TD 1

Fonctions, Pointeurs, Références et Allocation dynamique

Ce premier TD porte sur des fonctionnalités de base du langage C++ (et sur leur différence par rapport au C), en particulier, il permet de :

- se (re)familiariser avec les espaces de nom (namespace) et avec la bibliothèque `<iostream>` pour afficher des messages dans la console via `std::cout`
- se familiariser avec la surdéfinition de fonctions
- comprendre la différence entre le passage par valeur et par référence (ou adresse)
- se familiariser avec les opérateurs `new[]` et `delete[]` du C++ pour l'allocation dynamique de mémoire
- réviser la construction de la boucle `for`
- se familiariser avec l'utilisation de la `std::vector`

Commençons par créer un fichier *main.cpp* qui va contenir le code du TD1. Pour l'affichage dans la console, nous avons besoin de `std::cout` qui est défini dans le header `<iostream>` :

```
#include <iostream>
```

Toutes les fonctions et objets de la bibliothèque standard font partie de l'espace de nom (*namespace*) `std`. La directive `using namespace std;` permet d'omettre le préfixe `std::` quand on accède aux fonctions de la libraire standard. On peut alors écrire `cout << "Hello" << endl;` au lieu de `std::cout << "Hello" << std::endl;`

1.1 Suite de Fibonacci

La suite de Fibonacci est une suite d'entiers devant son nom au mathématicien italien du XIII$^{\text{ème}}$ siècle Leonardo Fibonacci qui, dans un problème récréatif, décrit la croissance d'une population de lapins en ces termes

> *Un homme met un couple de lapins dans un lieu isolé de tous les côtés par un mur. Combien de couples obtient-on en un an si chaque couple engendre tous les mois un nouveau couple à compter du troisième mois de son existence ?*

Ce problème est à l'origine de la suite dont le $n^{\text{ième}}$(terme correspond au nombre de paires de lapins au $n^{\text{ième}}$ mois. En notant $\mathcal{F}_n$ le nombre de couples de lapin au début du mois n et en posant $\mathcal{F}_1 = \mathcal{F}_2 = 1$, Fibonacci déduit ainsi la relation de récurrence suivante :

$$\begin{aligned} \mathcal{F}_1 &= 1 \\ \mathcal{F}_2 &= 1 \\ &\dots \\ \mathcal{F}_n &= \mathcal{F}_{n-2} + \mathcal{F}_{n-1} \end{aligned}$$

à savoir que le nombre de couples de lapin au mois n est égal à la somme des couples de lapins lors des deux moins précédents.

1. Écrire une fonction calculant le nombre de Fibonacci au $n^{\text{ième}}$ terme, de façon itérative (avec des variables d'état et une boucle). Cette fonction prendra comme argument et utilisera des entiers courts non signé `unsigned short`. Son prototype sera `unsigned short fibonacci(unsigned short)`. Par défaut, cette fontion fournira le nombre de Fibonacci pour $n = 10$.
2. En outre, vous déclarerez une deuxième fonction `fibonacci` prenant comme argument et utilisant des entiers non signé `unsigned int`.
3. Tester l'appel de chacune des deux fonctions en changant le type de l'argument fourni (utiliser un *cast*, par exemple `(unsigned short)42` ; ou passer des variables de type différent). À partir de quelle valeur numérique de `n` la valeur retournée diffère-t-elle ? Pourquoi ? Supprimer la version `unsigned short`, qui ne sera plus nécessaire.

Correction

Surdéfinition de fonctions

Le code ci-dessous implémente deux fonctions, toutes deux nommées `fibonacci` mais dont la déclaration diffère en terme de type d'arguments et de type de valeur retournée [a] :

```
1  #include <iostream>
2  using namespace std;
3
4  unsigned short fibonacci(unsigned short n = 10) {
5    if (n <= 2) return 1;
6    unsigned short fn1 = 1, fn2 = 1;
7    unsigned short fn = fn1 + fn2;
8    for (unsigned short i = 2; i < n; i++) {
9      fn = fn1 + fn2;
10     fn1 = fn2;
11     fn2 = fn;
12   }
13   return fn;
14 }
15
16 unsigned int fibonacci(unsigned int n) {
17   if (n <= 3) return 1;
18   unsigned int fn1 = 1, fn2 = 1;
19   unsigned int fn = fn1 + fn2;
20   for (unsigned int i = 2; i < n; i++) {
21     fn = fn1 + fn2;
22     fn1 = fn2;
23     fn2 = fn;
24   }
25   return fn;
26 }
27
28 void main() {
29   const unsigned short n = 20;
30   cout << "F(n_defaut) = " << fibonacci() << endl;
31   cout << "F(" << n << ") = " << fibonacci(n) << endl;
32   cout << "F(" << n << ") = " << fibonacci((unsigned int)n) << endl;
33 }
```

