# Correction du partiel CMP1 & TYLA

EPITA – Apprentis promo 2011
**Tous documents (notes de cours, polycopiés, livres) autorisés**
**Calculatrices et ordinateurs interdits.**

Janvier 2009 (1h30)

**Correction:** Le sujet et sa correction ont été écrits par Roland Levillain.

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.

Merci de répondre sur votre copie, même pour les questions à choix multiples.

## 1 Un peu de C++

Considérons le code C++ ci-dessous :

```
#include <cstddef>
struct Buffer
{
 typedef unsigned char value_type;
 Buffer (size_t n) : data_ (new value_type[n]) {}
 ~Buffer () { delete[] data_; }
 value_type& operator[] (size_t n) { return data_[n]; }
 value_type* data_;
};
```

Tel quel, ce code comporte des problèmes ou des limitations que nous allons examiner.

1. Le code précédent, utilisé dans l'exemple ci-après, provoque une erreur.

   ```
   int main ()
   {
    Buffer b (51);
    for (unsigned i = 0; i < 51; ++i)
     b[i] = 42;
    {
     Buffer b2 = b;
    }
    {
     Buffer b3 (1);
     b3[0] = 64;
    }
    unsigned char c = b[0];
   }
   ```

   Pourquoi ?

**Correction:** La classe `Buffer` contient des membres de type pointeur, mais toutes les routines n'en tiennent pas compte. En cours, on a vu qu'il fallait dans cette situation porter attention à la gestion mémoire dans certaines méthodes : constructeur(s), destructeur, opérateur(s) d'affectation.
Ici, le destructeur libère systématiquement les données derrière `data_`. Or le constructeur par copie, fourni implicitement par le compilateur, a une sémantique de partage des données : il copie `data_` (le pointeur) directement, pas ses données. Dans l'exemple, cela signifie que la ligne

```
Buffer b2 = b;
```

qui construit `b2` par copie de `b`[a], fait partager à `b` et `b2` le même tableau derrière `data_` en mémoire. Ce qui pose évidemment un problème lors de la destruction de ces objets : les données sont libérées deux fois. Pire : le fait que `b2` meure avant `b` fait que ce dernier voit ses données détruites alors qu'elles sont réutilisées un peu plus bas (dans la dernière ligne de `main`).

*a.* Il s'agit bien d'un appel de constructeur, et non d'une affectation, malgré la présence de '='.

2. Comment corriger cela ?

**Correction:** Il suffit d'écrire (proprement) un constructeur par copie, ainsi qu'un `operator=`, pour éviter que le même problème ne se pose avec l'affectation. Pour cela, il faut conserver la taille du tableau alloué dans un attribut (`n_`, par exemple).

```
#include <cstddef>
#include <algorithm>
struct Buffer
{
  typedef unsigned char value_type;
  Buffer (size_t n) : n_ (n), data_ (new value_type[n]) {}
  Buffer (const Buffer& rhs)
    : n_ (rhs.n_), data_ (new value_type[rhs.n_])
  {
    std::copy (rhs.data_, rhs.data_ + rhs.n_, data_);
  }
  Buffer& rhs operator= (const Buffer& rhs)
  {
    if (&rhs != this)
    {
      n_ = rhs.n_;
      data_ = new value_type[rhs.n_];
      std::copy (rhs.data_, rhs.data_ + rhs.n_, data_);
    }
    return *this;
  }
  ~Buffer () { delete[] data_; }
  value_type& operator[] (size_t n) { return data_[n]; }
  size_t n_;
  value_type* data_;
};
```

3. Que peut-on changer pour simplifier le travail de debugging du développeur ?

**Correction:**
– Ajouter un `operator[]` constant :

```
const value_type& operator[](size_t n) const
{
  return data_[n];
}
```

– Vérifier que `n` n'est pas nul dans le constructeur.
– Conserver la taille du bloc alloué pour `data_` sous forme d'attribut, et vérifier que `n` est strictement inférieur à cette valeur dans `operator[]`.
– Éventuellement, traiter une exception levée par `new` dans le constructeur.

4. Que peut-on changer pour étendre le champ d'application de cette classe (dit autrement : pour pouvoir vraiment la réutiliser) ?

**Correction:**
– Une méthode `realloc`/`resize`.
– Une méthode `fill`, un constructeur prenant une valeur par défaut.
– Faire de `value_type` un paramètre de `Buffer` pour rendre la classe générique vis-à-vis du type de contenu.

5. Après les corrections de la question 2, on ajoute la ligne suivante à la fin de la définition de `main`.

```
std::cout << c << std::endl;
```

Qu'affiche-t-elle ?

**Correction:** Le caractère « * » (càd, le caractère numéro 42), suivi d'un retour à la ligne. En effet, l'opérateur de flux '«' surchargé pour `char`, `signed char` et `unsigned char`, affiche le code ASCII correspondant à l'argument, et non sa valeur numérique.

