appellerez `DroidMemory` . Cette classe contient les propriétés suivantes :

  ◦ `FingerPrint : size_t` // un identifiant de la `DroidMemory` .

  ◦ `Exp : size_t` // la valeur de l'expérience acquise.

  Avec leurs getter/setter respectifs.

Bon en réalite vous vous doutez bien que c'est bien plus complexe que ca la mémoire

d'un droid, mais c'est une bonne approximation de ce qui nous intéresse.

Il y a plusieurs interactions possibles sur cette mémoire :

- opérateur « : Ajoute l'expérience du membre de droite à celui de gauche, puis fait `xor` le `FingerPrint` de droite à celui de gauche. On doit pouvoir chainer l'opérateur «.
- opérateur » : Pareil que « mais dans l'autre sens.
- opérateur += :
  - Si l'opérande de droite est un `DroidMemory` , fait la meme chose que «.
  - Si le paramètre est un `size_t` on l'ajoute a l' `Exp` , puis on fait un xor sur le fingerprint avec ce même paramètre `size_t` . On doit pouvoir chaîner l'opérateur +=.
- opérateur + : Fait la meme chose que += mais retourne un nouveau `DroidMemory` On doit pouvoir le chainer. Evidement les opérandes `NE` sont `PAS` modifiées.

Bien que les actions de « et += soient identiques, ces deux opérateurs n'ont pas la même associativité. Donc intrinsèquement ils ne pourront pas avoir le même comportement lors du chainage, même s'ils font effectivement la même chose. Pour ceux qui ne comprennent pas la phrase précédente, il est normal qu'un chaînage de « et un autre chaînage de += avec la même entrée ne produira pas la même sortie.

La construction d'une nouvelle `DroidMemory` initialise `Exp` à 0 et `Fingerprint` à une valeur aléatoire grace à un appel à la fonction `random` . Pas besoin de faire appel à `srandom` , le grand Concepteur le fera pour vous.

Un passage devant `std::cout` affichera, avec les ' :

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

Vous ajouterez donc la propriété suivante à votre classe Droid:
- BattleData : DroidMemory * // La mémoire du droid.

Avec ses getter/setter. Évidement la `BattleData` est construite lors de la construction d'un `Droid` . Un droid sans mémoire c'est pas super pratique.

```
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]
```

Chapitre IV

Exercice 2

	Exercice : 02	points : 2
Roger Roger		
Répertoire de rendu: `(DÉPOT SVN - piscine_cpp_d08-promo-login_x)/ex02`		
Compilateur : `g++`	Flags de compilation: `-Wall -Wextra -Werror`	
Makefile : `Non`	Règles : `n/a`	
Fichiers a rendre : `droid.hh, droid.cpp, droidmemory.hh, droidmemory.cpp`		
Remarques : `n/a`		
Fonctions Interdites : `Aucune`		

Évidemment, il est possible de copier/d'écraser une `DroidMemory` par une autre. Vous surchargerez == et != pour pouvoir comparer nos `DroidMemory` sur les valeurs de `Exp` et de `FingerPrint` .

Vous surchargerez aussi <, >, <= et >= uniquement sur la valeur de l' `Exp` acquise par la `DroidMemory` . La comparaison peut se faire avec une `DroidMemory` ou directement avec un `size_t` .

Autre chose, lors de l'assignation d'un droid ou lors de sa construction par copie, le droid ne copie plus son énergie : juste son `Id` , son `Status` et sa `BattleData` . Quant à elle, son énergie est initialisée, si besoin est, à 50. Le King de l'Organisation de l'Application des Lois Applicables m'est tombé dessus, m'accusant d'avoir violé les lois de conservation de l'énergie.

Vous ajouterez la possibilite d'assigner une tâche à un droid via l'opérateur(). Celui-ci prend deux paramêtres, un `std::string const *` pour la tâche, et un `size_t` qui représente l'expérience requise pour exécuter cette dernière.

