**Final**

Jeudi 30 Juin 2011, de 8h00 à 10h00

**Note : Documents cours et TD papier autorisés. La copie du voisin même papier, ne l'est pas.**

# 1. Le factor passe toujours deux fois (2 pts)

Expliquez la différence entre les deux écritures de factorielle suivantes. Si vous devez choisir une des deux écritures, justifiez votre choix.

```
template <int N>
inline double Factorielle() {
   return N * Factorielle<N-1>() ;
};
template <>
inline double Factorielle<0>() {
    return 1;
};
```

```
template <int N>
struct Factorielle1 {
   static const double VAL= N *
Factorielle1<N-1>::VAL ;
};
template <>
  struct Factorielle1<0> {
    static const double VAL= 1;
};
```

# 2. Toujours plus (6 pts)

**1.** Ecrivez une fonction d'addition `int Add(int X, int Y)` qui réalise la somme de deux valeurs entières quelconques, ceci en utilisant uniquement l'incrémentation, la décrémentation ou l'addition d'une unité.

**2.** Cette fonction doit être utilisée uniquement pour ajouter à une valeur X une valeur Y constante. Nous voulons éviter les boucles ou les appels récursifs à l'exécution pour améliorer l'efficacité. Nous utilisons donc le inlining.

Modifiez votre fonction en ce sens et expliquez comment vérifier si cette demande est prise en compte ?

**3.** Après vérification nous voyons que le compilateur refuse de prendre en compte cette demande. Nous allons utiliser les fonctions templates pour répondre à notre besoin.
Ecrivez une solution avec les fonctions templates permettant une utilisation par `AddF<3>(x)` où 3 est la constante à ajouter à *x*.

**4.** Nous voulons essayer par les classes templates.

Nous souhaitons avoir un appel similaire à la forme précédente : `AddT<3>(x)` .

Le modèle général sera :

```
template <int Y>                    // Y = constante
struct AddT{
  int operator()(int X) {  …        // X = variable
  }
```

```
};
```

Complétez ce template pour obtenir le résultat souhaité.

**5.** A la compilation nous obtenons une erreur :

```
59   template <int Y>
60   struct AddT{
61   int operator()(int X) {

     …

         cout<< "valeur de AddT<3>(5)="
159           << AddT<3>(5) <<endl;
```

```
g++ -O3 -Wall -c -fmessage-length=0 -osrc\metaprogrammation.o ..\src\metaprogrammation.cpp
..\src\metaprogrammation.cpp: In function `int main()':
..\src\metaprogrammation.cpp:159: error: no matching function for call to `AddT<3>::AddT(int)'
..\src\metaprogrammation.cpp:60: note: candidates are: AddT<3>::AddT()
..\src\metaprogrammation.cpp:60: note:                 AddT<3>::AddT(const AddT<3>&)
```

Sachant que la définition de AddT ne contient que la définition de l'opérateur fonctionnel, expliquez cette erreur et indiquez à quoi correspondent les deux fonctions candidates.
Proposez une solution pour résoudre cette erreur d'appel.

**6.** La forme de notre appel ne nous convient pas. Nous souhaitons que l'utilisateur puisse écrire AddTF(x, 5).

Expliquez pourquoi ce qui suit n'est pas accepté par le compilateur. L'erreur est située sur l'appel Vous supposerez que l'appel "`AddT<Y>(X)`" a été corrigé.

```
inline int AddTF(int X, const int Y) {
    return Corrigé{AddT<Y>(X)};   // corrigé de l'appel en 2.5
}
```

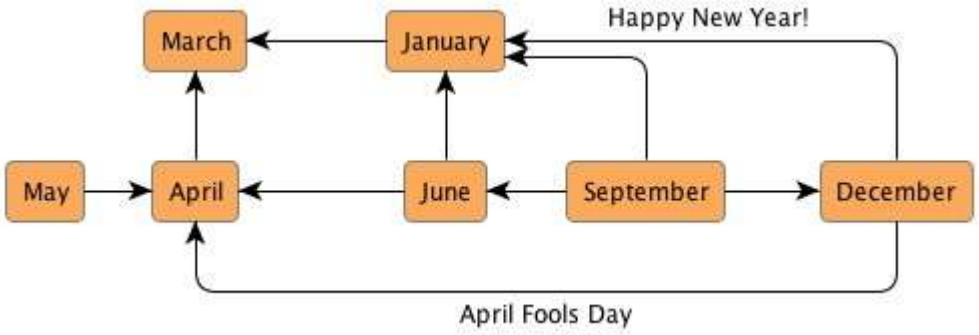
Proposez une solution sur la base des macros du pré-processeur.

