Correction du partiel TYLA

EPITA – Promo 2012
Tous documents (notes de cours, polycopiés, livres) autorisés.
Calculatrices et ordinateurs interdits.

Juin 2010 (1h30)

Correction: Le sujet et sa correction ont été écrits par Roland Levillain. Le *best of* est tiré des copies des étudiants, fautes de français y compris, le cas échéant.

Une copie synthétique, bien orthographiée, avec un affichage clair des résultats, sera toujours mieux notée qu'une autre demandant une quelconque forme d'effort de la part du correcteur. Une argumentation informelle mais convaincante, sera souvent suffisante.

Une lecture préalable du sujet est recommandée.

1 Généralités

1. Combien de nombres entiers différents peut-on représenter avec 1024 bits ?

   Correction: 2^{1024}.

   Best-of:
   - $2^{1024} - 1$
   - $2^{1025} - 1$
   - 1024/32 bits = 32 entiers

2. En Programmation Orientée Objet (POO), qu'est-ce que l'encapsulation ?

   Correction: Le fait de regrouper les données (les attributs) et les actions/algorithmes qui les manipulent (les méthodes) au sein d'une même entité (une classe). À ne pas confondre avec le masquage des données, cf. infra.

3. En POO, qu'appelle-t-on masquage de données ?

   Correction: Le fait d'empêcher l'accès (direct) aux données d'une classe, obtenu par les mots clefs `private`, `protected` (et `public`) en C++. L'accès se fait alors à l'aide de méthodes (accesseurs) restreignant éventuellement l'accès en lecture ou en écriture, et exerçant possiblement la vérification de contrats (préconditions, postconditions, invariants).

4. Qu'appelle-t-on structure de blocs ? Citer un langage qui dispose de cette fonctionnalité.

Correction: La possibilité, dans un programme, de définir des blocs d'instructions, ces blocs pouvant eux-mêmes être imbriqués dans d'autres blocs. Ainsi, un bloc de déclaration peut être imbriqué dans un bloc d'instructions, permettant la définition locale de variables, fonctions, etc.
Dans la plupart des langages, les noms des variables, fonctions, types, etc. déclarés dans les blocs extérieurs sont automatiquement accessibles depuis les blocs intérieurs, sauf s'ils sont masqués par une entité du même nom.
Parmi langages fournissant la structure de blocs, on trouve Algol, Lisp, Objective Caml, Haskell, et bien sûr, Tiger.
Les blocs délimités par les accolades dans les corps des fonctions dans les langages C et C++ ne répondent pas à cette définition : on ne peut définir de fonction imbriquée dans de tels blocs par exemple.
Attention également à ne pas confondre la structure de blocs avec les instructions composées (*compound statement*), qui sont des groupes d'instructions (entre accolades en C++) utilisables en lieu et place d'une instruction ("atomique"), particulièrement utiles avec les structures de contrôle (`if`, `for`, etc.).

5. Quelle différence faites-vous entre portée statique et portée dynamique ?

Correction: La portée statique (appelée autrement portée lexicale) définit la visibilité d'une entité (variable, fonction, type, etc.) en fonction de sa position dans le code source du programme. Ainsi, le compilateur peut déterminer statiquement pour chaque entité l'espace (au sens d'un intervalle d'instructions) dans lequel elle sera visible.
La portée dynamique s'étend à la durée de vie d'une entité : sa portée est globale et sa durée de vie est dynamique. Ainsi, il est impossible de la déterminer à la compilation ; c'est le *run time* qui sera chargé de gérer les accès à celle-ci (liaison dynamique).

2 C++

1. En C++, de quels éléments dépend la sélection d'une méthode surchargée ?

Correction: Sont pris en compte les éléments suivants :
- le nom de la méthode ;
- le nombre d'arguments et le type (statique) de chacun d'entre-eux ;
- le qualificatif `const` de la méthode.

Le type de retour n'est pas pris en compte.

2. Quelle différence faites-vous entre initialisation et affectation ? En C++, quels individus sont chargés de la première ? quels sont ceux qui sont chargés de la seconde ?