L'appel à la fonction `fibonacci` à la ligne 36 convertit, ou *cast*, la valeur de `n` en un entier non signé afin d'explicitement faire appel à la seconde fonction `unsigned int fibonacci(unsigned int)` définie à la ligne 17. Par défaut, seule la première des deux fonctions calculent la suite de Fibonacci au terme $n = 10$, cette valeur étant précisée dans le prototype des fonctions. Il est possible de préciser une valeur par défaut pour la seconde fonction mais il y aura alors une erreur de compilation lorsque l'utilisateur souhaitera appeler la fonction `fibonacci` sans préciser l'ordre : le compilateur indiquera une ambiguïté dans l'appel de la fonction car le contexte d'appel ne lui permettra plus de distinguer quelle fonction appeler.
La suite de Fibonacci divergeant rapidement, il est important de noter qu'en fonction du type retourné le résultat de ces deux fonctions ne sera plus valide à partir d'un certain ordre. Ainsi dès le $25^{\text{ème}}$ ordre, celle retournant un `unsigned short` donnera un résultat erroné ($\mathcal{F}_{25} = 75025$ car les valeurs du type `unsigned short` sont comprises entre 0 et $2^{16}-1$ *cf.* table 1.1).

Correction

Type	Taille en mémoire	Intervalle de valeurs
int	4 octets	-2 147 483 648 à 2 147 483 647
unsigned int	4 octets	0 à 4 294 967 295
short	2 octets	-32 768 à 32 767
unsigned short	2 octets	0 à 65 535

TABLE 1.1 – **Taille en mémoire et intervalle de valeurs de certains types entiers du C/C++**

a. nous évoquerons à la fin de ce cours comment la notion de *template* de fonction peut grandement simplifier l'écriture de ces deux fonctions.

4. Au sein du programme principal, réaliser $N \gg 1$ fois le calcul de la fonction de Fibonacci (pour un n donné) et estimer le temps d'exécution nécessaire au programme *via* la commande unix `time ./nom_executable`. Mesurer ce temps pour quelques valeurs de n. Quelle est la dépendance du temps d'exécution à n ?
5. Concevoir et implémenter une fonction `rfibonacci(unsigned int)` proposant une solution récursive au calcul de la suite de Fibonacci (c'est-à-dire sans boucle, implémentant directement la relation de récurrence). De même, mesurer le temps d'exécution pour la même valeur que `n`. Pourquoi est-il différent ? De même, quelle est sa dépendance à n ? Il est probable que votre programme plante pour des valeurs même modestes de n. Tentez de donner une explication.
6. Déporter la fonction récursive `rfibonacci`, qu'on aura renommée en `fibonacci`, dans une paire de fichiers : un premier fichier d'en-tête nommé `fibonacci.h` comprenant la déclaration (prototype) de la fonction et un second fichier source `fibonacci.cpp`, implémentant cette fonction. Modifier le programme principal afin d'utiliser cette fonction recursive et commenter provisoirement la fonction `fibonacci` itérative. Compiler et tester l'exécution.
7. Finalement, décommenter la version itérative de `fibonacci` *i.e.* celle contenue dans le fichier source principal et définie au début de cette exercice. Chercher à résoudre l'erreur de compilation en utilisant un espace de nom `espace_local` et tester le programme en forçant le choix de fonction à utiliser *via* l'opérateur de portée `espace_local::`.

Correction

Estimation du temps d'exécution d'un programme

Afin d'estimer le temps d'exécution de la fonction `fibonacci`, nous modifions le programme principal de telle sorte à réaliser un grand nombre de fois le calcul de la suite de Fibonacci.

```
1  void main() {
2    for (unsigned int i = 0; i < 1000000; i++) {
3      fibonacci(20);
4    }
5  }
```

On notera la déclaration de la variable `i` au sein de la boucle `for` comme nous y autorise le C++ . L'objectif ici est uniquement d'utiliser du temps processeur de l'ordinateur et d'obtenir une valeur moyenne du temps d'exécution de la fonction `fibonacci`. Pour connaître le temps d'exécution d'une

commande Unix, nous utilisons le programme `time` : ainsi, dans le terminal nous exécutons successivement les commandes de compilation

```
$  g++ fibonacci.cc -o fibonacci.exe
```

puis d'exécution en préfixant le nom du binaire par la commande `time`

```
$  time ./fibonacci.exe
real 0m0.007s
user 0m0.004s
sys 0m0.000s
```

On obtient ainsi trois temps qui correspondent respectivement à

`real` est le temps réel d'exécution de la commande *i.e.* l'intervalle de temps entre le moment où est lancée la commande et l'instant où le programme s'arrête.

