POO & C++ : TP 1

EISE4 2014—2015

Pablo Rauzy

rauzy@enst·fr

pablo.rauzy.name/teaching.html#epu-cpp

19 septembre 2014

N'oubliez pas :

- Les TPs doivent être rendus par courriel à `rauzy@enst·fr` au plus tard le lendemain du jour où ils ont lieu avec `[EISE4]` suivi de vos noms dans le sujet du mail.
- Le code rendu doit être propre !
- Le TP doit être rendu dans une archive tar gzippée : `tar czvf NOMS.tgz NOMS` où `NOMS` est le nom du répertoire dans lequel il y a votre code (idéalement, les noms des gens du groupe) ; attention à ne pas rendre un répertoire avec des fichiers temporaires dus à la compilation (pensez à faire un `make clean` avant `tar`).
- Dans le répertoire avec vos noms, chaque exercice doit être dans son propre répertoire.
- Quand un exercice demande des réponses qui ne sont pas du code, vous les mettrez dans un fichier texte `reponses.txt` dans le répertoire de l'exercice.
- Le code est de préférence en anglais, avec des commentaires en français ou anglais, en restant cohérent.
- Le code doit être proprement indenté.
- Respectez les conventions de nommage :
  - variables : `nom_explicite` (*e.g.*, `game_type`),
  - fonctions : `verbeAction` (*e.g.*, `launchNewGame`),
  - classes : `NomExplicite` (*e.g.*, `BoardGame`),
  - attributs : comme variables + `_` (*e.g.* `max_number_of_player_`),
  - méthodes : comme fonctions,
  - accesseurs d'affectation : `set_` + nom de l'attribut sans le `_` final (*e.g.*, `set_max_number_of_player`),
  - accesseurs de consultation : nom de l'attribut sans le `_` final (*e.g.*, `max_number_of_player`),
  - constantes : `NOM_EXPLICITE` (*e.g.*, `VERSION_NUMBER`).
- Pensez à chercher de la documentation par vous-mêmes sur le net !

Dans ce TP :

- Prise en main des outils de développement C++.
- Manipulations simples du langage pour comprendre les notions du premier cours.

Exercice 0.

Récupération des fichiers nécessaires.

1. Pensez à organiser correctement votre espace de travail, par exemple tout ce qui se passe dans ce TP devrait être dans `~/cpp/tp1/`.

2. Récupérez les fichiers nécessaires au TP sur la page du cours. Vous pouvez le faire directement avec la commande `wget http://pablo.rauzy.name/files/mon/cpp/tp1.tgz`.

3. Une fois que vous l'avez extrait de l'archive (`tar xzf tp1.tgz`), renommez le répertoire `tp1` en les noms de votre groupe (par exemple en `Dupont-Dupond`). Si vous ne le faites pas tout de suite, pensez à le faire avant de rendre votre TP.

Exercice 1.

Dans cet exercice, on va apprendre à utiliser le compilateur `g++`. Si vous êtes familier avec `gcc` ça ne devrait vous poser aucun problème puisqu'il utilise la même interface.

1. Vous allez écrire le fichier `hello.cpp` suivant :

```cpp
#include <iostream>
using namespace std;

int main (int argc, char *argv[])
{
  cout << "Coucou, tu veux voir mon C++ ?" << endl;
  return 0;
}
```

→ Compilez le avec la commande `g++ hello.cpp`. Par défaut le fichier binaire (ou exécutable) créé se nomme `a.out`. Vous pouvez nommer le binaire avec l'option `-o` : `g++ hello.cpp -o hello` produira le binaire `hello`.

Lorsque vous utiliserez `g++`, il vous est conseillé de toujours utiliser les options suivantes :

- `-Wall`, qui active tous les avertissements,
- `-Wfatal-errors`, qui arrête la compilation à la première erreur rencontrée (les suivantes seront la plupart du temps inexistantes une fois celle-ci corrigée).

Exercice 2.

Le but de cet exercice est de prendre en main le moteur de production `make`.

1. Si votre programme est composé de plusieurs modules, vous pouvez le compiler en mettant les noms de tous vos fichiers d'implémentation (`.cpp` ou `.cc`) sur la ligne de commande d'appel de `g++`.

→ Compilez le code vu pendant le premier cours. Vous devez créer un fichier binaire qui s'appellera `nuits_de_juin`. Donnez la ligne de commande que vous avez tapé pour cela.

2. Le compilateur va recompiler tout le code qu'on lui donne à chaque fois, et compiler du C++ peut être long.

On peut donc choisir de ne recompiler qu'un seul module par exemple, en utilisant l'option `-c` de `g++`. Avec celle-ci, il s'arrêtera après l'étape de compilation et ne fera pas l'édition des liens. On se retrouve donc avec un *fichier objet* qui pourra être lié avec d'autres fichiers objets, dont un contient une fonction `main`, pour former un binaire.

