

Solutions aux exercices du chapitre 13 :
TBB

Automne 2017

Exercice 13.1 Passage par valeur vs. par référence.

1. Pour chacun des programmes C++ ci-bas, indiquez ce qui sera affiché si on compile (avec g++) puis on exécute.

Dans tous les cas, on suppose qu'une instruction «`#include <cstdio>`» est présente au début du fichier.

2. Pour le programme `pgm5.cpp`, est-ce que les deux versions de la fonction `incinc` produisent toujours le même effet?
3. En FORTRAN, tous les paramètres sont toujours passés par référence.

Qu'est-ce qui était imprimé en FORTRAN-77 par le programme suivant — dans l'appel à `PRINT`, «*» dénote la sortie standard (*stdout*) :

```
FUNCTION inc( x )  
    x = x + 1  
END  
  
inc( 0 )  
  
PRINT *, "0 = ", 0
```

Solution :

pgm1.cpp

```
0
1
```

pgm2.cpp

```
1
2
```

pgm3.cpp

```
pgm3.cpp: In function 'int main(int, char**)':
pgm3.cpp:11:10: erreur:
      call of overloaded 'inc(int&)' is ambiguous
    inc( x ); printf( "%d\n", x );
      ^

pgm3.cpp:11:10: note: candidates are:
pgm3.cpp:3:6: note: void inc(int)
    void inc( int x ) { x += 1; }
      ^

pgm3.cpp:5:6: note: void inc(int&)
    void inc( int& x ) { x += 1; }
      ^

pgm3.cpp:13:10: erreur:
      call of overloaded 'inc(int&)' is ambiguous
    inc( x ); printf( "%d\n", x );
      ^

pgm3.cpp:13:10: note: candidates are:
pgm3.cpp:3:6: note: void inc(int)
    void inc( int x ) { x += 1; }
      ^

pgm3.cpp:5:6: note: void inc(int&)
    void inc( int& x ) { x += 1; }
      ^
```

pgm4.cpp

```
pgm4.cpp: In function 'int main(int, char**)':
pgm4.cpp:9:10: erreur: invalid initialization of non-const
      reference of type 'int&' from an rvalue of type 'int'
      inc( 1 ); printf( "%d\n", x );
          ^
pgm4.cpp:3:6: erreur: in passing argument 1 of 'void inc(int&)'
      void inc( int& x ) { x += 1; }
          ^
```

pgm5.cpp

```
10 10
10 10
```

Non, les deux versions **ne produisent pas toujours le même effet**, par exemple, en présence **d'alias** (synonymes, i.e., lorsque plusieurs identificateurs réfèrent au même objet, au même espace mémoire) :

```
x = 0;
incinc( &x, &x, 10 ); printf( "%d %d\n", x, x );
// 10 10

x = 0;
incinc( x, x, 10 ); printf( "%d %d\n", x, x );
// 20 20
```

Le programme FORTRAN-77 imprimait «0 = 1».

Exercice 13.2 Une utilisation non-triviale — et quelque peu étonnante! — des templates.

Que fait le programme ci-bas?

```
#include <string>

template <int N>
struct Foo
{
    enum { val = N * Foo<N-1>::val };
};

template <>
struct Foo<0>
{
    enum { val = 1 };
};

int main( int argc, char *argv[] )
{
    const int N = 10;
    printf( "%d => %d\n", N, Foo<N>::val );
    return 0;
}
```

```
-----

$ /usr/bin/g++ -m64 -ltbb -std=c++11 foo.cpp -o foo
$ foo
10 => 3628800
```

Solution :

Voici le même programme, mais cette fois avec un identificateur plus significatif ☺

```
#include <string>

template <int N>
struct Factoriel
{
    enum { valeur = N * Factoriel<N-1>::valeur };
};

template <>
struct Factoriel<0>
{
    enum { valeur = 1 };
};

int main( int argc, char *argv[] )
{
    const int N = 10;
    printf( "%d => %d\n", N, Factoriel<N>::valeur );
    return 0;
}
```

```
-----

$ /usr/bin/g++ -m64 -ltbb -std=c++11 factoriel.cpp -o factoriel
$ factoriel
10 => 3628800
```

