ÉCOLE POLYTECHNIQUE FÉDÉRALE DE LAUSANNE
EIDGENÖSSISCHE TECHNISCHE HOCHSCHULE – LAUSANNE
POLITECNICO FEDERALE – LOSANNA
SWISS FEDERAL INSTITUTE OF TECHNOLOGY – LAUSANNE

Faculté Informatique et Communications
Cours de Programmation Orientée Système, sections MA et PH
Chappelier J.-C.

Programmation Orientée Objet (SMA/SPH) : Correction examen final

2 juin 2022

INSTRUCTIONS (à lire attentivement)

IMPORTANT ! Veuillez suivre les instructions suivantes à la lettre sous peine de voir votre examen annulé dans le cas contraire.

1. Vous disposez d'une heure quarante-cinq minutes pour faire cet examen (9h15 – 11h00).
2. Vous devez **écrire à l'encre noire ou bleu foncée**, pas de crayon ni d'autre couleur.
   N'utilisez **pas non plus de stylo effaçable** (perte de l'information à la chaleur).
3. Vous avez droit à toute documentation papier.
   En revanche, vous ne pouvez pas utiliser d'ordinateur personnel, ni de téléphone portable, ni aucun autre matériel électronique.
4. Répondez aux questions directement sur la donnée ; ne joignez aucune feuille supplémentaire ; **seul ce document sera corrigé**.
   Vous disposez, si nécessaire, d'une page blanche supplémentaire en fin de sujet.
5. Lisez attentivement et *complètement* les questions de façon à ne faire que ce qui vous est demandé. Si l'énoncé ne vous paraît pas clair, ou si vous avez un doute, demandez des précisions à l'un des assistants.
6. L'examen comporte quatre exercices indépendants (sur 13 pages), qui peuvent être traités dans n'importe quel ordre, mais qui ne rapportent pas la même chose (les points sont indiqués, le total est de 115 points) ; tous les exercices comptent pour la note finale :
   — questions « courtes » : 24 points ;
   — conception : 35 points ;
   — trouver les erreurs : 26 points ; et
   — déroulement de programmes : 30 points.

Question 1 – Questions de cours [24 points]

1.1 La classe ! [2 points]

Considérez le code ci-dessous :

```
1   class C {
2    public:
3     C() {}
4     C(double a = 7.1) : c(a) {}
5    protected:
6     double c;
7   };
8
9   int main() {
10    C mon_c(3.14);
11    return 0;
12  }
```

Entourer la ou les option(s) ci-dessous qui sont correcte(s). (pénalités pour réponses fausses entourées.)

1. Le code ne compile pas.
2. Le code compile, mais produit un erreur si on l'exécute.
3. Le code compile et, à la ligne 10, le constructeur `C::C(double)` est appelé.
4. Le code compile et, à la ligne 10, le constructeur par défaut est appelé.

Justifiez brièvement votre réponse : La bonne réponse est a).

En effet, le constructeur `C(double)` ayant une valeur par défaut à son paramètre, fournit aussi un constructeur par défaut. Ce qui crée une ambiguïté avec celui de la ligne 3.

1.2 Ayants droit [5 points]

Considérez une classe C++. Dans quelles conditions décrites ci-dessous une méthode/fonction peut elle accéder aux méthodes privées/protégées/publiques de cette classe ? Mettez un plus (+) si l'accès est possible et un moins (-) si c'est interdit. Laissez blanc si vous ne savez pas (pénalité pour réponse fausse).

	peut accéder à une méthode :		
	`private`	`protected`	`public`
un opérateur surchargé en externe	-	-	+
un opérateur surchargé en interne	+	+	+
une méthode de la même classe	+	+	+
une méthode d'une autre classe quelconque	-	-	+
fonction quelconque	-	-	+
une fonction `friend` de la classe	+	+	+
une méthode d'une sous-classe, accès direct	-	+	+
une méthode d'une sous-classe, accès au travers d'un argument	-	-	+

1.3 Les losanges sont éternels [7 points]

Considérez les classes ci-dessous :

```
class X {
public:
  virtual double g() = 0;
};

class Y : public X {
public:
  double g() { ... }
};

class Z : public X {
 public:
  double g() { ... }
};
```

Quel problème a-t-on si l'on ajoute une quatrième classe, disons `A`, qui hérite de `Y` *et* de `Z` ?

`A` hériterait la méthode `g()` des deux classes, ce qui conduit à une ambiguïté.