`user` est le temps pendant lequel la commande utilise le processeur indépendamment d'autre processus (navigateur internet, lecture/écriture disque, ...) s'exécutant sur la machine. C'est ce temps (divisé par 1000000) qu'il faut considérer pour estimer le temps d'exécution de la fonction `fibonacci`.

`sys` est le temps pris par le système pour gérer la tâche.

Fonction récursive `rfibonacci`

Une fonction récursive est par définition une fonction qui s'appelle, le risque évident étant l'appel infini si les conditions de sortie de la fonction ne sont dûment remplies. Dans le cas de l'implémentation d'une suite récursive comme la suite de Fibonacci, une fonction récursive est relativement facile à écrire. Ainsi, la fonction `rfibonacci` s'écrit

```
unsigned int rfibonacci(unsigned int n) {
  if (n <= 2) return 1 ;
  else return rfibonacci(n-2) + rfibonacci(n-1);
}
```

On note bien que le corps de la fonction `rfibonacci` réalise deux appels à cette même fonction. La condition de sortie se fait dans l'hypothèse où `n` est inférieur à 2 auquel cas la valeur retournée sera 1. On imagine bien qu'à cause du double appel, le temps d'exécution sera exponentiel. En effet, la complexité asymptotique est en $\mathcal{O}(\Phi^n)$, où $\Phi = \frac{\sqrt{5}+1}{2} \approx 1.62$ est le nombre d'or. Alors qu'elle était clairement linéaire ($\mathcal{O}(n^1)$) pour la version itérative.

Il est possible de condenser encore plus l'écriture de cette fonction :

```
unsigned int rfibonacci(unsigned int n) {
  return n <= 2 ? 1 : rfibonacci(n-2) + rfibonacci(n-1);
}
```

Compilation séparée

Le fichier d'en-tête nommé `fibonacci.h` contient la déclaration (prototype) de la fonction

```
#ifndef _FIBO_H_
#define _FIBO_H_
unsigned int fibonacci (unsigned int n); // ici le nom de l'argument est facultatif
#endif
```

et le fichier source `fibonacci.cc`, contient la définition de la fonction.

```
#include "fibonacci.h"  // inclusion du prototype de la fonction
unsigned int fibonacci (unsigned int n) {
  if (n <= 2) return 1 ;
  else return rfibonacci(n-2) + rfibonacci(n-1);
}
```

Pour pouvoir utiliser cette fonction au sein du programme principal, il faut donc que le compilateur ait connaissance de la fonction au niveau du bloc `main`, et de son prototype pour savoir si son utilisation est correctement faite lors de son appel. On doit ainsi inclure de nouveau le fichier `fibonacci.h` dans le fichier contenant le bloc `main`.
Le programme principal modifié est le suivant (on a supprimé la première fonction `fibonacci` prenant en argument un `unsigned short`, la deuxième fonction `fibonacci`) :

```
#include "fibonacci.h" //inclusion de la fonction recursive
#include <iostream>
using namespace std;

//unsigned int fibonacci(unsigned int n) {
// ...
//}

void main() {
  const unsigned int n = 20;
  cout << "version récursive : F(" << n << ") = " << fibonacci(n) << endl;
}
```

Pour pouvoir compiler l'ensemble et lier le programme principal à l'implémentation de la fonction `fibonacci` contenu dans le fichier `fibonacci.cc`, il faut alors compiler les deux fichiers sources *via* la commande

```
$  g++ fibonacci.cpp main.cpp -o test.exe
```

Note : les fichiers d'en-tête ne doivent **jamais** figurer en tant que fichier à compiler. Cette opération en plus d'être une source éventuelle de problème [a] est totalement superflue puisque les fichiers d'en-tête sont, par inclusion, présents dans les fichiers sources.
Si l'implémentation de la fonction `fibonacci` n'est pas amenée à évoluer et ne suppose donc pas de recompilation du fichier `fibonacci.cpp`, on peut alors créer le fichier objet (code machine) et ne réaliser que l'opération d'édition de liens. La création d'un fichier objet se fait *via* la commande

```
$  g++ -c fibonacci.cpp
```

qui créera un fichier `fibonacci.o`.
Ensuite, on compile le `main` et génère l'exécutable en utilisant la commande

```
$  g++ fibonacci.o main.cpp -o test.exe
```