Chaque assignation de tâche coûte 10 d'énergie. Si l'énergie atteint 0 ou est insuffisante, l'assignation retourne `false` et le `Status` est mis à jour en conséquence. Dans le cas ou il n'y a pas assez d'énergie, le reste de l'énergie est quand même consommée.

Puis lorsque la vérification de l'énergie est faite, si le droid possède l'expérience suffisante pour exécuter cette tâche il retourne `true` , met à jour son statut (voir plus bas) et

augmente son `Exp` de la moitié de l'expérience requise. Dans le cas contraire, retourne `false` et augmente son expérience de la valeur de l'expérience requise (c'est ce qu'on appelle apprendre de ses échecs !).

Suite à une assignation de tâche vous devez mettre à jour le `Status` du Droid de la manière suivante:

- "tache - Completed!" dans le cas d'une réussite
- "tache - Failed!" dans le cas d'un échec
- "Battery Low" si il n'y a plus d'énergie.

```
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]$
```

Voila ca devrait suffir pour le droid, passons au Carrier. Comment ça vous êtes déjà fatigué ? Tsss...

Chapitre V

Exercice 3

	Exercice : 03	points : 4
Carrier		
Répertoire de rendu: `(DÉPOT SVN - piscine_cpp_d08-promo-login_x)/ex03`		
Compilateur : `g++`	Flags de compilation: `-Wall -Wextra -Werror`	
Makefile : `Non`	Règles : `n/a`	
Fichiers a rendre : `droid.hh, droid.cpp, carrier.hh, carrier.cpp, droidmemory.hh, droidmemory.cpp`		
Remarques : `n/a`		
Fonctions Interdites : `Aucune`		

Donc je disais le Carrier. Il doit pouvoir transporter jusqu'à 5 Droids. Un Carrier a les propriétés suivantes :

- `Id : std::string;`
  Parce que c'est plus pratique.
- `Energy : size_t;`
  Ben ouai, ca marche aussi à pile...
- `Attack : size_t const;`
  Oui, notre Carrier est armé, c'est la guerre!
- `Toughness : size_t const;` // Sa resistance.
- `Speed : size_t;` // Sa vitesse.
- `Droids : Droid*[5];` // Les emplacements a Droids.

Un `Carrier` se construit en lui passant son `Id` . Le constructeur par défaut initialisera les propriétés `Id` , `Energy` , `Attack` , `Toughness` aux valeurs suivantes ", 300, 100, 90. Speed vaut zéro si il n'y a aucun droid à bord. Elle vaut 100 dès qu'il y a un droid à bord du Carrier (ben oui faut bien un pilote) et est décrementée de 10 pour chaque Droid ajouté, dès le PREMIER Droid (c'est que ca pèse, un tas de ferraille).

L'embarquement d'un droid se fait grâce à l'opérateur `«` et le débarquement grace à `»` . S'il y a déjà 5 droids embarqués ou qu'il n'y a plus de droid à débarquer il ne se passe rien. Un emplacement est consideré comme vide si son pointeur vaut `NULL` .

Quand un droid debarque vous setterez son pointeur à `NULL` de manière à les empêcher d'être à deux endroits différents. Un `Carrier` n'est pas copiable. Et sa destruction entraîne celle de tous les droids a son bord, logique.

On peut accéder à chaque emplacement grâce à l'opérateur []. Vous pouvez donc brutalement remplacer le poste que vous voulez, sans vous préoccuper des conséquences. N'oubliez pas que l'on peut aussi l'appeler si le `Carrier` est `const` ...

L'opérateur `~` permet de lancer un check-up complet du `Carrier` , pratique pour pouvoir réévaluer la vitesse en cas de passagers clandestins.