Proposez des solutions *sans* modifier le code founi, et discutez leurs avantages et inconvénients.

Solutions :

- Une solution serait d'expliciter, à *chaque* appel, quelle méthode `g()` appeler en utilisant l'opérateur de résolution de portée : `Y::g()` ou `Z::g()`.
- Une autre solution consiste à désambiguiser (dans la définition de la classe `A`) avec la directive `using` : p.ex. `using Y::g`.
- Une troisième solution est de redéfinir (« *override* ») `g()` dans `A`.

Discussion :

- La première solution n'est pas bonne car elle laisse la décision au programmeur utilisateur de la classe (en gros le problème n'est pas résolu).
- La seconde n'est pas mal si c'est vraiment le sens que l'on veut donner, mais à l'inconvénient ci-dessous (si `X` classe virtuelle).
- La 3^{e} solution me semble la meilleure car elle fixe clairement la décision de quoi faire pour `g()` dans `A`.

Est-ce que si `X` était une classe *virtuelle* cela changerait votre réponse ? Si oui, comment/en quoi ?

Le fait d'avoir un héritage virtuel fait que `X::g()` devient unique dans `A`. Cette même fonction se retrouve donc avec **deux** redéfinitions terminales (celle de `Y` et celle `Z`) si on n'ajoute pas de redéfinition dans `A` (c.-à-d. si on ne fait pas las 3^{e} solution ci-dessus). Or il n'est pas permit par la norme d'avoir plusieurs redéfinitions terminales (« *final override* ») d'une fonction virtuelle.

Bref, en cas d'héritage virtuel, seule la 3^{e} solution est possible.

1.4 Concepts [10 points]

Expliquez clairement en *une* phrase les concepts suivants et donnez *un* exemple illustratif *simple* mais *pertinent* en code C++ :

1. classe virtuelle

   Une classe virtuelle, à ne pas confondre avec une classe abstraite (!!), est une classe dont on dérive virtuellement (p.ex. en cas d'héritage multiple).

```
class Ovipare : public virtual Animal ...
```

2. attribut de classe
   Un attribut de classe est un attribut qui existe, dans la portée de la classe, indépendemment de toute instance. Il se déclare avec le mot clé `static`.

   ```
   class Employe { ... private: static double taux_legal; };
   ```

   (tou(te)s les employé(e)s ont le même « taux légal » de je-ne-sais-quoi, qui est fixé (par la loi) indépendemment d'elles/eux.)
3. méthode constante
   Une méthode qui ne modifie pas l'instance (on parle aussi d'*accesseur*).

   ```
   double Cercle::surface() const;
   ```

4. méthode virtuelle pure
   Une méthode virtuelle pure est une méthode virtuelle (polymorphisme) ayant « `= 0` » à la fin de son prototype.
   Cela rend la classe *abstraite*, c.-à-d. non instanciable.
   Notez que, même si c'est rarement le cas, une méthode virtuelle pure *peut* être définie dans la classe abstraite elle-même. Cependant, même si ce point est explicitement cité dans les transparents, je n'ai pas trop insisté là dessus en cours. On considèrera donc quand même comme correct ici si les étudiant(e)s disent « qui n'est pas définie ».
   Exemple :
   ```
   class Figure { virtual double surface() const = 0; }
   ```

5. `this`
   `this` est un pointeur vers l'instance courante.
   C'est « chez moi », « mon adresse à moi ».

   ```
   void A::set_attribute(type x) { this->value = x; }
   ```

Question 2 Conception OO et programmation [35 points]

Un fleuriste souhaite pouvoir gérer la composition des bouquets qu'il peut proposer à sa clientèle.

Les bouquets sont composés de fleurs et de décorations (p.ex. tiges, branches, feuilles, tissus, ...). On parle ici de prestation (bouquets) et de « modèles de fleurs présentes en catalogue », pas des vrais bouquets effectivement faits avec les vraies fleurs. Par exemple, la « fleur » « `tulipe` » désigne ci-dessous un type de fleurs présent dans le catalogue du fleuriste et non pas une tulipe concrète.

Tous les composants d'un bouquet sont caractérisés par :
- leur nom (une chaîne de caractères, unique pour chaque composant) ;
- leur couleur ;
- leur prix à l'unité (un `double`).

Les décoration ont en plus une rigidité (disons : soit souple, soit solide) et les fleurs une odeur (disons : parmi 8 odeurs codifiées).