Correction

Utilisation d'espace de nom

Si deux fonctions portent le même nom et que leur contexte d'appel ne permet pas au compilateur de les différencier, on peut alors user d'un espace de nom pour distinguer chacune d'elle. Ainsi, en supposant que nous souhaitions faire cohabiter deux fonctions de calcul de suite de Fibonacci – la version récursive et la première version proposée dans ce TD –, il faut alors encapsuler la déclaration et la définition de ces fonctions dans des espaces de nom distinct. On pourra ainsi écrire le code suivant

```
#include "fibonacci.h" //inclusion de la fonction recursive
#include <iostream>
using namespace std;

namespace espace_local {

  unsigned int fibonacci(unsigned int n) {
    if (n <= 2) return 1;
    unsigned int fn1 = 1, fn2 = 1;
    unsigned int fn = fn1 + fn2;
    for (unsigned int i = 2; i < n; i++) {
      fn = fn1 + fn2;
      fn1 = fn2;
      fn2 = fn;
    }
    return fn;
  }

}

void main() {
  const unsigned int n = 20;
  cout << "version récursive : F(" << n << ") = " << fibonacci(n) << endl;
  cout << "version locale : F(" << n << ") = " << espace_local::fibonacci(n) << endl;
}
```

Par défaut, la fonction `fibonacci` appelée sera la version récursive dont la définition est contenue dans le fichier `fibonacci.cpp`. En revanche, le second appel *i.e.* celui prefixé par l'espace de nom `espace_local` se réferera à la version "locale" c'est-à-dire celle encapsulée entre `namespace espace_loca {...}`. Cette fonctionnalité introduite par le C++ a surtout pour vocation d'identifier des fonctions, classes d'objet ayant des propriétés communes tels qu'un ensemble de fonctions mathématiques qu'on pourra naturellement encapsuler dans un espace `math` afin d'éviter d'éventuel conflit de nom.

a. la compilation d'un fichier d'en-tête provoque la création d'un fichier d'extension `.gch` qui est systématiquement utilisé en lieu et place du fichier d'en-tête. Ainsi, l'utilisateur aura beau procéder à toute une série de changement dans le fichier d'en-tête, ces dernièrs ne seront jamais effectifs car toujours écrantés par le fichier d'extension `.gch`. Si tel était le cas, il faut alors supprimer ce fichier `.gch`.

1.2 Nombres à virgule flottante

Testez le code suivant :

```cpp
#include <iostream>

int main()
{
  double a = 0.1, b = 0.2, c = 0.3;
  if (a + b == c)
    std::cout << a + b << " est égal à " << c << std::endl;
  else
    std::cout << a + b << " n'est pas égal à " << c << std::endl;

  return 0;
}
```

Que se passe-t-il ? (Piste d'explication : ajoutez `std::cout.precision(19); std::cout.setf(std::ios::fixed);` au début du programme).
Essayez avec `a = 0.1, b = -0.1, c = 0.0`.
Est-ce spécifique au C++ ?

1.3 Déclaration de fonctions

Déclarer puis définir une fonction `affiche_complexe` qui prend en arguments la partie réelle et la partie imaginaire d'un nombre complexe et qui affiche le nombre complexe sous forme "a+ib" dans la console. Afficher en notation scientifique avec 2 chiffres après la virgule en utilisant `std::scientific` et `std::ios_base::precision` (on trouvera un exemple ici).

Correction

L'énoncé demande de déclarer une fonction `affiche_complexe` qui servira à afficher un nombre complexe. Comme cette fonction ne renvoie pas de valeur, et qu'elle prend deux `double`s en argument, la déclaration aura la forme :

```cpp
void affiche_complexe(double re, double im);
```

Le point d'entrée d'un programme en C++ est la fonction `int main()`. Nous y déclarons deux `double`s ; `a` sera la partie réelle et `b` la partie imaginaire, puis nous faisons appel à la fonction `affiche_complexe`. Comme la fonction `main` doit renvoyer un entier, nous finissons la fonction par `return 0;` (en général, la fonction main renvoie un code d'erreur, et 0 quand l'exécution a eu lieu sans problème).

```cpp
int main() {
  double a = 3;
  double b = 4;

  affiche_complexe(a,b);

  return 0;
}
```

