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 Objet, sections MA et PH
Chappelier J.-C.

Programmation Orientée Objet (SMA/SPH) :

Correction de l'examen final

30 mai 2024

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(e) des assistant(e)s.
6. L'examen comporte quatre exercices indépendants, 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 120) :
   1. questions de cours : 14 points ;
   2. correction de code : 34 points ; et
   3. conception : 40 points ;
   4. déroulement de programme : 32 points.

   Tous les exercices comptent pour la note finale.

Question 1 – Les bases [14 points]

1.1 Définition [2 points]

Expliquez clairement et brièvement ce qu'est une classe *virtuelle* et donnez un exemple illustratif simple mais pertinent en code C++ :

Une classe virtuelle est une classe dont on hérite virtuellement, pour éviter des duplications lors d'héritage multiple.

Par exemple :

```
class Personnage;
class Magicien : public virtual Personnage;
class Guerrier : public virtual Personnage;
class GuerrierMagicien : public Guerrier, public Magicien;
```

Ainsi le `GuerrierMagicien` n'est qu'*un seul* `Personnage` (et non pas deux).

1.2 Destruction [1.5 point]

Quel destructeur une classe *abstraite* devrait elle avoir ?

(a) aucun (b) un destructeur de copie (c) virtuel (d) abstrait (e) celui par défaut

Justifiez *brièvement* votre réponse :

Le but d'une classe abstraite est d'être utilisée de façon polymorphique. Dans ces cas, il est nécessaire que son destructeur soit *virtuel* (**réponse c**) pour permettre l'appel du destructeur de ses sous-classes (afin de permettre, si nécessaire, qu'elles libèrent leurs ressources).

1.3 Appels à la pelle [2.5 points]

Dans la définition d'une méthode `m()` d'une sous-classe `Y`, quelle est la syntaxe pour faire appel à une méthode de la super-classe `X` ?

Explicitez les différents cas concernant la méthode de la classe `X` qui est appelée (p.ex. si c'est `m()` ou une autre méthode, ou suivant les droits d'accès).

Si c'est la même méthode (`m()`), il faut la démasquer avec l'opérateur de résolution de portée : `X::m()`, sinon il suffit simplement de l'appeler directement (héritée).

L'accès à une telle méthode est permise ou non en fonctions des droits d'accès (de départ, *indépendemment* de la nature `public`, `protected` ou `private` de l'héritage) : possible si `public` ou `protected`, interdit si `private`.

1.4 Taxonomie (ou taxidermie ?) [8 points]

On a défini les classes et prototypé les fonctions suivantes :

```
class Mammifere {
  // ... ici toute la définition de la classe, dont
  virtual void se_deplace() = 0;
};

class Renard : public Mammifere {
  // ... ici toute la définition de la classe, dont celle de
  virtual void se_deplace() override;
};

void f(Renard& c);
void g(Renard* c);
void h(Mammifere& m);
void k(Mammifere* m);
```

Dites si les lignes suivantes compilent et fonctionnent correctement (quand elles compilent). Justifiez *brièvement* votre réponse.

	compile ? (oui/non)	fonctionne ? (oui/non)	justification :
`f(new Renard);`	non	– (ou non)	L'argument de `f()` doit être une variable, pas un pointeur.
`f(new Mammifere);`	non	– (ou non)	idem (ou aussi parce que `Mammifere` est une classe abstraite ; on ne peut pas en créer d'instance)
`f(*new Renard);`	oui	oui/non	Ca fonctionne **MAIS** impossibilité de libérer la mémoire (« *memory leak* »). La réponse oui ou non doit donc être justifiée !
`f(*new Mammifere);`	non	– (ou non)	`f()` attend une instance de `Renard`, pas de `Mammifere` (et on ne peut pas créer d'instance de `Mammifere`)
`g(new Renard);`	oui	oui	(ici c'est «oui» car `g()` pourrait libérer la mémoire)
`g(new Mammifere);`	non	– (ou non)	`g()` attend un pointeur sur `Renard`, pas sur `Mammifere` (et on ne peut pas créer d'instance de `Mammifere`)
`g(*new Renard);`	non	– (ou non)	g a besoin d'un pointeur pas d'une valeur
`g(*new Mammifere);`	non	– (ou non)	idem (ou aussi parce que `Mammifere` est une classe abstraite)