Le fleuriste souhaite gérer son stock de composants (fleur ou décoration) comme un ensemble de paires (composant, quantité en stock) ; p.ex. : `(rose, 45), (tige de bambou, 23), (tulipes, 85)`.

Par ailleurs, un bouquet est caractérisé par :
- un nom (une chaîne de caractères, unique pour chaque bouquet) ;
- un ensemble de paires (composant, quantité de ce composant présente dans le bouquet) ;
  p.ex. : `(tulipe, 5), (rose, 3), (tige de bambou, 1)` ;
  pour simplifier, on supposera que tous les composants possibles existent plus longtemps que tous les bouquets (les composants sont présents en catalogue plus longtemps que les bouquets ; ils restent présents en catalogue même si la quantité en stock est 0) ;
- une marge (`double`), représentant le prix à ajouter à la somme des prix de base des composants pour avoir le prix total du bouquet.

Le fleuriste souhaite se doter d'un programme permettant au moins :

1. de débiter/augmenter d'une quantité donnée le stock d'un composant donné (par exemple : augmenter de 150 le nombre de tulipes en stock) ;
2. de modifier la quantité d'un composant donné (fleur ou décoration) dans un bouquet donné ;
3. d'afficher l'état détaillé du stock (noms et quantités disponibles) ;
4. de calculer (pas afficher !) le prix d'un bouquet (somme des prix des composants plus la marge) ;
5. de tester si les quantités en stock sont suffisantes pour un nombre donné de bouquets (identiques) ; par exemple de tester si l'on peut faire 7 bouquets qui s'appellent « réveil enchanté » ;
6. de composer un bouquet donné : cette fonction doit tester si les stock sont suffisants pour faire le bouquet, et, si oui, elle doit débiter le stock du nombre de composants nécessaire à ce bouquet.

   Par exemple si la fonction compose un bouquet constitué de 5 tulipes, 3 roses et une tige de bambou, il doit diminuer le stock de 5 tulipes, 3 roses et une tige de bambou et retourner `true`, ou afficher un message et retourner `false` si le stock est insuffisant pour composer le bouquet.

2.1 Conception (22 points)

Sans donner tout le détail du code complet (on ne demande ici qu'une *conception*), écrivez **en C++** les classes, les éventuelles relations d'héritage, les attributs et les méthodes des classes, les droits d'accès et les éventuelles fonctions (externes) que vous utiliseriez pour implémenter un tel programme.

Précisez les types des attributs et les prototypes des méthodes/fonctions, mais ne donnez pas leur définition (on répète : il s'agit ici de la partie *conception*, pas de l'implémentation ; c.-à-d. les prototypes, pas les définitions des méthodes).

Lorsque c'est nécessaire, indiquez également les constructeurs et les destructeurs (sans leur définition).

Veuille en particulier donner explicitement les *prototypes*[1] des fonctions/méthodes nécessaires aux six fonctionnalités désirées par le fleuriste.
Dites à chaque fois clairement si c'est une fonction ou une méthode (et de quelle classe).

Voici *une* réponse possible :

Types de base :

```
// whatever... (pas explicitement demandé)
enum Odeur    { A, B, C, D, E, F, G, H };
enum Rigidite { souple, rigide };
enum Couleur { whatever };

// -------------------------------------------------------------
class Composant {
public:
  // + constructeur(s) et méthodes get
  // pas forcément de set() (ici j'ai tout mis const,
  //  mais c'est optionnel)
private:
  const string nom;
  const Couleur c; // ou string
  const double prix;
};

// -------------------------------------------------------------
class Fleur : public Composant {
public:
  // + constructeur(s) et méthodes get
  // pas forcément de set() (cont optionnel)
private:
  const Odeur odeur;
};

// class Fleur
class Decoration : public Composant {
public:
  // + constructeur(s) et méthodes get
  // pas forcément de set() (cont optionnel)
private:
  const Rigidite rigidite;
};
```

La classe `Composant` peut être abstraite (via, p.ex., un destructeur virtuel pur (mais défini !!)).