Correction

Il ne reste plus qu'à implémenter la fonction `affiche_complexe`. On peut simplement écrire :

```
using namespace std;

void affiche_complexe(double re, double im) {
  cout.precision(2);
  cout.setf(ios::scientific);
  cout << "Nombre complexe: " << re << " + " << im << "i" << endl;
}
```

Autre code possible avec l'utilisation de manipulateurs plutôt que l'appel des méthodes de `cout` :

```
#include <iomanip>

void affiche_complexe(double re, double im) {
  std::cout << std::scientific << std::setprecision(2);
  std::cout << "Nombre complexe: " << re << " + " << im << "i" << std::endl;
}
```

Déclarer puis définir deux fonctions `norme` et `argument` qui prennent en arguments la partie réelle et la partie imaginaire d'un nombre complexe et qui retournent la norme et l'argument (dans $]-\pi;\pi]$). On pourra utiliser les fonctions `sqrt` et `atan` de la bibliothèque `<cmath>`.
Note : on pourait utiliser la fonction `atan2` directement, mais c'est un bon exercice de l'implémenter soi-même. Toutefois, dans un vrai code, il ne faut pas, dans la mesure du possible, ré-implémenter ce qui existe déjà !
Bien évidemment, tester ces fonctions, en particuliers dans les cas limites pertinents.

Correction

Pour avoir accès aux fonctions mathématiques telles que `std::sqrt` et `std::atan`, nous devons inclure la bibliothèque `<cmath>` de la bibliothèque standard. De manière générale, les directives `#include` se trouvent en tête des fichiers **.h* et **.cpp*. On rajoutera donc au début de notre fichier *main.cpp*

```
#include <cmath>
```

Les fonctions `norme` et `argument` retournent un nombre de type `double` et prennent en argument un nombre complexe représenté par sa partie réelle et imaginaire, donc la déclaration des fonctions `norme` et `argument` est :

```
double norme(double re, double im);
double argument(double re, double im);
```

Comme la déclaration des fonctions doit être faite avant leur utilisation, et comme nous allons tester ces fonctions dans `main`, les deux lignes ci-dessus doivent être insérées avant le début de la fonction `main`.
Quant à la définition de ces fonctions, elle peut avoir lieu n'importe où, en général, leur définition aura lieu après la fonction `main`. `<cmath>` défini les constantes `M_PI` et `M_PI_2`, qui seront pratiques pour `argument`.

Correction

```cpp
double norme(double re, double im) {
  return sqrt(re*re + im*im);
}

double argument(double re, double im) {
  if (re == 0) {
    return (im < 0) ? -M_PI_2 : M_PI_2;
  }
  double phi = atan(im / re);
  if(re > 0) {
    return phi;
  } else {
    if (im > 0) {
      return phi + M_PI;
    } else {
      return phi - M_PI;
    }
  }
}
```

Évidemment, on n'oubliera pas de tester les fonctions qu'on vient d'écrire modifiant la fonction `main`.

```cpp
int main()
{
  double a = -sqrt(2);
  double b = sqrt(2);

  affiche_complexe(a,b);
  cout << "Norme : " << norme(a,b) << endl;
  cout << "Argument : " << argument(a,b) << " (" << 180.0 * argument(a,b) / M_PI <<
  ↪  "deg)" << endl;

  return 0;
}
```

1.4 Pointeurs et références

Créer un programme dans lequel vous déclarerez :
- un entier `i`,
- une référence vers cet entier `ref_i`,
- un pointeur vers cet entier `ptr_i`.

Effectuer les opérations suivantes :

1. afficher l'adresse de la variable `i`
2. afficher la valeur de la variable `i`, directement et à travers le pointeur `ptr_i`

3. modifier la valeur de la variable en utilisant sa référence `ref_i`
4. afficher de nouveau la valeur de la variable, à travers le pointeur `ptr_i`

Correction

```cpp
#include <iostream>
using namespace std;
void main()
{
  int i = 666;
  int& ref_i = i;
  int* ptr_i = &i;

  cout << "Valeur de i = " << i << endl;
  cout << "Adresse (en mémoire) de i = " << ptr_i << endl;
  cout << "Valeur pointée par le pointeur = " << *ptr_i << endl;
  ref_i = 42;
  cout << "Nouvelle valeur pointée par le pointeur = " << *ptr_i << endl;
}
```