	compile ? (oui/non)	fonctionne ? (oui/non)	justification :
`h(new Renard);`	non	– (ou non)	l'argument de `h` doit être une variable, pas un pointeur.
`h(new Mammifere);`	non	– (ou non)	idem (ou aussi parce que `Mammifere` est une classe abstraite)
`h(*new Renard);`	oui	oui/non	oui c'est possible car un `Renard` est un `Mammifere`, mais par contre on a une fuite de mémoire comme pour `f()` plus haut
`h(*new Mammifere);`	non	non	`Mammifere` est une classe abstraite ; on ne peut pas en créer d'instance
`k(new Renard);`	oui	oui	un `Renard` est un `Mammifere`
`k(new Mammifere);`	non	non	`Mammifere` est une classe abstraite ; on ne peut pas en créer d'instance
`k(*new Renard);`	non	– (ou non)	poiteur, pas valeur
`k(*new Mammifere);`	non	– (ou non)	idem (ou aussi parce que `Mammifere` est une classe abstraite)

suite au dos ☞

Question 2 – Ça cloche [34 points]

Le programme fourni ci-contre contient plusieurs erreurs, dont certaines sont détectées par le compilateur (messages ci-dessous), d'autres pas et dont une pourrait causer un arrêt prématuré lors de l'exécution du programme.

Indiquez toutes les erreurs, en donnant à chaque fois le numéro de ligne correspondant (ou en l'indiquant directement sur le code ci-contre) et une **explication** de l'erreur. Si cela fait sens, proposez également une correction.
Attention ! On ôtera des points pour toute indication d'une erreur qui n'en est pas une.

Voici les messages d'erreurs du compilateur :

```
debug.cc:17:9: error: conflicting return type specified for
    'virtual int B::f2(std::ostream&) const'
debug.cc:8:18: error:   overriding 'virtual void A::f2(std::ostream&) const'
debug.cc: In function 'int main()':
debug.cc:34:5: error: cannot declare variable 'a' to be of abstract type 'A'
debug.cc:4:7: note:   because the following virtual functions are pure within 'A':
debug.cc:7:18: note: virtual void A::f1() const
debug.cc: In constructor 'B::B()':
debug.cc:13:7: error: no matching function for call to 'A::A()'
debug.cc:13:7: note: candidates are:
debug.cc:6:5: note: A::A(int)
debug.cc:6:5: note:   candidate expects 1 argument, 0 provided
debug.cc:4:7: note: A::A(const A&)
debug.cc:4:7: note:   candidate expects 1 argument, 0 provided
debug.cc:37:8: error: could not convert '* c' from 'A' to 'B'
debug.cc:38:16: error: invalid operands of types 'void' and
    '<unresolved overloaded function type>' to binary 'operator<<'
debug.cc:42:6: error: request for member 'f3' in 'b2', which is of non-class type 'B()'
debug.cc:43:5: error: request for member 'f2' in 'c',
    which is of pointer type 'A*' (maybe you meant to use '->' ?)
debug.cc:14:10: error: 'void B::f3()' is private
debug.cc:44:8: error: within this context
```

Il y a 11 erreurs en tout : voir le code commenté plus loin. Il y a plusieurs erreurs qui peuvent être indiquées différemment à différents endroits, mais qui restent la même erreur.