suite au dos ☞

1. **pas** les définitions

Collections :

```
class CollectionComposants {
public:
  // regroupe les fonctionnalités demandées 1. et 2.
  void change_quantite(const string& nom, int difference);

  // + methodes get/set commme par exemple :
  void ajoute_composant(Composant& c, // ou Composant*, au choix
                        unsigned int quantite);
  unsigned int get_quantite(const string& nom) const;

  /* + des moyens de parcourir la collection :
   *  soit size(), acces() ou []
   *  soit itérateurs
   */

private:
  map<Composant*, int> collection;
 // ou vector<pair<Composant*, int>>
 // ou définir sa propre classe Paire...
};

// ========================================================================
class Bouquet : public CollectionComposants {
public:
  // + constructeur(s) et méthodes get/set
  double prix() const;  // fonctionnalité demandée 4.
private:
  const string code;
  double marge;
};

// ========================================================================
class Stock : public CollectionComposants {
public:
  void affiche() const; // fonctionnalité demandée 3.

 // fonctionnalité demandée 5.
  bool en_stock(const Bouquet& b, unsigned int nombre) const;

 // fonctionnalité demandée 6.
  bool compose_bouquet(const Bouquet& b);
};
```

J'ai mis tous les attributs en `private`, mais on peut les tolérer en `protected`.

Pour toutes les classes : s'il y a des attributs, il faut le constructeur pour les initialiser ; s'il n'y a pas d'attribut (ou si l'attribut a un constructeur par défaut qui suffit), le constructeur par défaut par défaut suffit (sauf pour les sous-classes dont la super-classe a un constructeur spécifique, mais il n'y en a pas ici).

2.2 Programmation (13 points)

Écrire (c.-à-d. donner les définitions) des méthodes/fonctions demandées par le fleuriste aux points 2, 4 et 6 de la page 5 :

2. modifier la quantité d'un composant donné (fleur ou décoration) dans un bouquet donné ;
4. calculer (pas afficher !) le prix d'un bouquet (somme des prix des composants plus la marge) ;
6. composer un bouquet donné : cette fonction doit tester si les stock sont suffisants pour faire le bouquet, et, si oui, elle doit débiter le stock du nombre de composants nécessaire à ce bouquet. Par exemple, si la fonction compose un bouquet constitué de 5 tulipes, 3 roses et 1 tige de bambou, il doit diminuer le stock de 5 tulipes, 3 roses et 1 tige de bambou et retourner `true`, ou afficher un message et retourner `false` si le stock est insuffisant pour composer le bouquet.

Voici *une* réponse possible. Il existe bien sûr plusieurs solutions en fonction de la conception, des méthodes supposées exister. Ce qui compte c'est que ce soit cohérent.

Fonctionnalité 2. :

```
void CollectionComposants::change_quantite(const string& nom, int difference) {
  // cherche le composant -- recherche linéaire
  for (size_t i(0); i < collection.size(); ++i) {
    if (nom == collection[i].first->get_nom()
        // ou get_nom(i), en fonction des accesseurs implémentés
      ) {
      // trouvé !
      collection[i].second += difference;
      // ou : change_quantite(i, difference); si existe (typiquement privée !!)
    }
  }
}
```

Fonctionnalité 4. :

```
double Bouquet::prix() const {
  double prix(0.0);
  // add price of material
  for (size_t i = 0; i < size(); ++i) // par exemple, ou toute itération sur le bouquet
  {
    prix += collection[i].first->get_prix() * collection[i].second;
    // par exemple, ou :
    //  prix += get_prix(i) * get_quantite(i);
    // ce qui compte c'est la cohérence des accès
  }
  // ajoute la marge
  prix += marge;
  return prix;
}
```

Fonctionnalité 6. :

```
bool Stock::compose_bouquet(const Bouquet & b) {
  // vérifie qu'il y en a assez (via une méthode, ou sinon l'écrire ici ;
  //   MAIS il faut vérifier tous les composants AVANT de modifier le stock !)
  if (en_stock(b, 1)) {
    // supprime les composants du stock
    for (int i = 0; i < b.size(); ++i) {
      change_quantite(b[i]->get_nom(),
                      // ou b.get_nom(i), ou b.get_composant(i)-> get_nom(),
                      // par exemples...

                      - b.get_quantite(b[i]->get_nom())
                      // ou - b.get_quantite(i), par exemple...
                      );
    }
    return true;
  }
  return false;
}
```

L'idée est bien sûr ici d'utiliser la 5e fonctionnalité (`en_stock()`).

On peut par ailleurs avoir une conception un peu différente et passer une `CollectionComposants` et un `Bouquet` par référence, et considérer que cette fonction « crée » (affecte) également le `Bouquet` (au lieu de considérer qu'il a déjà été construit et passé, comme dans la solution proposée ci-dessus).