Soit le résultat à l'écran

```
Valeur de i = 666
Adresse (en mémoire) de i = 0x7ffd0514283c
Valeur pointée par le pointeur = 666
Valeur pointée par le pointeur = 42
```

1.5 Passage par valeur, référence et adresse

Pouvoir calculer l'exponentielle d'un nombre complexe est utile, pour, par exemple, calculer une transformée de Fourier. Écrire 3 fonctions `exponentielle_par_valeur`, `exponentielle_par_reference` et `exponentielle_par_adresse` qui ont 4 arguments : les deux premiers sont la partie réelle et imaginaire du nombre dont on souhaite calculer l'exponentielle, et les deux derniers sont la partie réelle et imaginaire du résultat. On dit que l'on fait un *retour par argument* (en pratique en C++, on ne fait presque plus jamais comme ça...).
La fonction `exponentielle_par_valeur` passera ses deux derniers arguments par valeur :

```cpp
double re_result, double im_result
```

tandis que la fonction `exponentielle_par_reference` les passera par référence :

```cpp
double& re_result, double& im_result
```

et la fonction `exponentielle_par_adresse` les passera par adresse :

```cpp
double* re_result, double* im_result
```

Tester les 3 différentes méthodes. Pourquoi `exponentielle_par_valeur` ne donne pas le résultat attendu ? Pour la suite des exercices, on utilisera uniquement `exponentielle_par_reference`, que l'on renommera en `exponentielle`.

Correction

L'énoncé demande de déclarer les fonctions suivantes :

```
void exponentielle_par_valeur(double re, double im, double re_result, double
↪  im_result);
void exponentielle_par_adresse(double re, double im, double * re_result, double *
↪  im_result);
void exponentielle_par_reference(double re, double im, double & re_result, double &
↪  im_result);
```

Pour la définition, nous avons besoin des fonctions `cos`, `sin` et `exp` qui font partie de la bibliothèque `<cmath>`, que nous avons déjà inclus.

```
void exponentielle_par_valeur(double re, double im, double re_result, double
↪  im_result)
{
  re_result = exp(re) * cos(im);
  im_result = exp(re) * sin(im);
}

void exponentielle_par_reference(double re, double im, double & re_result, double &
↪  im_result)
{
  re_result = exp(re) * cos(im);
  im_result = exp(re) * sin(im);
}

void exponentielle_par_adresse(double re, double im, double *re_result, double
↪  *im_result)
{
  *re_result = exp(re) * cos(im);
  *im_result = exp(re) * sin(im);
}
```

Comme test, on pourra par exemple utiliser

```
int main()
{
  // Nombre complexe z = log(2) + i Pi/2 ainsi, exp(z) = 2i, et on peut facilement
  ↪  vérifier que le code écrit retourne le bon résultat
  double a = log(2);
  double b = M_PI_2;

  double a1 = 0, b1 = 0;
  exponentielle_par_valeur(a,b, a1, b1);
  cout << "Exp par valeur : ";
  affiche_complexe(a1,b1);
```

Correction

```
  double a2, b2;
  exponentielle_par_reference(a, b, a2, b2);
  cout << "Exp par reference : ";
  affiche_complexe(a2,b2);

  double a3, b3;
  exponentielle_par_adresse(a,b, &a3, &b3);
  cout << "Exp par adresse : ";
  affiche_complexe(a3, b3);

  return 0;
}
```

Le passage par valeur ne donne pas le bon résultat car les variables à l'intérieur de `exponentielle_par_valeur` contiennent seulement des copies des valeurs passées à la fonction. Donc dans la fonction `exponentielle_par_valeur`, `re_result` contient 0 (qui est la valeur de `a1` au moment de l'appel de la fonction), mais la modification de `re_result` n'a aucun effet sur `a1`. Dans la fonction `exponentielle_par_reference`, `re_result` est en fait `a2`, donc modifier `re_result` revient de fait à modifier directement `a2`.

On vera par la suite comment, en C++, on retourne un tuple de deux valeurs plutôt que de faire un retour par arguments.

1.6 Opérateurs `new` et `delete`

Écrire en C++ , en utilisant les outils spécifiques à ce langage, les instructions C suivantes :

```
double * pt_tabular;
int n_value;
printf("Combien de valeurs souhaitez vous allouer en mémoire ?");
scanf("%d", &n_value);
if (n_value > 0) {
  pt_tabular = (double *) malloc (sizeof(double) * n_value);
  pt_tabular[n_value-1] = 0;
  free(pt_tabular);
}
```

On utilisera `std::cin`, qui est défini dans `<iostream>` et dont on trouvera la documentation ici.

