Compilation

Mise à niveau en C++ (TP 0.1 - le retour)

G.Iooss & C.Alias

Ce TP s'adresse aux personnes qui n'ont pas l'habitude de programmer en C++. Afin de pouvoir venir à bout des prochains TPs, il est essentiel de pouvoir lire un code C++ (relativement simple), et de connaître les bases de la syntaxe (pour pouvoir manipuler les outils fournis). On suppose que vous connaissez un minimum de C (Du côté théorique: déclaration de variables, boucle `for`, pointeurs, .../ Du côté pratique: et headers, compilation, exécution, ...) [1]

Exercice 1 : Rappel du C et nouveautés du C++

Voilà un exemple de programme en C++:

```
#include <iostream>                // Pour utiliser cout, cin et endl
using namespace std;     // Pour utiliser la STD

// Fonction annexe: rend "vrai" si a est plus grand que b
bool estplusgrand (int a, int b) {
   return (a >= b);
}

// Fonction principale (exécuté au lancement du programme)
int main(void) {          // "main" peut avoir des arguments (optionnels)
   int tab[5] = {1,5,3,0,7};
   int plusgrand = -1;
   for (int i=0; i<5; i++) {
      bool cond = estplusgrand(plusgrand,tab[i]);
      if (!bool)
         plusgrand = tab[i];
      // Affiche la valeur du plus grand élément
      cout << "A l'itération " << i
           << " le plus grand élément est: " << plusgrand << endl;
   }
   return 0;
}
```

On peut observer quelques différences avec le C:

- *Flots de données:* L'utilisation de `cout` (flux de sortie standard, càd l'écran) et de l'opérateur `«` permettent d'afficher des chaînes de caractères à l'écran. Pour passer à la ligne, on peut utiliser `endl`. De même, `cin` est le flux d'entrée standard. Il est également possible d'écrire dans un fichier (cf la classe`fstream`).

  Ces deux flux sont définis dans la librairie `iostream`, donc un "`#include <iostream>;`" est nécessaire avant leur utilisation. La ligne "`using namespace std;`" est un espace de nom, et permet de signaler qu'on utilise des objets de la STanDard library [2].

- *Type bool:* Un type booléen a été définie (avec les mots clef `true` et `false`). Il est possible de convertir un entier en booléen (`0` correspondant à `false` et toutes les autres valeurs à `true`).

- *Déclaration:* Il est possible de faire ses déclarations de variables à n'importe quel points du programme (y compris à l'intérieur d'une boucle `for`, comme c'est le cas pour l'entier `i` ou le booléen `cond` dans la boucle).

- *Allocation:* `malloc` et `free` sont toujours utilisable, mais deux nouveaux mots-clefs apparaissent: `new` et `delete` (pour allouer et désallouer un objet). Ainsi, il est possible de faire:

[1] Si ce n'est pas le cas, signalez-le moi rapidement (je ne mords pas... enfin, sauf les nuits de pleine lune)

[2] Il est possible de s'affranchir de cette ligne, mais on est obligé d'utiliser `std::[nom de l'objet]` (comme "`std::cout`"), ce qui est très (trop?) verbeux

```
int *variable = NULL;
variable = new int;      // Allocation
// Programme utilisant le pointeur "variable"
delete variable;                 // Désallocation
```

De même, les tableaux peuvent être déclarés/alloués/désalloués avec `int* tab = NULL; tab = new int[42]; delete[] tab;`.

Après, la majeure partie du C++ est identique au C, donc vous ne devrez pas être trop dépaysé. Voilà quelques rappels au cas où.

- Au niveau pratique, les fichiers "`.h`" sont les fichiers headers et contiennent les signatures des fonctions/classes (équivalent des `.mli` en Caml). Les fonctions et méthodes correspondantes sont implémentées (sous réserve d'existence) dans le fichier "`.cc`" (équivalent des `.ml` en Caml).

- Si vous implémentez les fonctions correspondant à un `.h`, n'oubliez pas d'inclure au début de votre fichier `.cc` un "`#include "[Nom-de-l'header].h";`". Le compilateur fera une étape de pré-processing qui remplacera cet `include` par le contenu du fichier `.h` mentionné (afin d'avoir les déclarations des fonctions et de savoir de quoi vous parlez par la suite).

- Pour le débuggage, la solution "classique" consiste à insérer des "`cout «`" aux points stratégiques du programme, afin de surveiller l'évolution des variables via la sortie du terminal [3]. Une autre solution consiste à utiliser *gdb* afin d'exécuter un programme et d'examiner sa trace au moment du bug. L'intérêt de *gdb* est le repérage instantané du point dans le programme qui a planté, ainsi que sa capacité (partielle) à afficher les valeurs des variables à ce moment là [4].

  En guise de rappel, voilà la procédure à suivre pour faire marcher *gdb*:

  - Compiler le programme en utilisant l'option "-g" (inclus de base dans les `Makefile` fournis)
  - Lancez `gdb` sur l'exécutable (autre option: tapez `ulimit -c 1000000` pour générer un fichier core à la prochaine tentative d'exécution de programme et lancez `gdb [nom du fichier] [nom du core]`
  - Commandes de base: `help`, `quit`, `running`, `backtrace`, `frame` et `print`.

Exercice: Écrivez un programme qui:

- Demande à qui vous voulez dire bonjour (puis passe à la ligne)
- Récupère une chaîne de caractère rentrée par l'utilisateur
- Répond "Hello [la-dite chaîne de caractère]", puis re-passe à la ligne.