On déplace le `Carrier` grâce a l'opérateur() en lui fournissant des coordonnées `X` et `Y` de type `int` . Le cout du déplacement en énergie se calcule comme suit :

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

Où `NbDroid` est bien évidement le nombre de droid a bord du `Carrier` . Un `Carrier` ne peut pas bouger si la vitesse vaut zéro, ou s'il n'y a pas assez d'énergie pour effectuer le déplacement. L'opération retournera donc `false` , ou `true` si elle a bien été éxecutée.

On peut bien evidement recharger un `Carrier` de la même manière qu'un `Droid` grace a l'opérateur `«` . Suivez les mêmes instructions que pour le `Droid` si vous ne vous en souvenez plus, au détail près que l' `Energy` max d'un `Carrier` est 600.

N'oubliez pas le grand `std::cout` . Pour le formatage c'est dans l'exemple, vous êtes habitués maintenant, vous devriez vous y retrouver.

```
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/$
```

Chapitre VI

Exercice 4

	Exercice : 04	points : 4
L'Usine à robots - Première partie		
Répertoire de rendu: `(DÉPOT SVN - piscine_cpp_d08-promo-login_x)/ex04`		
Compilateur : `g++`	Flags de compilation: `-Wall -Wextra -Werror`	
Makefile : `Non`	Règles : `n/a`	
Fichiers a rendre : `supply.hh, supply.cpp, droid.hh, droid.cpp, droidmemory.hh, droidmemory.cpp`		
Remarques : `n/a`		
Fonctions Interdites : `Aucune`		

Bon c'est bien gentil de faire des prototypes et de la conception mais pour gagner une guerre il faut produire tout ce petit monde en masse et vite. Vous allez donc me construire une usine à robot toute neuve. Votre classe s'appellera `DroidFactory` . Mais j'y reviendrais plus tard. Une usine ca a besoin de ressources, dans notre cas du fer et du silicium. On va donc formaliser tout ca sous forme d'un conteneur de livraison standard, c'est plus pratique.

Vous créerez une classe `Supply` , ce sera notre conteneur de ressources. Cette classe contiendra:

- Type : `Types` // Une enum à déclarer dans votre classe.
- Amount : `size_t` // La quantitè de la ressource fournie.
- Wrecks : `Droid**` // Un tableau de `Droid*` à recycler, ca fait moderne.

L'enum `Types` (à déclarer dans votre classe `Supply` , je me repète...) doit contenir les valeurs suivantes:

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

Dans le cas où `Type` vaut `Iron` ou `Silicon` (exclusivement), `Amount` représente la quantité qu'apporte le conteneur. Dans le cas où `Type` vaut `Wreck` , `Amount` vaut le nombre de droid présent dans le tableau `Wrecks` . Les Droids sont présents dans le conteneur sur un rack tournant, un peu comme un barillet de revolver. L'opérateur * permet d'accéder à un pointeur sur `Droid` . Donc l'opérateur -> permettra d'accéder directement à ses membres. Pour faire défiler tous les `Droids` présents, vous pouvez faire un `++` (prefix) sur le conteneur, ou `--` (prefix). J'insite, c'est cyclique.

L'accès à la valeur `Amount` se fera grace à une surcharge sur l'opérateur de cast implicite en `size_t` (pensez `const` ... ou pas!). Vous pouvez ajouter un getter classique si vous le souhaitez, mais n'oubliez pas que les autres usines procèdent de cette façon.

Votre classe `Supply` ne doit pas être "copy-constructible". Elle doit pouvoir être construite de deux manières différentes.

- La première en passant le type et la quantité de ressources en paramètre.
- La deuxième, idem que la première, mais avec un `Droid**` en plus représentant les droids à recycler.

L'opérateur ! permet de purger le conteneur, c-a-d qu'il affecte `Amount` à zéro et delete les `Droids` présents dans le conteneur.

L'opérateur == permet de tester quel est le type de ressources que transporte le conteneur. N'oubliez pas de faire !=.