Par exemple on peut compiler le module `Person` avec la commande :
`g++ -Wall -Wfatal-errors -c person.cpp -o person.o`.

Ensuite, on peut lier les fichiers objets entre eux en rappelant simplement le compilateur avec ces fichiers en argument.

→ Compilez de manière modulaire les trois modules puis créez à nouveau le binaire à partir des trois fichiers objets. Chaque fois, donnez les lignes de commandes utilisées.

Comme vous pouvez le constater, faire cela à la main est fatiguant, et en plus il faut se rappeler de quels modules on a modifié à chaque fois, ce qui n'est plus possible dès que le projet sur lequel on travaille devient un peu important, on simplement si on travail à plusieurs sur un même projet. Il vaut donc mieux utiliser un moteur de production.

3. L'intérêt de l'outil `make` est de ne relancer que les compilations nécessaires de manière automatique. Pour cela, il utilise des informations stockées dans un fichier `Makefile`. Dans ce fichier sont décrites les cibles de compilation et leurs dépendances.

Par exemple dans l'exercice précédent, la cible finale (le binaire) de compilation est `nuits_de_juin`. Cette cible dépend de trois fichiers objets : `person.o`, `poem.o`, et `main.o`.

Ces fichiers objets sont eux-mêmes des cibles (intermédiaires), qui dépendent chacun de leur fichier d'implémentation, et de certains fichiers d'interface (ceux qui sont requis par le module avec la directive `#include`).

Dans le fichier `Makefile` ces informations se notent de la façon suivante :

```
cible: dépendances
<TAB>commande de génération de la cible # notez la tabulation en début de ligne
```

Par exemple :

```
person.o: person.cpp person.hpp
<TAB>g++ -Wall -Wfatal-errors -c person.cpp -o person.o
```

Par défaut, la commande `make` va tenter de créer la première cible du fichier. On peut lui préciser en argument le nom d'une autre cible.

→ Créez un fichier `Makefile` pour compiler nos trois modules.

4. Comme vous l'avez sûrement constaté, c'est très répétitif. Pour éviter cela, on peut définir des variables en leur donnant un nom en majuscules comme `FOO`, et en y faisant référence avec la syntaxe `$(FOO)`.

Il en existe aussi certaines qui sont prédéfinies et utilisables dans les commandes :

- `$@` : la cible,
- `$<` : la première dépendance,
- `$?` : toutes les dépendances plus récentes que la cible,
- `$^` : toutes les dépendances.

Il y en a d'autres mais celles-là devraient vous suffire la plupart du temps.

On peut aussi définir des règles génériques à l'aide du joker `%`. Par exemple `%.o: %.cpp` signifie que chaque fichier objet `.o` dépend du fichier d'implementation `.cpp` correspondant.

On peut donc écrire le `Makefile` suivant :

```
CC = g++
CFLAGS = -Wall -Wfatal-errors
BIN = nuits_de_juin
OBJ = person.o poem.o main.o

$(BIN): $(OBJ)
	$(CC) $^ -o $@

%.o: %.cpp
	$(CC) $(CFLAGS) -c $< -o $@
```

→ Que manque-t-il dans ce `Makefile` qui ne manquait pas dans votre réponse à la question 1 (si elle était juste !) ?

5. Vous pouvez ajouter des dépendances en plusieurs fois pour chaque cible. Par exemple, il est équivalent d'écrire :

```
nuits_de_juin:
	g++ person.o poem.o main.o -o nuits_de_juin
nuits_de_juin: person.o
nuits_de_juin: poem.o
nuits_de_juin: main.o
```

et :

```
nuits_de_juin: person.o poem.o main.o
	g++ person.o poem.o main.o -o nuits_de_juin
```

→ Sachant ça, ajoutez ce qu'il manque dans le `Makefile` de la question précédente.

6. Une bonne habitude est de pouvoir nettoyer son espace de travail. Pour cela une convention est d'utiliser une pseudo-cible `clean` dans le `Makefile`. On appelle cela une pseudo-cible car elle ne crée rien (il n'y aura pas de création d'un fichier `clean`). On indique les pseudo-cibles avec un doublement des `:`.

Par exemple :

```
clean::
	rm -f nuits_de_juin
```

→ Ajoutez une pseudo-cible `clean` dans votre `Makefile`, qui supprime le fichier binaire et les fichiers objets en utilisant les variables que l'ont a définies.

Exercice 3.

Cet exercice a pour but de vous familiariser un peu avec le debugger `gdb`.

1. La première chose à retenir quand on écrit un programme, c'est que le moins de debug on doit faire, le mieux c'est.

Il est donc très important de compiler régulièrement avec tous les avertissements activés et de s'occuper de corriger les problèmes au fur et à mesure que l'on développe le programme.