1. ligne 14 ou 44 : `B::f3()` est `private` ; la mettre en `public` ;
2. ligne 17 : `B::f2()` n'a pas le bon type de retour : il faudrait mettre `void` et supprimer le `return 0;` ;
3. ligne 20 (ou 38) : l'opérateur d'affichage pour `B` ne retourne rien ; on ne peut donc pas faire l'appel « `<< endl` » après « `cout << b` » ;
4. ligne 25 : oubli du `const` dans `C::f2()`, est-ce volontaire ou une erreur ? ? en tout cas cela introduit un *autre* `f2()` et doit être signalé ;
5. ligne 34 : `A` n'est pas instanciable (classe abstraite) ; supprimer cette ligne inutile (ou supprimer l'abstraction de `A`...) ;
6. ligne 35 : `B` n'a pas de constructeur par défaut (soit en ajouter un, soit corriger l'instanciation de `b`) ;

7. ligne 37 : `*c` est de type `A`, pas `B`

   il faudrait :
   — soit utiliser le polymorphisme des *méthodes* `f2()` et appeler `c->f2(cout);`
   — soit changer le type de `c` pour un `B` :
   ```
   B* c = new B();
   f2(*c);
   ```
   — soit changer le type de `c` pour un `C` et changer le type de la *fonction* `f2` (ligne 31) :
   ```
   void f2(C c)...
   C* c = new C();
   ```
8. ligne 39 : `a1` non initialisé
9. ligne 41 : `b2` est une fonction (qui retourne un `B`), pas une instance de `B` ; il faut ajouter une valeur au constructeur (ou supprimer les parenthèses si l'on a fourni un constructeur par défaut (correction précédente))
10. ligne 43 : `.` au lieu de `->` sur le pointeur de `c`
11. ligne 45 : il faut libérer la mémoire pointée par `c`.

On pourrait aussi signaler qu'il serait en principe bon, quoique pas strictement nécessaire ici, d'ajouter un destructeur virtuel dans `A`.

Code commenté :

```
1   #include <iostream>
2   using namespace std;
3
4   class A {
5     public:
6       A(int x) : x(x) {}
7       virtual void f1() const = 0;
8       virtual void f2(ostream& out) const { out << "A::f2" << endl; }
9     private:
10      int x;
11  };
12
13  class B : public A {
14      void f3() { cout << "B::f3" << endl; }    // (1a) private (par défaut)
15    public:
16      void f1() const { cout << "B::f1" <<endl; }
17      int f2(ostream& out) const { out << "B::f2" << endl; return 0; } // (2) type de retour
18  };
19
20  void operator<<(ostream& out, const B & b) { b.f2(out); } // (3a) oubli du return out
21
22  class C : public A {
23    public:
24      C() : A(0), y(0) {}
25    void f2(ostream& out) { out << "C::f2" << endl; }  // (4) oubli du const => nouvelle méthode
26    private:
27      void f1() const { cout <<"C::f1" <<endl; }
28      int y;
29  };
30
31  void f2(B b) { b.f2(cout); }
32
33  int main() {
34    A a(1); // (5) impossible A est abstraite
35    B b; // (6) B n'a pas de constructeur par défaut
36    A* c = new C();
37    f2(*c); // (7) mauvais type (A au lieu de B)
38    cout << b << endl; // (3b) l'opérateur << doit retourner un ostream& (ou écrit en 2 lignes)
39    A* a1;
40    a1->f2(cout); // (8) certainement Segmentation fault, a1 n'a pas été initialisé
41    B b2();  // (9a) Ceci est une fonction. Cause une erreur à la ligne suivante.
42    b2.f3(); // (9b)
43    c.f2(cout); // (10) . au lieu de ->
44    b.f3();  // (1b) f3 est privée.
45    return 0; // (11) il faut désallouer c
46  }
```

Question 3 – Conception OO et programmation [40 points]

Cette question porte sur la conception d'un système de gestion pour un hôtel qui doit permettre de prendre des réservations et de consulter les réservations existantes, en respectant les consignes suivantes.

Données

Les données utilisées par le système correspondent à la description suivante :

- le système de gestion doit contenir une liste des chambres de l'hôtel, ainsi que des réservations existantes ;
- une chambre est caractérisée par un numéro, les lits qu'elle contient, son prix par personne et par jour et le nombre d'occupants (le nombre de personnes qui réservent la chambre) ;

  il existe des lits simples (à une place), doubles (à deux places), et superposés (à nombre d'étages variable, ayant une place par étage) ;
- une réservation est caractérisée par un numéro de réservation, un ensemble de chambres à occuper, une date d'arrivée et une date de départ [1].