# 3. L'esprit du graphe (6 pts)

**TGF :** "**T**rivial **G**raph **F**ormat," permet de définir une structure de graphe avec au plus un nom par élément du graphe. Les autres informations (position, dimension, forme) sont ignorées. TGF ne supporte pas non plus l'imbrication de graphes.

La description TGF ci-dessous à gauche permet de représenter le graphe à droite.

```
1 January
2 March
3 April
4 May
5 December
6 June
7 September
#
1 2
3 2
4 3
5 1 Happy New Year!
5 3 April Fools Day
6 3
6 1
7 5
7 6
7 1
```




Note : 1 2 signifie qu'il existe un arc de sommet 1 au sommet 2

Voici un squelette de grammaire SPIRIT :

```
#include <boost/spirit/include/qi.hpp>
#include <boost/fusion/include/adapt_struct.hpp>

namespace fusion = boost::fusion;
namespace phoenix = boost::phoenix;
namespace qi = boost::spirit::qi;
namespace ascii = boost::spirit::ascii;

namespace client {
}

BOOST_FUSION_ADAPT_STRUCT(
)

using namespace std;
using boost::spirit::unused_type;

typedef unused_type TypeResult;

template <typename Iterator>
struct gml : qi::grammar<Iterator,TypeResult,ascii::space_type> {
    gml() : gml::base_type(graph) {
        using qi::lit;
        using qi::_1;
        using ascii::char_;
        using qi::int_;
        using qi::eps;

        graph = eps;

    qi::rule<Iterator, TypeResult(),ascii::space_type> graph;
};
```

1. Ecrivez une grammaire TGF permettant d'analyser dans un texte de la forme identique à celui de l'exemple, la partie déclaration des sommets.

Note : Dans un premier temps vous ne tenez pas compte des possibles informations supplémentaires.

**2.** Complétez pour intégrer la partie déclaration des arcs

-lexeme[*(alpha|'_'|' ')]; il faudra utiliser omit ou intégrer un
// optional dans la structure d'accueil question 4Modifiez la grammaire pour accepter et ignorer d'éventuelles informations entre '[' et ']', après le nom d'un sommet.

**4.** Nous voulons récupérer, à la suite de l'analyse, une structure de données contenant les informations typées. Quelle peut être la forme de cette structure ? Modifiez la grammaire en conséquence.

# 4. Bi_map l'outsider. (8 pts)

Nous avons vu que la définition d'une bi_map (utilisation de paires où chaque donnée peut être la clé ou la valeur) posait des problèmes au niveau de l'utilisation de l'opérateur d'indexation. En effet, l'utilisation de cet opérateur sur une map restitue un accès à une paire constituante de la map. Ceci quelque soit l'état de la map c'est-à-dire présence ou absence de la clé utilisée pour l'accès. Il y a création avec une valeur par défaut si la clé n'existe pas.
Ce fonctionnement ne convient pas, dans le cas d'une bi_map car dans ce cas la valeur par défaut peut être associée à plusieurs clés. Ce que nous ne souhaitons pas.
D'autre part l'utilisation de  l'opérateur sur une map donne accès à la référence de la valeur, laquelle peut être changée. Dans notre cas, cette nouvelle valeur peut déjà exister dans notre bi_map d'où un problème identique que précédemment.

Une solution est d'interdire cet opérateur pour la bi_map.
Une autre solution est de ne pas fournir la référence à la valeur associée à la clé, mais de retourner un objet spécifique.
Cet objet fournit :

- un opérateur d'affectation contrôlant l'existence de la clé et d'une valeur associée.
  En cas d'existence on choisit de changer la valeur et de détruire une autre association éventuelle de cette valeur avec une autre clé.
- Un opérateur de cast permettant de fournir la valeur associée en cas d'utilisation de l'objet en tant que lvaleur (à droite d'une affectation)

Bien sûr, le type de l'objet retourné par l'opérateur d'affectation dépendra du type de la clé utilisée.