Question 3 – Trouver les erreurs [26 points]

Le programme fourni ci-dessous contient huit erreurs, dont six sont détectées par le compilateur, une lors de l'édition de liens et une se voit lors de l'exécution du programme.

On vous demande de corriger ces erreurs en les marquant directement sur le code ci-dessous, ou juste en dessous.

Attention ! On ôtera 2 points pour toute indication d'une erreur qui n'en est pas une.

Afin de facilité votre réponse/lecture, nous fournissons à nouveau le même code en page **??**). Vous pouvez aussi répondre là-bas, ainsi que donner des explications complémentaires en bas de page **??**.

Voici le code corrigé :

```
1   #include <iostream>
2   #include <cmath>
3   using namespace std;
4
5   class A {
6   public:
7     void f(double a) const
8     { cout << a + 7.1 << endl; }
9   };
10
11  class B {
12  public:
13    B(double x = 3.14) : b(x) {}
14    B(const B& b2) : b(b2.get_b()) {
15      cout << "Copie d'un B(" << b
16           << ")." << endl;
17    }                          ②
18    double get_b() const { return b; }
19    void set_b(double x) { b = x; }
20    void f() const {
21      cout << 12.1 + 2.2 * b * b << endl;
22    }                                  ⑦
23    virtual void print() const = 0;
24  protected:
25    double b;
26  };    ①
27
28  class C : public A, public B {
29  public:
30    C() = default;          ⑧
31    C(const C& c2) : B(c2) {
32      cout << "Copie d'un C : "
33           << c2.get_b() << " -> "
34           << b << endl;
35    }
```

```
                                   ③
36    void increment() { ++b; }
37    void print() const {
38      cout << "je suis un C(" << get_b()
39           << ")" << endl;
40    }
                                   ⑤
41    void print_equal(B const& b2) const {
42      if (abs(b - b2.get_b()) < 1e-10)④{
43        cout << "OK, pareils." << endl;
44      } else {
45        cout << "Différents !" << endl;
46      }
47    }
48  };
49
50  int main() {
51    C un_c;
52    un_c.set_b( 2.71 );
53
54    C un_autre_c(un_c);
55    un_autre_c.print();
56    un_autre_c.print_equal(un_c);
57
58    un_autre_c.increment();
59    un_autre_c.print();
60                    ⑥
61    un_autre_c.A::f( 44.44 );
62
63    return 0;
64  }
```

Pour les erreurs de compilation (dans l'ordre détectées par le compilateur) :

① **l.26 :** il manque le `;` à la fin de la classe `B`

② **l.18 :** `B::get_b()` n'est pas `const` : ajouter `const`

③ **l.36 :** `C::increment()` est tout sauf `const` !

④ deux erreurs à la **l.42 :** tout d'abord il faut un `==` au lieu d'un `=` — en fait il faudrait comparer avec la valeur absolue par rapport à un « epsilon » ;

⑤ ...et ensuite il faut passer par un accesseur pour accédér à `b2.b`

⑥ **l.61 :** `f()` est ambigüe, il faut désambiguiser (soit avec un opérateur de résolution de portée, p.ex. `B::`, soit avec un `using`, cf question 1.3, page 3).

⑦ L'erreur d'édition de liens vient du fait que la méthode virtuelle `B::print()` n'est pas définie (**l.23**). Il suffit de le faire ou alors de la rendre virtuelle pure (en ajoutant `= 0`) (les *deux* solution sont valides, au choix).

⑧ L'erreur de déroulement qui reste est liée au fait que le constructeur de copie de `C` n'appelle pas le constructeur explicitement le constructeur de copie de `B`. C'est donc le constructeur *par défaut* de `B` qui est appelé ici. La correction consiste à explicitement mettre l'appel au constructeur de copie de `B` dans la « section deux-points » du constructeur de copie de `C` (**l.31**).

suite au dos ☞

Question 4 – Exécution de programme [30 points]

Le programme suivant compile et s'exécute sans erreurs. Qu'affiche-t-il ?
Répondez-ci dessous puis justifiez (page ci-contre) votre réponse en expliquant les points *importants* (on ne vous demande pas ici de paraphraser le code, mais bien de montrer que vous avez compris ce qui se passe !).