Fonctionnalités

Grâce à ces données, le système doit fournir les fonctionnalités suivantes :

- spécifier la liste des chambres lors de l'initialisation de l'hôtel ;
- ajouter et supprimer des réservations ;

  on doit pouvoir créer une nouvelle réservation sans nécessairement spécifier toutes ses caractéristiques, mais chaque réservation aura au moins un numéro unique permettant de l'identifier ;
- étant donné une réservation, mettre à jour toutes ses caractéristiques (sauf le numéro de réservation qui doit rester constant) ;
- calculer le prix total du séjour correspondant à la réservation ;

  le prix total est la somme des prix de toutes les chambres (prix par personne de la chambre fois son nombre d'occupants), le tout multiplié par la durée du séjour (nombre de nuits) ;
- étant donnée une chambre, obtenir une description textuelle de toutes ses caractéristiques ainsi que son nombre maximal d'occupants ;

  étant donné un numéro de chambre, on doit également pouvoir afficher ces informations sur le terminal ; par exemple :

```
Chambre numero 1, 36 CHF par jour (par personne)
  Lit simple
  Lit double
Nb max. personnes : 3
Nb occupants : 2
```

- obtenir une description textuelle de toutes les caractéristiques et du prix total de la réservation ;

  étant donné un numéro de réservation, on doit aussi pouvoir afficher ces informations sur le terminal ; par exemple :

1. Peu importe le format des dates ici, choisissez le vôtre, et, si nécessaire, supposez qu'il existe une fonction `nuits()` qui prend deux dates et retourne le nombre de nuits correspondant.

```
Reservation numero 1, pour 2 personnes
  Chambre(s) : numero 4,
  Dates : du 3 juillet au 12 juillet (9 nuits)
  Prix total : 648 CHF
```

— finalement, il doit être possible d'afficher sur le terminal la liste de toutes les chambres, avec leurs caractéristiques, et la liste de toutes les réservations avec leurs caractéristiques et prix.

Consignes supplémentaires

La conception de votre programme doit minimiser la duplication de code et, dans chaque partie du programme, abstraire les détails inutiles à cette partie, de manière à minimiser les dépendances entre parties.

Par exemple, calculer le nombre de places total d'une chambre ne demande pas de connaître la nature exacte des lits, mais simplement le nombre de places qu'ils contiennent.

Finalement, il faudrait pouvoir construire la description textuelle de chaque entité composée (chambre, réservation) sans devoir inspecter les détails d'implémentation de leurs composants.

Questions

1. **(29 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).

   N'indiquez par contre *pas* les constructeurs, destructeurs, « méthodes set » (modificateurs) et « méthodes get » autres que ceux explicitement évoqués dans la liste de fonctionnalités précédente.
2. **(3 points)** Détaillez la manière dont la mémoire est gérée dans votre programme en spécifiant quelle partie ou classe a la responsabilité d'allouer la mémoire et de la libérer, et dans quelles méthodes ou fonctions vous l'implémenteriez (sans donner le détail du code lui-même. On demande ici toujours une conception).
3. **(2 points)** Détaillez, s'il y a lieu, les précautions à prendre lors de la définition du constructeur de copie et de l'opérateur d'affectation de vos différentes classes.
4. **(6 points)** Implémentez [2] la méthode/fonction du calcul du prix total d'une réservation.

Ci-dessous *une* réponse possible pour la conception. Il peut y avoir des variantes (qui font sens). Il n'est en particulier pas faux d'avoir ajouté des éléments supplémentaires (classes, attributs ou méthodes), tant qu'ils sont pertinents.

Note : il n'y a ici aucune raison de mettre les attributs en en `protected`.