Correction: L'initialisation s'effectue à la construction d'un objet (au sens large : on parle aussi d'initialisation pour les types atomiques du langage comme `int`, `float`, etc.). L'affectation est une opération qui a lieu *après* la construction d'un objet.
La distinction entre les deux est importante pour plusieurs raisons. La première est que, dans le cas des instances de classes, la routine appelée est différente. Dans le cas de l'initialisation, il s'agit d'un constructeur; dans le cas de l'affectation, il s'agit d'un opérateur d'affectation (`operator=`). Le programmeur est libre d'implémenter les deux différemment, d'où l'importance de les définir ensemble et de façon cohérente le cas échéant, par exemple, si l'objet doit allouer/libérer de la mémoire dynamiquement pour un attribut manipulé par pointeur.
Une autre raison est que le mécanisme d'initialisation est le seul qui permette d'initialiser des références et des constantes (à moins de tricher avec des mécanismes "bas niveau", type casts ou unions). Une fois passée l'initialisation d'une référence, il n'est plus possible de l'associer à un autre objet ; de même, affecter une valeur à une constante après son initialisation provoque une erreur de typage à la compilation.
Attention, le C++ n'aide pas toujours à distinguer initialisation et affectation. En effet, l'opérateur '=' peut également être utilisé lors de la construction d'un objet, et pourtant déclencher l'appel d'un constructeur. Par exemple :

```
#include <iostream>

struct A
{
  A(int)
  {
    std::cout << "A::A(int)"         << std::endl;
  }
  A& operator=(int)
  {
    std::cout << "A::operator=(int)" << std::endl;
    return *this;
  }
};

int main()
{
  A a = 42;  // Calls A::A(int).
  a = 51;    // Calls A::operator=(int).
}
```

Cependant, le fait qu'une initialisation ait lieu lors de la déclaration d'une variable (comme dans le code précédent, ou à l'intérieur d'une liste d'initialisation d'un constructeur) suffit à distinguer les deux opérations.

3. En C++, l'ordre d'initialisation des objets globaux situés dans des unités de compilation différentes n'est pas précisé par le standard. En quoi cela peut-il être un problème ?

   Correction: Parce que cela pourrait introduire des comportements non déterministes ; ou dans une moindre mesure des comportements dépendants de l'implémentation (*implementation-defined*), sans garantie de portabilité.
   Cela se produit par exemple lorsque la construction d'objet global `a`, compilé dans un fichier `a.o`, dépend de la construction d'un objet global `b`, compilé dans un fichier `b.o`. Il est tout à fait possible que `b` ne soit pas initialisé au moment de la construction de `a`, produisant potentiellement des erreurs.
   Une exception à ce comportement est celle des objets globaux de la bibliothèque standard représentant les flux standard (`std::cout`, `std::cin`, etc.). Le standard impose que le runtime C++ initialise ces objets *avant* la construction des objets globaux, de sorte que ces dernier puissent les utiliser sans déclencher à l'exécution les problèmes évoqués ci-avant.

4. Pourquoi est-il conseillé de définir un constructeur par copie et un opérateur d'affectation dans une classe possédant un (des) attribut(s) de type pointeur ?

   Correction: Pour s'assurer de la cohérence de la gestion mémoire, quelle que soit la façon dont un objet est construit ou modifié.
   Prenons l'exemple d'un objet dont le constructeur alloue dynamiquement (avec `new`) de la mémoire pour son attribut de type pointeur sur entier, et dont le destructeur libère cet espace mémoire (avec `delete`). Imaginons que son `operator=` se contente de faire une copie peu profonde (*shallow copy*) en copiant uniquement l'adresse de l'attribut de l'argument (l'objet original, copié). Alors deux objets référencent désormais la même zone mémoire (l'original et sa copie), qui sera potentiellement libérée deux fois et pourra entraîner des erreurs à l'exécution.
   Pour éviter de tels désagréments, il est important d'écrire constructeur et opérateur d'affectation (ainsi que destructeur) ensemble, pour garantir une gestion cohérente des ressources (mémoire, fichiers, etc.)
   Ceci est d'autant plus vrai lorsque la classe en question dispose de plusieurs constructeurs ou opérateurs d'affectations (surchargés).

5. On rappelle que la *mise en ligne* du corps d'une fonction (*inlining*) est une optimisation consistant à remplacer un appel de fonction ou de méthode par le corps de celle-ci au site d'appel (en remplaçant les arguments formels par les arguments effectifs).

   En C++, il existe plusieurs situations dans lesquelles une fonction (ou méthode) ne peut être mise en ligne ; citez-en deux. (On ne s'attache pas ici à la qualité de l'optimisation, donc une réponse telle que "le compilateur refuse l'inlining car la fonction à mettre en ligne est trop grosse" n'est pas pertinente).