Correction

```
  int n_value = 0;
  cout << "Combien de valeurs souhaitez vous allouer en mémoire ? ";
  cin >> n_value;
  if (n_value > 0) {
    double * pt_tabular = new double[n_value];
    pt_tabular[n_value-1] = 0;
    delete[] pt_tabular;
  }
```

Correction

Le code ci-dessus en plus d'utiliser les opérateurs `new` et `delete` en lieu et place de `malloc` et `free`, exploite la possibilité offerte par le C++ de déclarer les variables à l'endroit où elles sont pertinentes et non nécessairement au début du programme comme l'oblige le langage C. Ainsi, la déclaration de la variable `pt_tabular` se fait au moment de son allocation. On notera également que la restitution de mémoire dans le cas de l'allocation d'un tableau se fait par le biais de l'opérateur `delete[]`.

1.7 Allocation et dé-allocation de mémoire

Pour vérifier que la fonction `exponentielle` écrite à l'exercice 1.5 donne des résultats corrects, on va utiliser le fait que $\sum_{n=0}^{N-1} e^{\frac{2i\pi n}{N}} = 0$ quel que soit N un entier positif supérieur à 1. Écrire un programme qui invite l'utilisateur à enter N (on utilisera `std::cin >> N;`), alloue un tableau de `double` contenant la partie réelle de $e^{\frac{2i\pi n}{N}}$ et un deuxième tableau contenant la partie imaginaire, valeurs qui seront calculées dans une première boucle `for`. Une deuxième boucle `for` servira à faire la somme. Enfin, le programme affichera le résultat.

Correction

Pour allouer un tableau de double, la syntaxe est `double * tableau = new double [taille];`, tandis que pour le désallouer, il faut utiliser `delete[] tableau;`. Une solution possible est donc :

```
int main()
{
  unsigned int N = 0;
  cout << "N = ";
  cin >> N;

  double * re = new double[N];
  double * im = new double[N];

  for (unsigned int n = 0; n < N; ++n) {
    exponentielle_par_reference(0, 2 * n * M_PI / N, re[n], im[n]);
  }

  double re_result = 0, im_result = 0;
  for (unsigned int n = 0; n < N; ++n) {
    re_result += re[n];
    im_result += im[n];
  }

  affiche_complexe(re_result, im_result);

  delete[] re;
  delete[] im;

  return 0;
}
```

Correction

On pourra noter que les variables `re[n]` et `im[n]` sont passées par référence à la fonction `exponentielle_par_reference`. Ce sont en effet des variables (de type `double`). Règle pratique pour savoir si une expression `x` (ici `re[n]`) peut être passée en tant que référence : l'instruction `x = ...;` doit est valide (on parle de *left-value*, ou *lvalue*).

On remarquera aussi que les variables sont déclarées le plus proche possible de leur utilisation (en particulier, les variables `unsigned int n` sont déclarées dans la boucle `for`).

En pratique, en C++, on évitera d'écrire ce genre de code avec une allocation manuelle de mémoire. On écrira des objets dédiés, ou on utilisera les objets existants dans la bibliothèque standard.

1.8 std::vector<double>

Pré-requis : cours 3 sur la bibliothèque standard.

Dans l'exercice précédent, nous avons manuellement alloué puis dé-alloué un tableau. La bibliothèque standard contient des objets qui permettent d'automatiser cette gestion de la mémoire : les objets `std::vector<type>` de `<vector>`. Ré-écrire le programme de l'exercice précédent en utilisant `std::vector<double>`, soit avec la méthode `push_back`, soit en créant un tableau de la bonne taille dès le début.

Rappel de méthodes qui peuvent être utiles :

- `vector<double> monTableau;` pour déclarer un objet de type `vector<double>` (un tableau contigu de taille variable contenant des `double`), qui sera vide par défaut.
- `monTableau.push_back(nombre)` pour rajouter un nombre à la fin du tableau. La taille du tableau est ajustée automatiquement.
- `monTableau.size()` pour connaître le nombre d'éléments dans le tableau, de type `unsigned int`.
- `monTableau[n]` ou `monTableau.at(n)` pour accéder ou modifier le `n`ème élément (toujours en partant de 0) du tableau. Attention, il faut que `n < monTableau.size()`, sinon on risque une erreur de segmentation (avec `operator[]`) ou une exception (avec `at()`).