2. C'est-à-dire donnez sa définition en C++.

```
typedef .... Date; // optionnel

// ========================================================================
class Affichable {
public:
  virtual string description() const = 0;
  virtual ~Affichable() = default; // ou simplement {}
};

ostream& operator<<(ostream& out, const Affichable& a);

// ========================================================================
class Reservation : public Affichable {
public:
  Reservation(unsigned int numero);
  void ajoute_chambre(Chambre& reference);
  void dates(Date const& arrivee, Date const& depart);
  double prix() const;
  virtual string description() const override;
  unsigned int nb_personnes() const; // optionnel
private:
  const unsigned int numero;
  vector<Chambre*> chambres; // références ; mieux : set<>
  Date arrivee;
  Date depart;
};

// ========================================================================
class Lit : public Affichable {
public:
  virtual unsigned int places() const;
  virtual string description() const override;
private:
  bool vertical;
  unsigned int places_;
};

// ========================================================================
class Chambre : public Affichable {
public:
  double prix_de_base() const;
  double prix_total() const; // optionnel, mais mieux : cf point (4)
  unsigned int places_totales() const; // optionnel
  unsigned int places_libres()  const; // optionnel
  bool reserver(unsigned int nb_occupants); // optionnel
  virtual string description() const override;
private:
  const unsigned int numero;
  vector<Lit> lits;
  double prix_;
  unsigned int occupants;
};
```

```
// ========================================================================
class Hotel : public Affichable  {
public:
  Hotel(vector<Chambre> const& chambres);
  Hotel(initilazer_list<Chambre> chambres); // autre possibilité
  bool ajouter(Reservation const&); // ou autres variantes avec autres paramètres
  void supprimer_reservation(unsigned int numero);
  bool mise_à_jour(unsigned int numero, Reservation const&); // optionnel
  virtual string description() const override;
  void afficher_chambres    (ostream& out) const;
  void afficher_reservations(ostream& out) const;
private:
  vector<Reservation> reservations; // mieux : set<>
  vector<Chambre> chambres;         // mieux : set<>
};
```

Alternativement, pour les `Lits`, on peut faire un peu plus compliqué (même si cela ne me semble pas nécessaire) :

```
class Lit : public Affichable {
public:
    virtual unsigned int places() const = 0;
};

// concrètes
class LitSimple     : public Lit;
class LitDouble     : public Lit;
class LitSupperpose : public Lit;
```

② Pour le modèle de mémoire : tout d'abord il n'y a en fait aucun polymorphisme à part celui de l'affichage (cf la série notée) et donc aucunement nécessité d'avoir des collections hétérogènes (pour une fois !).
(Note : il y a du polymorphimse sur les lits dans la version alternative qui créerait des sous-classes de lits ; et donc là il *doit* y avoir du polymorphisme à ce niveau.)

L'hôtel peut, et doit, être propriétaire de ses chambres et ses réservations.
De même, les chambres peuvent être propriétaires de leur lits. On parle ici d'un système de réservation, pas d'un hôtel physique (les « lits » sont des abstraction représentant la prestation offerte) et je ne vois donc aucune raison de mettre les lits ailleurs que directement dans les chambres.
Les reservations, par contre, doivent à mon avis *référencer* des chambres (propriétés de l'hôtel).
Il n'y a donc dans la version la plus simple aucune gestion mémoire à faire nous-même ; tout peut être fait avec de l'allocation statique ; par exemple (simple, et non demandé – juste pour illustrer l'esprit) :

```
int main() {
  Hotel h({
    Chambre(1, { Lit(1), Lit(1)       }, 10.5),
    Chambre(2, { Lit() , Lit(2, true) }, 15.0)
  });

  cout << h << endl;
}


void Reservation::ajoute_chambre(Chambre& c) { chambres.push_back(&c); }
```

Une solution avec allocation dynamique est bien sûr possible tant quelle reste cohérente (pas de duplication d'information, pas de fuite de mémoire) et justifiée (qui est propriétaire de quoi et pourquoi).

③ Pour les copies, il faut toujours faire attention aux pointeurs et savoir si la copie de surface (références vers le même objet) est correcte (rien à faire alors) ou s'il faut une copie profonde ou alors interdire la copie.