→ Compilez le fichier `debug.cpp` et exécutez le binaire obtenu. Que se passe-t-il ?

2. Malgré cette bonne habitude, il arrive que l'on ait écrit un bug difficile à trouver. Dans ce cas, on utilise un debugger, dont le rôle est de nous aider à trouver l'origine du problème, qu'on peut ensuite aller corriger.

Le debugger fonctionne directement sur le binaire. Pour qu'il soit capable de vous aider au mieux, en connaissant les noms de vos variables, fonctions, les numéros de lignes dans vos fichiers etc., il faut demander au compilateur de conserver ces informations à destination du debugger dans le binaire. Cela se fait avec l'option `-g`, ou mieux dans le cas spécifique de `g++` (ou `gcc`) et `gdb`, `-ggdb`.

→ Compilez le fichier `debug.cpp` avec l'option de debug. (Comme d'habitude, notez la ligne de commande employée dans `reponses.txt`.)

3. Pour commencer à debugger un programme, il faut invoquer `gdb` avec ce programme comme argument. Par exemple si vous avez créé un binaire `debug` en compilant `debug.cpp` à la question précédente, vous pouvez lancer la commande `gdb debug`.

Une fois dans `gdb`, vous pouvez taper des commandes. La commande `help` vous permet de naviguer dans l'aide, et comme d'habitude n'hésitez pas à chercher de la documentation sur le Web.

La première commande à connaître est `run`, qui permet de lancer l'exécution du programme.

Si une erreur survient, comme ça sera le cas ici, l'exécution du programme est interrompue et vous retombez sur le prompt de `gdb`. Ici, la commande `backtrace` vous sera utile : elle permet d'afficher la pile d'appels de fonctions, ce qui vous permettra souvent de localiser le soucis dans votre code.

Pour quitter `gdb`, utilisez la commande `quit`.

→ À l'aide de `gdb`, trouvez dans quelle fonction il y a un problème. (Comme chaque fois vous reporterez les commandes `gdb` que vous utilisez dans le fichier `reponses.txt`.)

4. → Maintenant, allez voir la fonction qui semble être à l'origine du problème dans le code. Si c'est bien elle, corrigez là. Sinon, votre réponse à la question précédente est fausse.

5. Une fois que le bug est corrigé, recompilez le fichier `debug.cpp` et exécutez le.

→ En comparant la sortie à l'exécution et le code présent dans le `main` du fichier, pensez-vous que ce qui se passe correspond à ce que le programmeur voulait ?

6. Dans `gdb`, on peut aussi ajouter des *points d'arrêt* là où on le souhaite, ce qui aura l'effet d'arrêter l'exécution du programme à ces points d'arrêt et de nous rendre la main sur le prompt de `gdb`, plutôt que d'attendre que le programme plante.

On peut faire cela avec la commande `break` qui prend en argument soit le nom d'une fonction, soit un nom de fichier puis '`:`' puis un numéro de ligne.

Par exemple si on dit `break main` à `gdb`, puis `run`, il va lancer l'exécution du programme jusqu'à entrer dans le `main`, puis vous rendre la main.

→ Relancez une nouvelle session `gdb` pour notre program `debug`, et avant de lancer l'exécution du programme avec `run`, ajoutez un point d'arrêt pour la fonction `Stack::pop`.

7. Quand l'exécution stoppe à un point d'arrêt, plusieurs choix s'offrent à vous.

- Vous pouvez utiliser la commande `continue` pour reprendre l'exécution jusqu'au prochain point d'arrêt ou la fin du programme.
- Vous pouvez utiliser la commande `next` pour avancer d'une instruction (si vous êtes sur un appel de fonction cela avance jusqu'à la fin de celui ci).
- Vous pouvez utiliser la commande `step` pour avancer d'un pas (si vous êtes sur un appel de fonction cela entre dans la fonction).

Et à tout moment vous pouvez afficher les valeurs des variables du programme avec la commande `print` qui prend en argument le nom de la variable dont vous voulez afficher le contenu.

→ À chaque entrée dans `Stack::pop`, utilisez les commandes nécessaires pour afficher la valeur de `length_` avant et après sa décrémentation. Donnez les commandes que vous utilisez en plus des valeurs que vous affichez.

8. → Maintenant que vous en savez assez sur `gdb`, utilisez-le pour trouver la source du problème de la question 5.

9. → Maintenant que vous avez trouvé quel est le problème, essayez d'expliquer ce qui s'est passé, puis corrigez le.

Exercice 4.

Cet exercice a pour but de vous familiariser un peu avec le profileur `valgrind`.

1. `valgrind` est un outil qui va simuler une plate-forme d'exécution pour faire tourner votre binaire dessus, et va au passage récupérer plein d'informations dynamiques sur son exécution.