## 2 Passage d'arguments

1. Quel(s) type(s) de passage(s) d'argument(s) est (sont) utilisé(s) en C ?
   (a) passage par valeur/copie (*call by value*)
   (b) passage par référence (*call by reference*)
   (c) passage par résultat (*call by result*)
   (d) passage par nom (*call by name*)

**Correction:** Seule la réponse 1a est valide : les autres types de passage d'arguments n'existent pas en C.

2. Même question avec C++ (attention, il y a un piège !).

**Correction:** En sus du passage par valeur, C++ supporte le passage par référence grâce à `&`.

3. Même question avec Tiger.

**Correction:** Tiger a la même sémantique de passage par copie que C et C++. Les valeurs complexes (tableaux, objets, etc.) sont manipulées par référence.

4. Il n'y a pas de type référence en Tiger ; comment peut-on les simuler avec des constructions du langage existantes ?

**Correction:** En utilisant des tableaux, des enregistrements ou des objets, puisque ceux-ci sont implicitement manipulés par pointeurs/référence en Tiger. Le programme suivant affiche `51` sur la sortie standard.

```
let
  type arr = array of int
  var a := arr[1] of 42
  var b := arr[1] of 0
in
  b := a;
  b[0] := 51;
  print_int(a[0]);
  print("\n")
end
```

Pour information, c'est d'ailleurs grâce à des enregistrements à champ unique que sont conçues les référence en Objective Caml :

```
# let a = ref 42;;
val a : int ref = { contents = 42 }
```

Attention, plusieurs copies mentionnent la possibilité d'utiliser des alias de type avec la construction `type` $t = u$ pour faire profiter $t$ d'une sémantique de passage par référence que $u$ n'aurait pas. Il n'en est rien : $t$ est un synonyme de $u$, et de ce fait, il copie les comportements de $u$ ; si $u$ est un type passé par valeur (copie), alors $t$ l'est aussi.

# 3 Const

À la différence de C++, Tiger est dépourvu de mot clef `const`. On se propose dans cet exercice de le rajouter dans le langage, en passant notamment en revue les différentes parties de notre compilateur afin de déterminer quels sont les aménagements nécessaires.

Rappel : le langage Tiger utilisé dans cet exercice est celui qui est décrit dans le Manuel de Référence du Compilateur Tiger, qui est utilisé comme référence tout au long du projet du même nom.

## 3.1 Préliminaires

1. À quoi sert `const` en C++ ? Il y a plusieurs réponses possibles, une seule est attendue).

   **Correction:**
   - Créer des constantes, au sens strict.
   - Créer des alias (via des pointeurs ou des références) de variables existantes, en restreignant leur accès à la simple lecture.
   - Appliqué à une méthode, garantir que le code de celle-ci ne modifiera pas l'objet cible.
   - Par extension des points précédents, permettre certaines optimisations (propagation de constantes, élimination de constantes, évaluation partielle, etc.).

   Attention, `const` ne sert pas à « figer » une zone mémoire comme certains l'ont écrit. Dans le code ci-dessous, `i` et `j` désignent la même zone mémoire, mais le fait que `j` soit qualifié de `const` ne donne aucune garantie sur la constante de `i` !

   ```
   int i = 42;
   const int& j = i;
   ```

2. Parmi les lignes suivantes, quelles sont celles qui sont valides en C++ ? Justifiez votre réponse.

(a) `int i = 42; const int* const p = &i;`
(b) `int i = 42; const int const * p = &i;`
(c) `int i = 42; int const * p = &i;`
(d) `int i = 42; int const * const p = &i;`

**Correction:** Les symboles '&' ont sauté dans la version du sujet distribuée au partiel, désolé. J'ai ajusté la notation en conséquence.
Seule la ligne 2b est incorrecte ; les autres sont juste. Rappelons la règle du placement du mot clef `const` : celui-ci s'applique à ce qui précède, sauf lorsqu'il est placé en tête ; dans ce cas, il qualifie ce qui suit. Dans le cas de la ligne 2b, les deux `const` s'appliqueraient donc tous les deux à `int`, ce que le compilateur refuse.

3. Même question.
(a) `int i = 42; const int& const p = i;`
(b) `int i = 42; const int const & p = i;`
(c) `int i = 42; int const & p = i;`
(d) `int i = 42; int const & const p = i;`

**Correction:** Ici, une règle supplémentaire s'applique : une référence est implicitement un pointeur constant déguisé. En conséquence, `const` ne peut s'appliquer à `&`. Seule la réponse 3c est donc valide ici.

4. Soit la classe suivante :

```
struct C
{
 void m1 () {}       // (0).
 void m1 () const {} // (1).
 void m2 () {}       // (2).
 void m3 () const {} // (3).
};
```

Pour chacun des cas suivants, indiquer quelle méthode est appelée (0, 1, 2 ou 3) ou s'il y a une erreur de compilation, le cas échéant.