Pour une bi_map<char, int> mci :

- mci['a'] retourne un objet travaillant avec un entier
- mci[3]  retourne un objet travaillant avec un caractère

Voici un exemple d'utilisation :

| **Code** | **Sortie** |
|---|---|
| `bi_map<int, char> bic ;` | |
| `bic[2]='E';` | |
| `bic[3]= 'b';` | |
| `bic['b']=5; // destruction de (3, 'b')` | |
| `std::cout<< bic[5]<<std::endl;` | `b` |
| `std::cout<< bic['b']<<std::endl;` | `5` |
| `std::cout<<bic['b']+4.5<<std::endl;` | `9.5` |
| `std::string s("essai chaine");` | |
| `s.replace(s.begin(),s.begin()+1, 1, bic[2]);` | |
| `std::cout<<s<<endl;` | `Essai chaine` |

Voici un squelette simplifié de définition d'une bi_map

```
template<class T, class U>
class bi_map{
       typedef std::pair<T,U> *Tdonnee;        // pt sur  association de données de base
       typedef typename std::map<T, Tdonnee> Tm1;
       typedef typename std::map<U, Tdonnee> Tm2;
       typedef typename Tm1::iterator Tit1;
       typedef typename Tm2::iterator Tit2;

private :
       Tm1 m1;              // map pour accès par un index de type T
       Tm2 m2;              // map pour accès par un index de type U

       class CTtoU{
              T index;
              bi_map<T, U> *ref;  // la structure à modifier éventuellement
         public:
              CTtoU(T t, bi_map *bm);
              inline operator U();
              inline bi_map &operator=(U val);
       };
       class CUtoT{
              U index;
              bi_map<T, U> *ref;
         public:
              CUtoT(U u, bi_map *bm) ;
              inline operator T();
              inline bi_map<T, U> &operator=(T val) ;  // marche aussi
       };
public :
       …
       inline CTtoU &operator[](T c){return *new CTtoU(c, this);}
       inline CUtoT &operator[](U c){return *new CUtoT(c, this);}
       …
};
```

1. Ecrivez les implantations du constructeur et de l'opérateur cast de CTtoU. Ce dernier gèrera le cas de la non existence d'une valeur.

Une rapide étude met en évidence que les classes CTtoU et CUtoT sont très semblables. Où CTtoU travaille avec T, U, m1, m2, first et second, CUtoT travaille respectivement avec U, T, m2 , m1, second et first.
Nous décidons donc de définir un template pour les deux classes :

```
template <class T, class U, class K>
class CXtoY ;
```

Pour implanter celui-ci nous allons définir un modèle de conversion `paire_get`

```
template <class T, class U, class K>
struct paire_get ;
```

**Note :** nous choisirons d'optimiser le code si le compilateur le permet.

2. Implantez le template de manière à ce que celui-ci nous renseigne sur le type de l'index (`type_cle`) et le type de la valeur (`type_val`).

3. Ajouter à votre template une fonction permettant de fournir la valeur associée à une clé dans une paire. La paire et la clé seront transmises en paramètres.

**4.** La recherche d'une paire se fera avec la map m1 ou la map m2 en fonction du type de l'index utilisé. Définissez dans le template deux fonctions `map_cle(bi_map<T,U> &bm)` et `map_val(bi_map<T,U> &bm)` permettant d'extraire de la bi_map respectivement les références de la map associée à l'index et de la map associée à la valeur.

**5.** Nous souhaitons créer une paire de base pour une bi_map<int, char>. Que l'on utilise `bic[2]='a'` ou `bic['a']=2`, la paire créée doit être (2, 'a'). Complétez votre template de façon à fournir une fonction de création de la paire `std::pair<int,char> *toPaire(T1 c, T2, v)`.

**6.** Pour des raisons de facilité de développement nous avons défini ces templates en dehors de la classe bi_map. Assurez vous que la compilation soit possible.

**7.** Evidemment une ***bi_map<int, int>*** ne peut pas fonctionner. Ajoutez un mécanisme permettant, dans un tel cas, de provoquer une erreur de compilation et d'afficher une information circonstanciée.