Voici une version révisée du programme principal :

```
int main( int argc, char *argv[] )
{
    int N = 10; // const a ete supprime!
    printf( "%d => %d\n", N, Factoriel<N>::valeur );
    return 0;
}
```

```
-----

$ /usr/bin/g++ -m64 -ltbb -std=c++11 factoriel.cpp -o factoriel
/usr/bin/g++ -m64 -ltbb -std=c++11 factoriel.cpp -o factoriel
factoriel.cpp: In function 'int main(int, char**)':
factoriel.cpp:18:38: erreur: the value of 'N' is not usable
                    in a constant expression
    printf( "%d => %d\n", N, Factoriel<N>::valeur );
                                   ^
factoriel.cpp:17:7: note: 'int N' is not const
    int N = 10;
    ^
factoriel.cpp:18:39: erreur: the value of 'N' is not usable
                    in a constant expression
    printf( "%d => %d\n", N, Factoriel<N>::valeur );
                                   ^
factoriel.cpp:17:7: note: 'int N' is not const
    int N = 10;
    ^
factoriel.cpp:18:39: note: in template argument for type 'int'
    printf( "%d => %d\n", N, Factoriel<N>::valeur );
```

Et une autre version — sans le cas de base N=0 :

```
#include <string>

template <int N>
struct Factoriel
{
    enum { valeur = N * Factoriel<N-1>::valeur };
};

int main( int argc, char *argv[] )
{
    const int N = 10;
    printf( "%d => %d\n", N, Factoriel<N>::valeur );
    return 0;
}
```

```
-----
$ g++ factoriel.cpp -o f
factoriel.cpp:6:23: fatal error: recursive template instantiation
exceeded maximum depth of 256
    enum { valeur = N * Factoriel<N-1>::valeur };
                        ^
factoriel.cpp:6:23: note: in instantiation of template class
'Factoriel<-246>' requested here
    enum { valeur = N * Factoriel<N-1>::valeur };
                        ^
[...]
factoriel.cpp:6:23: note: (skipping 247 contexts in backtrace; use -ftemplate-backt
[...]
factoriel.cpp:12:28: note: in instantiation of template class
'Factoriel<10>' requested here
    printf( "%d => %d\n", N, Factoriel<N>::valeur );
                        ^
factoriel.cpp:6:23: note: use -ftemplate-depth=N to increase recursive
template instantiation depth
    enum { valeur = N * Factoriel<N-1>::valeur };
                        ^
1 error generated.
```

Note : Le site Web suivant montre comment on peut définir, uniquement avec des *templates* C++, **un interpréteur de λ -calcul**, illustrant ainsi que les *templates* C++ sont *Turing-complete*!

<http://matt.might.net/articles/c++-template-meta-programming-with-lambda-calculus/>

Exercice 13.3 Capture par valeur vs. par référence.

Pour chacun des segments de code C++ ci-bas, indiquez ce qui sera affiché si on compile (avec g++) puis on exécute.

1. Capture par référence d'un tableau statique :

```
int a[n];
auto init_zero = [&](int i) { a[i] = 0; };
init_zero(0);
printf( "%d\n", a[0] );
```

2. Capture par valeur d'un tableau statique :

```
int a[n];
auto init_zero = [=](int i) { a[i] = 0; };
init_zero(0);
printf( "%d\n", a[0] );
```

3. Capture par valeur d'un tableau passé en argument :

```
void init_zero(int a[], int i) {
    [=](int i){ a[i] = 0; }( i );
}

int a[n];
init_zero(a, 0);
printf( "%d\n", a[0] );
```

4. Capture par valeur d'un tableau dynamique :

```
int* a = (int*) malloc(n * sizeof(int));
auto init_zero = [=](int i) { a[i] = 0; };
init_zero(0);
printf( "%d\n", a[0] );
```

Exercice 13.4 Problème de l'histogramme.