Il a aussi besoin que les programmes soient compilés avec l'option `-g` de debug. En plus de cela, il aime bien que l'option `-O0`, qui désactive toute tentative d'optimisation du code, soit présente.

→ Après avoir fait un `make clean` vous allez copier les fichiers de l'exercice 2 dans un nouveau répertoire `exo4`, puis vous allez éditer le `Makefile` pour y ajouter les options nécessaires.

2. Si votre binaire s'appelle `foo`, vous pouvez l'analyser avec `valgrind` en lançant la commande :
`valgrind --leak-check=yes foo`.

Comme les erreurs de g++, les messages de `valgrind` sont parfois difficiles à interpréter. N'hésitez pas dans ces cas là à faire des recherches sur le Web, ou à aller voir la documentation du logiciel : `http://valgrind.org/docs/`.

→ Depuis le début, une fuite de mémoire se cache dans le code qui vous a été fourni, trouvez-la à l'aide de `valgrind` avant de la corriger puis de l'expliquer.

Exercice 5.

Maintenant que vous avez vu comment on utilise les outils de développement, il est temps de programmer un peu !

1. Dans cet exercice on va implémenter une classe qui permet de manipuler des piles de nombres aussi grande que l'on veut. Cette classe aura donc parmi ses membres un tableau d'entiers qu'elle devra redimensionner au besoin. Comme c'est une pile, elle aura au moins les actions `push`, qui permet de pousser un entier sur la pile, et `pop`, qui permet de dépiler l'entier présent sur le haut de la pile.

→ Sans implémenter les méthodes, écrivez déjà la déclaration de la classe. Pensez aux attributs dont vous allez avoir besoin, et n'oubliez pas de pratiquer l'encapsulation !

2. On décide que la taille de la pile par défaut est 10, mais aussi qu'on peut vouloir décider de cette taille à l'instanciation de l'objet.

→ Apportez les modifications nécessaires à votre déclaration de classe, et implémenter le ou les constructeurs, ainsi que le destructeur de la classe.

3. Chaque fois qu'il n'y a plus de place dans la pile, on va doubler sa taille, afin de faire un minimum d'allocations. Il n'y a pas de `realloc` en C++, donc il faut faire un autre `new`, puis copier les données.

→ Implémentez la méthode `push` de telle sorte à ce que la pile grandisse automatiquement en cas de besoin.

4. → Implémentez la méthode `pop`. Pensez au(x) cas limite(s).

5. → Rajoutez une méthode `display` qui affiche le contenu de la pile.

6. Maintenant on veut pouvoir trier la pile en appelant une méthode `sort`.

→ Implémentez la méthode `sort` qui trie la pile sur place.

7. → Testez votre code en l'utilisant dans un `main`. Vérifiez que vous n'avez pas de fuite de mémoire avec `valgrind`.

Exercice 6.

Comprendre dans quel ordre les objets sont construit et détruit.

Supposez qu'on a le code suivant :

```
1  #include <iostream>
2  using namespace std;
3
4  class A {
5  public:
6    A () { cout << "Construction A" << endl; }
7    A (const A &a) { cout << "Construction par copie A" << endl; }
8    ~A () { cout << "Destruction A" << endl; }
9  };
10
11 class B {
12 public:
13   B () { cout << "Construction B" << endl; }
14   B (const B &b) : a_(b.a_) { cout << "Construction par copie B" << endl; }
15   B (A a) : a_(a) { cout << "Construction B(a)" << endl; }
16   B (char c, A a) { a_ = a; cout << "Construction B(=a)" << endl; }
17   B (int n, A &a) : a_(a) { cout << "Construction B(&a)" << endl; }
18   ~B () { cout << "Destruction B" << endl; }
19 private:
20   A a_;
21 };
22
23 class C {
24 public:
25   C () { cout << "Construction C" << endl; }
26   C (const C &c) : b_(c.b_) { cout << "Construction par copie C" << endl; }
27   C (A a) : b_(a) { cout << "Construction C(a)" << endl; }
28   C (int n, A &a) : b_(0, a) { cout << "Construction C(&a)" << endl; }
29   ~C () { cout << "Destruction C" << endl; }
30 private:
31   B b_;
32 };
```

1. → Qu'affiche le programme `A a;` ?

2. → Qu'affiche le programme `B b;` ?

3. → Qu'affiche le programme `C c;` ?

4. → Qu'affiche le programme `A a; B b(a);` ?

5. → Qu'affiche le programme `A a; B b(' ', a);` ?

6. → Qu'affiche le programme `A a; A &a_ref = a; B b(0, a_ref);` ?

7. → Qu'affiche le programme `A a; C c(a);` ?

8. → Qu'affiche le programme `A a; A &a_ref = a; C c(0, a_ref);` ?