   Correction: Au choix:
   - la définition de la fonction n'est pas visible depuis le site d'appel (c'est-à-dire, dans la même unité de compilation) ;
   - la fonction candidate à la mise en ligne est récursive (directement ou indirectement) : la mise en ligne ne peut avoir lieu, car il s'agirait d'un processus sans fin. Bien entendu, on pourrait proposer de limiter la profondeur d'inlining à un certain nombre d'appels récursifs, mais ça commence à devenir trop subtil pour une question qui se voulait initialement simple !
   - on ne peut mettre en ligne une méthode polymorphe (c'est-à-dire une fonction membre virtuelle en C++), car sa liaison est effectuée à l'exécution, alors que la mise en ligne est une opération effectuée à la compilation ;
   - un peu pour les mêmes raisons, on ne peut mettre en ligne une fonction appelée via un pointeur sur fonction ;
   - enfin, il se trouve que certains compilateurs refusent de mettre en ligne une fonction dès lors qu'elle accepte un nombre variable d'arguments (opérateur "`...`"du C++). Sans être impossible, ce choix d'implémentation est compréhensible car le passage et la récupération d'arguments variables s'effectuent via une manipulation du cadre de pile, dont la traduction lors de la mise en ligne pourrait être non triviale.

6. On suppose que vous disposez du code d'une classe C++ nommée `Foo`, qui contient déjà des méthodes et des attributs. Si vous ajoutez des méthodes non virtuelles dans le code de `Foo`, est-ce que les instances (objets) de type `Foo` prendront plus de place en mémoire ? Justifiez votre réponse.

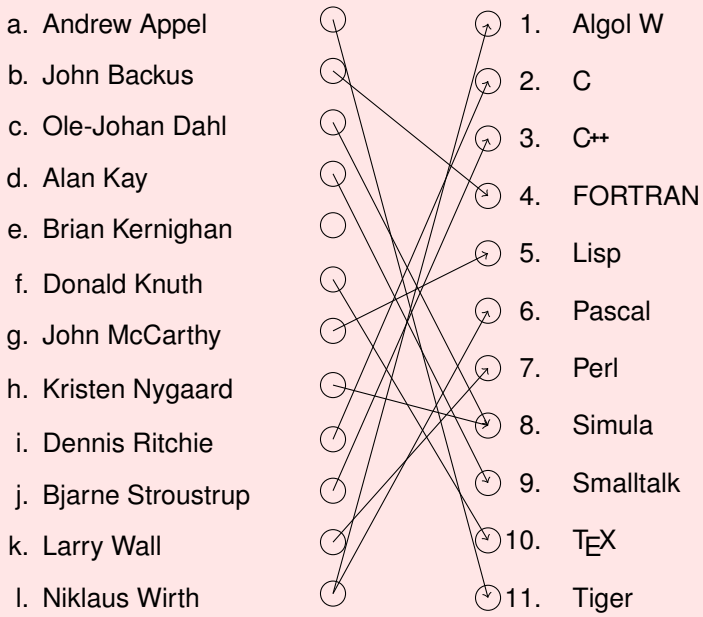
   Correction: Le fait d'ajouter des méthodes non virtuelles n'a pas d'impact sur la taille des instances. La taille d'un objet dépend de ses données (la valeur des attributs) et de la présence éventuelle d'un pointeur sur table de fonctions virtuelles, lorsque la classe comporte des méthodes polymorphes (virtuelles).

3 Autres langages de programmation

1. Appariez (sur votre copie) chaque auteur avec son langage :

a. Andrew Appel	1. Algol W
b. John Backus	2. C
c. Ole-Johan Dahl	3. C++
d. Alan Kay	4. FORTRAN
e. Brian Kernighan	5. Lisp
f. Donald Knuth	6. Pascal
g. John McCarthy	7. Perl
h. Kristen Nygaard	8. Simula
i. Dennis Ritchie	9. Smalltalk
j. Bjarne Stroustrup	10. TEX
k. Larry Wall	11. Tiger
l. Niklaus Wirth	

Correction:



Brian Kernighan n'est pas auteur du C, même s'il a participé à le rendre populaire en tant qu'auteur principal du livre *The C Programming Language* (avec Dennis Ritchie).

2. En Algol 60, on peut implémenter une technique de programmation appelée *Jensen's Device* (inventée par Jørn Jensen, qui a travaillé avec Peter Naur), qui permet d'écrire ceci :

```
real procedure sum (i, lo, hi, term);
  value lo, hi;
  integer i, lo, hi;
  real term;
  comment 'term' is passed by−name, and so is 'i';
begin
  real temp;
  temp := 0;
  for i := lo step 1 until hi do
    temp := temp + term;
  sum := temp
end;
```

Cette technique repose sur le passage d'argument par nom (*call by name*) présent dans Algol 60.

(a) Rappelez la signification de "Algol".

Correction: ALGOL: **ALGO**rithmic **L**anguage.

Best-of: Cela signifie algorithme léger.

(b) Qu'appelle-t-on passage par nom ?