Exercice 2 : Classe et Objet

Quand on programme, on a tendance à regrouper dans un fichier toutes les fonctions qui manipulent une structure de donnée. Par exemple, voilà un exemple de fichier `List.h` en C:

```
typedef struct {...} list_t;     // Définition de la liste

// Fonctions sur la liste (séparé de la définition de type)
list_t* empty_list();
list_t* add(list_t*, int);

int get_nth(list_t , int);
int length(list_t*);
```

[3] D'où l'intérêt de faire des pretty-printers, sauf si vous tenez à savoir à quel adresse est stocké un objet
[4] Ce qui évite les recherches dichotomique (et fastidieuses) au "printf", mais est limité niveau pretty-printage

Le paradigme objet fourni un cadre syntaxique pour expliciter ce regroupement. Les fonctions (appelées *méthodes*) sont syntaxiquement rattachées à une variable (appelé *objet*) d'un certain type (appelé *classe*). Par exemple, pour ajouter un élément `2` à une liste `l1`, au lieu de devoir écrire `add(l1,2)`, on écrit `l1.add(2)`.

La classe `List` peut se déclarer de la façon suivante. Le fichier `List.h` contient généralement la définition de la classe:

```
Fichier List.h:

class List
{
 public:        // Ce qui suit est accessible par toute les fonctions
  // Attributs
  boolean is_empty;
  int head;
  List* tail;

  // Constructeurs (appelé via "new")
  List();
  List(List*,int);

  // Destructeur (appelé via "delete")
  ~List();

  // Méthodes
  void add(int e);
  int get_nth(int);
  int length();
};
```

Soit `l1` un objet de type `List`. Pour accéder à un de ses attributs, il suffit d'écrire "`l1.head`", par exemple. De même, pour invoquer une méthode sur l1, il suffit d'écrire "`l1.length()`". Les constructeurs et destructeurs doivent avoir le même nom que la classe et ne sont pas appelable via cette notation (mais via `new` et `delete`, respectivement).

Les implémentations des méthodes se trouvent dans `List.cc`:

```
#include "List.h"

List::List()
{ this->is_empty = true; }       // this: pointeur sur l'objet en construction

List::List(List* t, int h)
{ is_empty = false; tail = t; head = h; }

...
```

Ces 2 premières méthodes sont des *constructeurs*. Elles permettent de construire un objet de type/-classe `List` en mémoire. Elles s'utilisent de la manière suivante:

```
List* l1 = new List(); //[], appel au premier constructeur
List* l2 = new List(new List(),1); //[1]
```

`this` est un pointeur sur l'objet en cours de construction (càd, dans l'implémentation d'une méthode, `this` est un pointeur vers l'objet sur lequel la méthode est appelée [5]. Un constructeur retourne automatiquement `this` (l'objet qui est construit).

La flèche "`this->is_empty`" est un sucre syntaxique pour dénoter "`(*this).is_empty`". Cependant, si on veut accéder aux attributs de l'objet courant, au lieu d'écrire "`this->is_empty`", il est possible d'écrire tout simplement "`is_empty`" (comme c'est le cas dans le second constructeur).

[5] Par exemple, si on a `l1.add(2)`, `this` pointera sur `l1` dans l'implémentation de `add`

Il est possible de définir des opérateurs (+, -, * ,«, etc) sur des objets. Par exemple, écrivons un « sur nos `List` afin de pouvoir l'afficher avec la syntaxe `cout « l1` (de façon naturelle). Pour cela, il faut ajouter à la fin de `List.h` [6]:

```
ostream& operator<< (ostream& sout, List& l);
```

Et dans `List.cc`;

```
ostream& operator<< (ostream& sout, List& l) {
  if(l.is_empty) {
   sout << "[]";
   return sout;
  }
  sout << "[ << l.head << " " << *(l.tail); // Appel récursif sur la queue de la liste
  return sout;
}
```

Exercice:

- Recopier le code ci-dessus dans les fichiers `List.h` et `List.cc` respectivement. Finir d'implémenter les méthodes. Testez via un fichier `main.cc`

Exercice 3. *STL*

La STL (Standard Template Library) est une bibliothèque qui implémente une fois pour toutes les listes et les ensembles (ce qui évite de tout refaire à chaque fois). Les structures importantes sont `vector`, `map`, `pair` et `set` [7]

Voilà quelques idiomes à connaître (sans forcement comprendre le détail):

```
#include <iostream>      // Pour les flux
#include <vector>        // Pour la structure de vecteur
#include <map>           // Pour la structure de map

using namespace std;

int main(void)
{
  // Vector
  vector<int> vec;
  vec.push_back(7); // vec = [7]
  vec.push_back(5); // vec = [7,5]
  cout << vec.size << endl // "2"
       << vec[0] << " " << vec[1] << endl;  // "7 5"

  // Map
  map<string,int> age;
  age["toto"] = 25;              // age: ("toto" -> 25)
  age["titi"] = 8;               // age: ("titi" -> 8 ; "toto" -> 25)
  age["toto"] = 24;              // age: ("titi" -> 8 ; "toto" -> 24)
  if(age.find("toto") != age.end())              //toto existe?
    cout << "toto a " << age["toto"] << "ans" << endl;

  for(map<string,int>::iterator it = age.begin(); it != age.end(); ++it)
    cout << (*it).first << " -> " << (*it).second << endl;
}
```

Manip.

- Recopier le code et expérimentez.
- Si l'heure n'est pas encore écoulée, répondez à la question 3A du TP1 (qui est un excellent exercice de manipulation des `map` et des `pair`)

[6] sans oublier le `#include` à `iostream`

[7] Bonne page de référence sur la STL: http://www.cplusplus.com/reference/stl/