On veut paralléliser la fonction `histogramme` présentée plus haut (Programme C 13.2), fonction qui produit un histogramme pour les entiers du tableau `elems`.

En utilisant les *Threading Building Blocks* d'Intel, écrivez une version parallèle de la fonction `histogramme`. Utilisez une approche de **parallélisme de résultat** — même si cela implique de parcourir plusieurs fois la matrice `elems`.

- Parallélisez tout d'abord `histogramme`. Ensuite, parallélisez `nb_occurrences!`
-

Solution :

Fonction histogramme parallélisée :

```
int* histogramme( int elems[], int nb, int valMax )
{
    // On alloue et on initialise l'histogramme.
    int *histo = (int*) malloc( (valMax+1) * sizeof(int) );

    parallel_for( blocked_range<size_t>(0, valMax+1),
        [=]( blocked_range<size_t> r ) {
            for( size_t val = r.begin(); val < r.end(); val++ ) {
                histo[val] = nb_occurrences( val, elems, nb );
            }
        }
    );

    return histo;
}
```

Fonction nb_occurrences parallélisée :

```
int nb_occurrences( int val, int elems[], int nb )
{
    return parallel_reduce(
        blocked_range<size_t>(0, nb),
        0,
        [=]( blocked_range<size_t> r, int nb_occs ) {
            for( size_t k = r.begin(); k < r.end(); k++ ) {
                if ( elems[k] == val ) { nb_occs += 1; }
            }
            return nb_occs;
        },
        std::plus<int>()
    );
}
```

Exercice 13.5 Programme mystère.

Que fait la fonction suivante?

```
int* mystere( int elems[], int nb, int vm )
{
    auto init = [vm]() {
        int* h0 = (int*) malloc( (vm+1) * sizeof(int) );
        for( int k = 0; k <= vm; k++ ) {
            h0[k] = 0;
        }
        return h0;
    };

    auto foo = [=]( blocked_range<size_t> r,
                   int* h ) {
        for( auto i = r.begin(); i < r.end(); i++ ) {
            h[elems[i]] += 1;
        }
        return h;
    };

    auto bar = [vm]( int* h1, int* h2 ) {
        for( int k = 0; k <= vm; k++ ) {
            h1[k] += h2[k];
        }
        return h1;
    };

    return
        parallel_reduce( blocked_range<size_t>(0, nb),
                        init(),
                        foo,
                        bar
                        );
}
```

Solution :

Produit, comme dans l'exercice précédent, un histogramme de façon parallèle, mais uniquement avec `parallel_reduce`.

Chaque appel à `foo` sur une tranche génère un histogramme pour cette tranche de valeurs à traiter. Ces divers histogrammes sont ensuite combinés avec `bar` — `init()` génère un histogramme nul, avec 0 partout comme nombre d'occurrences.

Exercice 13.0 Passage des arguments en Java.

Quel mécanisme est utilisé pour le passage des arguments en Java?

Indice : Voici quelques mécanismes utilisés dans divers langages de programmation :

- Par valeur (*copy-in*) ;
- Par résultat (*copy-out*) ;
- Par valeur/résultat (*copy-in/copy-out*) ;
- Par référence ;
- Par nécessité (*call-by-need*) ;
- Par nom (*call-by-name*) ;

	Val.	Rés.	Val./Rés.	Réf.	<i>Need</i>	Nom
ALGOL-60	X					X
FORTRAN				X		
Pascal	X		X			
C	X					
C++	X			X		
Ada	X	X	X			
Haskell					X	

Solution :

Uniquement passage **par valeur**, tel qu'illustré par l'exécution du programme ci-bas.

```
class Reference {
    static void reset( int x )
    { x = 0; }

    static void reset( Integer x )
    { x = null; }

    public static void main( String[] args ) {
        int x = 999;
        reset( x );
        System.out.println( "x = " + x );

        Integer y = 999;
        reset( y );
        System.out.println( "y = " + y );
    }
}
```

```
$ javac Reference.java
$ java Reference
x = 999
y = 999
```