Correction: Une politique de passage d'argument dans laquelle la valeur symbolique (le nom) de l'argument effectif est passée à la fonction appelée, au lieu de sa valeur. L'argument est alors substitué textuellement dans la définition de la fonction, et réévalué à chaque utilisation.
Une fonction utilisant cet mode de passage d'argument se comporte un peu comme une macro du C/C++.

(c) Réécrivez ce bout de code en C++.

Correction: Cette question n'est pas évidente, car le C++ ne dispose pas de la sémantique de passage par nom. Il y a plusieurs façons de procéder. La plus immédiate consiste effectivement à considérer la fonction `sum` comme une macro en C++ pour passer ses arguments '`i`' et '`term`' par nom. On fera attention de n'évaluer qu'une seule fois les autres arguments ('`lo`' et '`hi`') car ils doivent être passés par valeur. Enfin, comme nous ne disposons pas de valeur de retour, on ajoutera un cinquième argument pour retourner le résultat (sémantique de passage par résultat).

```
#define SUM(I, LO, HI, TERM, RESULT)          \
 do                                           \
  {                                           \
   /* Argument passed by value, evaluated only  \
      once. */                                 \
   int lo = LO;                                \
   int hi = HI;                                \
                                               \
   float temp = 0.f;                           \
   for (int I = lo; I <= hi; I = I + 1)        \
    temp += TERM;                              \
   RESULT = temp;                              \
  }                                            \
 while (0)
```

Correction: (suite)
Cette première solution a le mérite d'être aussi puissante que le code Algol, puisque l'on peut passer à peu près ce qu'on souhaite pour '`I`' et '`TERM`' (sous réserve que cela ait un sens et respecte la syntaxe du langage). La contrepartie est inhérente aux macros : les erreurs sont difficiles à trouver, il peut y avoir des effets de bord, il faudrait éventuellement protéger les arguments pour préserver l'ordre d'évaluation, etc.
Une autre solution est de s'appuyer sur des objets-fonctions (functors) pour représenter '`i`' et '`term`', par exemple en utilisant la bibliothèque Boost Lambda. On peut représenter les deux arguments passés par nom par des functors Boost Lambda, qui doivent donc être évalués à chaque utilisation (d'où l'emploi de l'opérateur '`()`') :

```
#include <boost/lambda/lambda.hpp>

// 'i' and 'term' must be Boost Lambda expressions
// (functors).
template <typename I, typename T>
float sum (I i, int lo, int hi, T term)
{
  float temp = 0.f;
  for ((i = lo)(); (i <= hi)(); (++i)())
    temp += term();
  return temp;
}
```

Cette seconde solution a l'avantage de reposer sur une vraie fonction (et non une macro), avec toute l'aide que le compilateur peut apporter. En revanche, on a (un peu) perdu en flexibilité, puisque '`i`' et '`term`' doivent désormais être des functors.
On trouvera des exemples d'utilisation de ces deux solutions dans la réponse à la question 2d.

(d) Écrivez un bout de code Algol utilisant la procédure `sum` du code Algol ci-dessus, permettant d'afficher le 100e nombre harmonique noté H_{100}, défini ainsi :

$$H_{100} = \sum_{i=1}^{100} \frac{1}{i}$$

(Vous ne serez bien entendu pas pénalisés sur des erreurs de syntaxe mineures.)

Correction:

```
integer i;
print (sum (i, 1, 100, 1/i))
```

L'équivalent C++ avec la solution basée sur la macro `SUM` s'écrirait ainsi :

```
float h100;
SUM (i, 1, 100, 1.f/i, h100);
std::cout << h100 << std::endl;
```

Quant à la solution à base de functors Boost Lambda, elle exige d'utiliser la construction `var(i)`, qui crée un functor d'arité 0 (c'est-à-dire, sans argument), évaluant '`i`'. De même, l'écriture '`1.f / var(i)`' crée un objet-fonction s'évaluant comme l'inverse de '`i`' (avec une division flottante).

```
int i;
using namespace boost::lambda;
std::cout << sum (var(i), 1, 100, 1.f / var(i))
          << std::endl;
```

3. Considérez le code Haskell ci-dessous.

```
1 let ones = 1 : ones
2 in head ones
```

(a) Que fait la ligne 1 ?

Correction: Elle définit '`ones`', une liste "infinie" de "1", définie récursivement.

(b) Pourquoi le programme ci-dessus ne boucle-t-il pas indéfiniment ?

Correction: Parce qu'Haskell est un langage à évaluation paresseuse (*lazy*) : la liste n'est pas évaluée (et donc pas construite) lors de sa définition, mais uniquement lorsque l'un de ses éléments est requis. En l'occurrence, `head ones`, qui retourne l'entier en tête de '`ones`', n'a besoin que de ce premier élément ; le reste de la liste n'est pas évalué.