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

class A {
public:
  A(int x = 2, int y = 3) : a(x), b(y) {}
  int f1() const { return 4*a; }
virtual int f2() const { return 5*b; }
protected:
  int a;
  int b;
};

class B : virtual public A {
public:
  B(int x = 6, int y = 7, int z = 8)
   : A(x,y), c(z) {}
  int f1() const { return  9*a; }
  int f2() const { return  b + 2*c; }
private:
  int c;
};

class C : virtual public A {
public:
  C(int x = 1, int y = 5) : A(x,y) {}
  int f1() const { return 7*a; }
virtual int f2() const { return 11*b; }
};

class D : public B, public C {
public:
  int f1() const { return B::f1(); }
  int f2() const { return C::f2(); }
};
```

```
int main() {
  A a(3, 5);
  A* pa(&a);
  cout << "A) " << a.f1()   << endl;
  cout << "B) " << a.f2()   << endl;
  cout << "C) " << pa->f1() << endl;
  cout << "D) " << pa->f2() << endl;
  B b;
  pa = &b;
  cout << "E) " << b.f1()   << endl;
  cout << "F) " << b.f2()   << endl;
  cout << "G) " << pa->f1() << endl;
  cout << "H) " << pa->f2() << endl;
  C c;
  pa = &c;
  cout << "I) " << c.f1()   << endl;
  cout << "J) " << c.f2()   << endl;
  cout << "K) " << pa->f1() << endl;
  cout << "L) " << pa->f2() << endl;
  D d;
  pa = &d;
  B* pb(&d);
  cout << "M) " << d.f1()   << endl;
  cout << "N) " << d.f2()   << endl;
  cout << "O) " << pa->f1() << endl;
  cout << "P) " << pa->f2() << endl;
  cout << "Q) " << pb->f1() << endl;
  cout << "R) " << pb->f2() << endl;
  a = d;
  pa = &a;
  cout << "S) " << a.f1()   << endl;
  cout << "T) " << a.f2()   << endl;
  cout << "U) " << pa->f1() << endl;
  cout << "V) " << pa->f2() << endl;
  return 0;
}
```

Réponses :

A) 12
B) 25
C) 12
D) 25

E) 54
F) 23
G) 24
H) 23

I) 7
J) 55
K) 4
L) 55

M) 18
N) 33
O) 8
P) 33
Q) 18
R) 33

S) 8
T) 15
U) 8
V) 15

Explications :

Pour `A)` à `D)` (classe `A`), il suffit de dire :

① les valeurs initiales sont `3` et `5` (appel explicite au constructeur)

② appel simple de `A::f1()` et `f2()` car c'est la classe de plus haut niveau.

Pour `E)` à `H)` (classe `B`), il faut dire :

⑥ les valeurs sont `6`, `7` et `8` (appel au constructeur par défaut de `B`)

⑦ appel de `B::f1()` (donne $54 = 6 \times 9$) dans le premier cas et `A::f1()` (donne $24 = 4 \times 6$) dans le second cas (pointeur) car *non* virtuelle dans `A`.

⑧ appel de `B::f2()` dans les deux cas (donne $23 = 7 + 2 \times 8$) car *virtuelle* (polymorphisme sur le second appel).

Pour `I)` à `L)` (classe `C`), il faut dire (comme pour la classe `B`) :

③ les valeurs sont `1` et `5` (appel au constructeur par défaut de `C`)

④ appel de `C::f1()` (donne $7 = 7 \times 1$) dans le premier cas et `A::f1()` (donne $4 = 4 \times 1$) dans le second cas (pointeur) car *non* virtuelle dans `A`.

⑤ appel de `C::f2()` dans les deux cas (donne $55 = 11 \times 5$) car *virtuelle*

Pour `M)` à `V)` (classe `D`), il faut dire :

⑨ les valeurs sont `2` et `3` (appel au constructeur par défaut de `A` car classe virtuelle) et `8` pour `z` (constructeur par défaut de `B`)

⑩ appel de `D::f1()` (= `B::f1()` qui donne $18 = 9 \times 2$) dans le premier cas et `A::f1()` (donne $8 = 2 \times 4$) dans le second cas (pointeur) car *non* virtuelle dans `A`.

❶ appel de `D::f2()`, c-à-d `C::f2()`, dans les deux cas (donne $33 = 1 \times 3$) car *virtuelle*

❷ pour `pb` (`Q)`) c'est `B::f1()` (18) directement, car non virtuelle, et `D::f2()` (33), car polymorphisme, qui sont appelées

❸ et pour `S)` à `V)`, c'est un `A` qui est manipulé, donc quatre fois les méthodes de `A` ($8 = 4 \times 2$ et $15 = 5 \times 3$).