Avec la version présentée, il n'y a des pointeurs que dans les `Reservations` et ces pointeurs sont effectivement des références vers des chambres. Il n'y a donc pas de problème en soit pour la copie des `Reservations` **d'un même hôtel**.

Par contre, s'il l'on copie un hôtel, la copie de base de ses réservations contiendra des pointeurs vers les chambres de l'hôtel de départ et non pas vers les chambres de la copie.
Il est donc nécessaire, pour la copie de l'hôtel, de soit récrire la copie des réservations (ne pas les copier mais créer de nouvelles réservations qui pointent vers les copies des chambres) ou de l'interdire.

Il faudrait bien sûr procéder de même pour l'opérateur d'affectation.

Pour des conceptions plus compliquées, il faut faire attention aux autres pointeurs et justifier ses choix concernant leur copie (ou non).

④ Pour le calcul du prix total d'une réservation :

```
double Reservation::prix() const
{
  double total(0.0);
  for (auto const& c : chambres) {
    total += c->prix_total();
  }
  return total * nb_nuits(depart, arrivee);
}
```

Il est beaucoup mieux au niveau conception d'avoir une méthode `prix()` de la chambre qui retourne le prix total de la chambre que de faire nous-même ici ce calcul à base de « getters ».

`nb_nuit()` est ici plutôt une fonction, mais pourrait aussi tout à fait être une méthode de la classe `Reservation` et donc s'appeler tout simplement sans argument.

Question 4 – Exécution de programme [32 points]

Le programme ci-contre compile et s'exécute sans erreurs. Qu'affiche-t-il ?
Répondez ci-dessous, puis justifiez 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 !).

Réponse :

```
1: X() Y()
2: X() Y() Z() D()
3: D::g
4: Z::f
5: X(double)
6: ~X()
7: 0
8:
9: 2.2
10:
11: 5.5
12: ~X() ~X()
13: 9.9
14: ~X()
15: 15.4
16: ~X()
17: 15.4
18: ~D() ~Z() ~Y() ~X()
19: ~X() ~Y() ~X()
```

Justifications : Par exemple :

1. construction par défaut d'un `Y`, lequel commence par construire (par défaut) un `X` (héritage) ;
2. construction par défaut d'un `D` (via le `new`), lequel *commence* par construire (par défaut) un `X` (classe virtuelle) ; ensuite, les classes intermédiaires sont construites, mais *sans* appel au constructeur par défaut de `X` ;
3. `Z::g()` est virtuelle et appelée au travers d'un pointeur (les *DEUX*) donc polymorphisme ;
4. par contre `Z::f()` n'est pas virtuelle donc pas de polymorphisme (donc priorité au type) ;
5. `X(0)` appelle le constructeur `X(double)` (plongement silencieux de `int` en `double`) et peut être (en fait non, mais hors du cours) appel silencieux d'une copie ;
6. copie (silencieuse) de `a` dans `x` lors de l'appel de `f1()` et donc destruction de cette copie ;
7. comme `f1()` travaille sur une copie, `x` n'a pas été modifié ;
8. `g1()` reçoit une référence, il n'y a donc pas de copie (donc rien) ;
9. `g1()` a bien modifié `a` (passage par référence) ;

10 et 11. comme 8 et 9 ;

12. comme 6, mais en plus une copie (silencieuse) de retour, qui est aussi détruite ;
13. contrairement à 6–7, il y a ici une affectation lors du retour ;

14 et 16. comme 12 (retour) ;

17. ici aussi affectation (comme 13 et 15), mais `h2()` a sauvegardé la valeur reçue (et la retourne) ;

18 et 19. destructeurs toujours dans l'ordre inverse des constructeurs ; sur 19, à noter la destruction correspondant à 5 *avant* celle de 1 (celle de 2 ayant été déclanchée en 18 par le `delete`).