(a) `C o; o.m1();`
**Correction:** Réponse : 0 (`o` est non constant).
(b) `const C o = C(); o.m1();`
**Correction:** Réponse : 1 (`o` est constant).
(c) `const C o = C(); o.m2();`
**Correction:** Erreur de compilation : `m2` est une méthode non constante, et ne peut être appelée avec une cible constante (`o`).
(d) `C o; o.m3();`
**Correction:** Réponse : 3 (on peut passer un objet mutable à une méthode constante).
(e) `void (C::*m)() const = &C::m1; (C().*m)();`
**Correction:** Réponse : 1. Ici, `m` est un pointeur sur une méthode qui a la signature `void (C::*)() const`, initialisé avec l'une des méthode `m1` de `C`. Le `const` à la fin de la signature de `m` sélectionne la version constante de `m1`, d'où la réponse.
(f) `void (C::*m)() = &C::m3; (C().*m)();`
**Correction:** Erreur de compilation : C::m3 ne respecte pas le type du pointeur sur méthode `m`, qui ne se termine pas par un `const`, alors que c'est le cas de `m3`.

## 3.2 Const en Tiger

Cette partie s'intéresse aux modifications à apporter au front end Tiger pour prendre en compte `const`. Nous ne considérerons ce nouveau mot-clef que comme qualificatif de types (le cas des méthodes « constantes » n'est pas à traiter).

1. **Spécification lexicales.** Que faut-il ajouter aux spécifications lexicales du langage pour supporter `const` ?

   **Correction:** Il suffit d'ajouter un nouveau mot-clef `const`. Le terme *token* ne convient pas ici : il s'agit de spécification, pas d'implémentation.

2. **Scanner.** Comment étendre le scanner, c'est-à-dire, que faut-il ajouter/modifier dans le fichier '`scantiger.ll`' pour supporter ces nouvelles spécifications lexicales ?

   **Correction:** Quelque chose de ce genre dans la section des règles lexicales (entre les deux occurrences de `%%`) :

   ```
   "const"          return token::CONST;
   ```

   Il faudra également penser à définir un nouveau token `CONST` dans '`parsetiger.yy`' (cf. réponse à la question 5).

3. **Spécification syntaxiques.** Que faut-il ajouter aux spécifications syntaxiques du langage pour supporter `const` ?

   **Correction:** L'extension la plus simple et qui conserve tout le pouvoir d'expression de `const` (au détriment de constructions plus lourdes par endroits) consiste à augmenter la règle de production `type-id` pour qu'elle accepte les identifiants précédés de `const`.

   ```
   type-id → id
   type-id → const id
   ```

   Certes, cela ne permet pas des écritures telles que :

   ```
   type t = const array of int
   ```

   mais il suffit de remplacer par :

   ```
   type u = array of int
   type t = const u
   ```

   pour contourner cette limitation. Il bien sûr possible de supporter la première écriture, en modifiant un peu plus la grammaire, et au prix de quelques modifications pour lever les ambiguïtés introduites. Dans le reste de la correction, on considère que c'est l'extension la plus simple (la première décrite ci-avant) qui est adoptée.

4. **Syntaxe abstraite.** Faut-il effectuer des modifications ou des ajouts dans la syntaxe abstraite du langage pour supporter `const` ? Si oui, lesquelles ?

   **Correction:** Oui, il faut pouvoir matérialiser `const` dans la syntaxe abstraite. Plusieurs solutions s'offrent à nous.
   - En ajoutant un attribut booléen (par exemple `const_`), à ast::NameTy, mis à `false` par défaut, et à `true` lorsque le type est précédé de `const`. Il faut équiper la classe : ajout d'accesseur(s), modification de constructeurs, etc.
   - En ajoutant une classe ast::ConstNameTy, symétrique de ast::NameTy. Pour des raisons de factorisation, on peut au choix
     - faire que ast::ConstNameTy, dérive de ast::NameTy ;
     - faire dériver ces deux classes d'une même classe abstraite pour les mettre au même niveau dans la hiérarchie.

   Pour des raisons de simplicité, nous choisissons la première solution pour le reste de la correction, mais toutes les réponses valides ont été acceptées.

5. **Parser.** Comment étendre le parser, c'est-à-dire, que faut-il ajouter/modifier dans le fichier '`parsetiger.yy`' pour supporter ces nouvelles spécifications syntaxiques ?

**Correction:** Les ajouts nécessaires sont :
– la définition du token `CONST`, déjà mentionnée plus haut ;
– la règle de production de la question précédente.

```
// ...
%token CONST       "const"
// ...
%%
// ...
typeid:
  ID      { $$ = new ast::NameTy (@$, take ($1), false); }
  // New rule.
| CONST ID { $$ = new ast::NameTy (@$, take ($2), true); }
;
// ...
```

où le constructeur de `ast::NameTy` est étendu pour prendre un booléen représentant la constance du type.