Correction

On n'oublie pas d'inclure la bibliothèque `<vector>`

```
#include <vector>
```

Il faut remplacer l'allocation de mémoire par la déclaration de deux `vector<double>`. Pour rajouter un élément à un `vector`, il faut utiliser la fonction `push_back`. L'accès au n-ième élément de `vector<double> tableau;` peut être réalisé de deux manières :

- soit avec la méthode `tableau.at(n)`
- soit avec l'opérateur `[](int index)`, c'est-à-dire `tableau[n]`

Dans les deux cas, il faut que `n < tableau.size()` [a].

```
void main() {
  unsigned int N = 0;
  cout << "N = ";
```

```
  cin >> N;

  vector<double> re, im;

  for (unsigned int n = 0; n < N; ++n) {
    double re_exp, im_exp;

    exponentielle_par_reference(0, 2 * n * M_PI / N, re_exp, im_exp);
    re.push_back(re_exp);
    im.push_back(im_exp);
  }

  double re_result = 0, im_result = 0;
  for (unsigned int n = 0; n < N; ++n) {
    re_result += re[n];
    im_result += im[n];
  }

  affiche_complexe(re_result, im_result);
}
```

On remarque que les opérateurs `new[]` et `delete[]` ne sont plus présents dans le code.

Une autre facon de faire consiste à utiliser initialiser le vecteur avec `N` éléments dès le début, en utilisant le *constructeur* `std::vector<double>(size_type count)`. On peut aussi utiliser la méthode `resize()` de l'objet `std::vector<double>`, qui fait en sorte que le `vector` contienne exactement le nombre demandé d'élément. Le code est alors plus concis :

```
void main() {
  unsigned int N = 0;
  cout << "N = ";
  cin >> N;

  vector<double> re(N), im(N);

  for (unsigned int n = 0; n < N; ++n) {
    exponentielle_par_reference(0, 2 * n * M_PI / N, re[n], im[n]);
  }

  double re_result = 0, im_result = 0;
  for (unsigned int n = 0; n < N; ++n) {
    re_result = re[n] + re_result;
    im_result = im[n] + im_result;
  }
```

```
  affiche_complexe(re_result, im_result);
}
```

a. Si $n >$ `tableau.size()`, le comportement des deux méthodes est différent. L'opérateur `[](int)` n'effectue aucune vérification, et une erreur de segementation se produira peut être ou peut être pas. Au contraire, la méthode `at(int)` effectue la vérification, et lance toujours une exception lorsque n dépasse la taille du tableau.

Correction

[Bonus] Dans un deuxième temps, simplifier le code en remplaçant la sommation manuelle par l'algorithme `std::accumulate(itérateur_début, itérateur_fin, zéro)` présent dans `<numeric>`. On trouvera un exemple ici.

Correction

La fonction `std::accumulate` effectue la somme des éléments d'un conteneur. Comme tous les algoritmes de la bibliothèque standard, elle manipule en fait des *itérateurs*, qui ne sont rien d'autre que des sortes de pointeurs adaptés à chaque type de conteneur. On fournit un itérateur de début `it_beg` et de fin `it_end` à l'algorithme : la somme sera effectée dans l'intervalle `[it_beg,it_end[`. Pour sommer le conteneur entier, il faut alors prendre `it_beg = monVector.begin()` et `it_end = monVector.end()`. On fournit aussi un élément d'initialisation pour commencer la somme ; ici, c'est simplement zéro, l'élément neutre de l'addition [a].

```
void main() {
  unsigned int N = 0;
  cout << "N = ";
  cin >> N;

  vector<double> re(N), im(N);

  for (unsigned int n = 0; n < N; ++n) {
    exponentielle_par_reference(0, 2 * n * M_PI / N, re[n], im[n]);
  }

  double re_result = std::accumulate(re.begin(), re.end(), 0.);
  double im_result = std::accumulate(im.begin(), im.end(), 0.);

  affiche_complexe(re_result, im_result);
}
```

a. Attention, `std::accumulate<type_t>` est une fonction template qui fonctionne sur n'importe quel type `type_t`. Il se trouve que le compilateur déduit `type_t` à partir du troisième argument. Et si on écrit `0`, il déduit `int`, ce qui n'est pas ce que l'on veut (il somme alors des entiers et retourne un entier). Pour lui faire déduire `double`, il faut soit écrire `0.` (notez le point !), soit `double(0)`, soit encore expliciter le type avec `std::accumulate<double>(it1, it1, 0);`.