Les points importants de l'explication sont

1. l'ordre d'appel des constructeurs et destructeurs...
2. ...en particulier dans le cas de classes virtuelles (qu'une seule fois l'appel) ;
3. deux aspects pour polymorphisme de `g()` ;
4. manque virtuel pour `f()`, donc pas de polymorphisme ;
5. copie lors de l'appel (mais appel silencieux au constructeur de copie qui n'est pas verbeux) et destruction de cette copie (`6:` et 1re partie de `12:`). À mon avis ceux-ci sont difficiles !
6. pour `7:` : pas de modification de `a` car passage par valeur (copie) ;

7. pas de copie lors du passage par référence (`8:`) ni par pointeur (`10:`) ;
8. fonctionnement du passage par référence et ou par pointeur : modification de l'instance (`9:` doit donc être différent de `7:`, et `11:` de `9:` (et `7:`), idem `15:` et `17:`)
9. copie lors du retour et destruction (2e partie de `12:`, `14:` et **surtout** `16:`)
10. affectation (`13:`, `15:` et `17:`).

Notes : au niveau de ce cours, on peut tolérer :

— un appel de plus au destructeur dans `5:` : `5: X(double) ~X()`

car on pourrait penser que le transitoire non nommé (`X(0)`) est créé puis détruit (ce qui n'est pas le cas en raison de l'élision de copie) ;

— que la création de la copie de retour de `f2()` et de `g2()` (qui ne peuvent pas être élidées) ne soient pas vues et donc pas de destruction de cette copie, donc :
  — un seul destructeur en `12` : `12: ~X()`
  — pas de destructeur en `14`,

mais il faut que ces deux là soient cohérents !

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

class X {
  public:
    X()                { cout << "X() " ;       }
    virtual ~X()       { cout << "~X() ";       }
    X(double x) : a(x) { cout << "X(double) ";  }
    void set(double x) { a = x; }
    double get() const { return a; }
    void print() const { cout << get() << " " ; }
  private:
    double a;
};

class Z : virtual public X {
  public:
    Z()                    { cout << "Z() " ; }
    virtual ~Z()           { cout << "~Z() "; }
    void f() const         { cout << "Z::f "; }
    virtual void g() const { cout << "Z::g "; }
};

class Y : virtual public X {
  public:
    Y()          { cout << "Y() " ; }
    virtual ~Y() { cout << "~Y() "; }
};

class D : public Y, public Z {
  public:
    D()            { cout << "D() " ; }
    virtual ~D()   { cout << "~D() "; }
    void f() const { cout << "D::f "; }
    void g() const { cout << "D::g "; }
};

void f1(X  x) { x.set( x.get()  + 1.1); }
void g1(X& x) { x.set( x.get()  + 2.2); }
void h1(X* x) { x->set(x->get() + 3.3); }

X f2(X  x) { x.set(x.get() + 4.4); return x; }
X g2(X& x) { x.set(x.get() + 5.5); return x; }
X h2(X* x) {
  X x1 = *x;
  x->set( x->get() + 6.6 );
  return x1;
}
```

```
void pinc(int& u) {
  cout << endl << u << ": ";
  ++u;
}

int main() {
  int i(1);
                    pinc(i); //  1
  Y c;
                    pinc(i); //  2
  Z* b = new D;
                    pinc(i); //  3
  b->g();
                    pinc(i); //  4
  b->f();

                    pinc(i); //  5
  X a = X(0);
                    pinc(i); //  6
  f1(a);
                    pinc(i); //  7
  a.print();
                    pinc(i); //  8
  g1(a);
                    pinc(i); //  9
  a.print();
                    pinc(i); // 10
  h1(&a);
                    pinc(i); // 11
  a.print();
                    pinc(i); // 12
  a = f2(a);
                    pinc(i); // 13
  a.print();
                    pinc(i); // 14
  a = g2(a);
                    pinc(i); // 15
  a.print();
                    pinc(i); // 16
  a = h2(&a);
                    pinc(i); // 17
  a.print();

                    pinc(i); // 18
  delete b;
                    pinc(i); // 19
  return 0;
}
```