La classe `Supply` à la charge de détruire les `Droid*` qu'elle contient si ceux-ci ne valent pas `NULL` lors de l'appel du destructeur. Evitez que ca arrive, je suis sur que votre mère vous a toujours dit de ne jamais gaspiller.

Voila pour `Supply` , et n'oubliez pas de rendre hommage au grand `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$
```

Chapitre VII

Exercice 5

	Exercice : 05	points : 4
L'usine à robot - Deuxième partie		
Répertoire de rendu: `(DÉPOT SVN - piscine_cpp_d08-promo-login_x)/ex05`		
Compilateur : `g++`	Flags de compilation: `-Wall -Wextra -Werror`	
Makefile : `Non`	Règles : `n/a`	
Fichiers a rendre : `droidfactory.hh, droidfactory.cpp, droid.hh, droid.cpp, supply.hh, supply.cpp, droidmemory.hh, droidmemory.cpp`		
Remarques : `n/a`		
Fonctions Interdites : `Aucune`		

Bon, maintenant que nous avons notre système de ressources, il va être temps de mettre notre première usine sur pied. Donc comme je disais votre classe devra s'appeller `DroidFactory` . Comme toute usine, il lui faut des ressources. Ces ressources seront fournies grace aux conteneurs `Supply` que vous avez écrit plus tôt.

L'approvisonnement se fera via l'opérateur «. Les consignes logistiques interdisent d'utiliser des pointeurs, ca pose trop de problèmes de référencement, à vous de faire avec. Évidemment on peut "chainer" l'approvisionnement en envoyant plusieurs conteneurs à la suite. Évidement le conteneur est vide suite à l'opération.

Chaque conteneur amène la quantité indiquée par son type. Dans le cas d'un recyclage, 80 unités d' `Iron` et 30 de `Silicon` sont extraites par droid. Sans oublier la `BattleData` propre à chaque droid en fonction du ratio (voir plus bas).

Il faut 100 unitéś d' `Iron` et 50 de `Silicon` pour construire un droid. La `BattleData` est distribuée suivant un facteur propre à chaque usine.

L'appel à l'opérateur » permet de produire un nouveau `Droid` et retourne un `Droid*` ou `NULL` si les conditions ne sont pas remplies.

Le ratio est passé en paramètre lors de la construction de l'usine. Il est de type `size_t` et vaut 2 par défaut. Toute construction doit se faire de manière explicite, vous voyez ce

que je veux dire n'est-ce pas ?

`DroidFactory` est canonique mais chut! Ca reste entre nous. Bien évidemment, l'usine créée est parfaitement identique à l'originale.

Évidement, c'est l'usine qui garde ses réserves, mais nul besoin d'y donner accès à qui que ce soit. L'usine affichera son statut sur la sortie standard (Le Grand coût) de cette manière grace à l'opérateur « :

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

Le ratio s'utilise comme tel:

- À chaque création de droid, celui-ci se voit attribuer la quantite d' `Exp` disponible dans l'usine moins cette totalité (d' `Exp` , hein) divisée par le ratio.
- Dans le cas d'un recyclage si le droid qu'on est en train de recycler a une `Exp` supérieure à celle de l'usine, on calcule la différence absolue d' `Exp` entre l'usine et le droid, puis on divise par le ratio. Enfin on l'ajoute à l' `Exp` de l'usine.
- On peut modifier ce ratio grâce aux opérateurs ++ et −−, postfix et prefix (pour les 2).

Ah oui, j'oubliais ! Il est parfois pratique d'utiliser des chemins alternatifs. Donc vous me rajouterez une surcharge sur l'opérateur » pour l'acheminement des conteneurs et une sur « pour la création de droid. Il est exclu de modifier les classes `Supply` et `Droid` . Vous rendrez ces surcharges d'opérateurs dans les fichiers `droidfactory` .hh.cpp puisqu'il font partie de l'interface de `DroidFactory` . Voir le main d'exemple.

```
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$
```

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