INFO-F105 — Langages de Programmation I
Corrigés des exercices

Christophe Petit, Attilio Discepoli, Quentin Magron,
Robin Petit, Pascal Tribel & Renaud Vilain

Remerciements à :
Youcef Bouharaoua, Hakim Boulahya, Arnaud Leponce, Svitlana Lukicheva,
Yannick Molinghen, Gian Marco Paldino,Arnaud Pollaris,
Chloé Terwagne & Nassim Versbraegen

Ce syllabus d'exercices est basé en partie sur une ancienne version réalisée par :
Jacopo De Stefani, Luciano Poretta, Cédric Ternon, Nikita Veshchikov
Stéphane Fernandes Medeiros, Keno Merckx, Jérôme Dossogne & Naïm Qachri

dont les correctifs ont été mis en page par :
Arnaud Pollaris, Nassim Versbraegen, Xavier Barthel,
Robin Petit & Sarah Van Bogaert

Année académique 2024 - 2025

Table des matières

Séance 1 — Compilation et exécution de programmes en C++

Exercice 1. La fonction `main` est la fonction principale de tout programme C++. L'exécution d'un programme entraîne obligatoirement l'appel de la fonction `main`. Tout programme doit retourner une valeur `int` comme code erreur à son environnement d'exécution. Il s'agit ici de la valeur de retour de la fonction `main`. La convention veut que cette valeur vaut 0 si le programme s'éxécute avec succès (sans anomalies ou erreurs fatales). Le programme suivant (qui se trouve dans le fichier source `hello.cpp`) affiche "Hello !" à l'écran.

Programme 1 – hello.cpp

```
1 #include <iostream> /* cin, cout et endl */
2
3 int main(){
4   // affiche "Hello!" a l'ecran:
5   std::cout << "Hello!" << std::endl; // endl est comme "\n"
6
7   return 0; // Tout va bien! Le programme se termine sans erreurs
            .
8 }
```

Sur la première ligne, la directive commençant par `#include` permet d'inclure des librairies. La librairie `<iostream>` permet de gérer les flux d'entrée et de sortie, pour afficher du texte à la console et récupérer la saisie au clavier. Elle permet ici d'utiliser `cout` (affichage à l'écran) et `endl` (passage à la ligne) dans l'espace de nommage `std`.

Exercice 4. Lorsqu'on sauvegarde le fichier et qu'on lance l'exécutable `hello`, le programme affiche "*Hello !*" au lieu de "*Hello, world*". En effet, il est nécessaire de compiler à nouveau le fichier `hello.cpp` pour mettre à jour l'exécutable qui correspond encore à l'ancienne compilation.

Exercice 5. Lorsqu'on supprime le symbole `;` et que l'on recompile le fichier, la compilation échoue et on a le message d'erreur *"error : expected ; before return"*. Dès lors, l'exécutable n'est pas mis à jour et correspond encore à la compilation précédente. Il est toutefois possible de lancer cet exécutable, ce qui donnera le message *"Hello, world"*.

Exercice 6. En compilant avec les flags `-Wall` et `-Wextra`, des avertissements supplémentaires sont affichés lors de la compilation. Lorsqu'une variable `int x` est définie sans être utilisée, le message suivant apparaît : `Warning: unused variable 'x'`. Toutefois, ce n'est pas une erreur de compilation et l'exécutable est tout de même créé et peut être exécuté sans problèmes.

Exercice 7. Tout comme `cout` permet l'affichage de données, `cin` permet la récupération de données. `cout` est en réalité un *flux* sur la *sortie standard* (également appelée *stdout*) qui n'est autre que le terminal en mode écriture, et `cin` en est un sur l'*entrée standard* (également appelée *stdin*) qui est le terminal en mode lecture.

Notons que pour écrire sur *stdout*, il faut utiliser les double chevrons gauches (i.e. `<<`) alors que pour lire sur *stdin*, il faut utiliser les double chevrons droits (i.e. `>>`).

Programme 2 – sum.cpp

```
1 #include <iostream>
2
3 int main(){
4   int x, y;  // déclare 2 variables entières
```

```
5
6     std::cout << "Calculateur de somme\n";
7
8     std::cout<< "Entrez un nombre X = ";
9     std::cin >> x;
10
11    std::cout << "Entrez un nombre Y = ";
12    std::cin >> y;
13
14    // std::endl est comme '\n'
15    std::cout << x << " + " << y << " = " << x+y << std::endl;
16
17    return 0;
18 }
```

Exercice 8.

Programme 3 – error.cpp

```
1  #include<iostream>
2
3  int main(){
4    int a1, a2;
5
6    std::cout << "Calculateur de division\n";
7
8    std::cout << "Entrez une valeur 1 : ";
9    std::cin >> a1;
10
11   std::cout << "Entrez une valeur 2 : ";
12   std::cin >> a2;
13
14   std::cout << "Le résultat : "  << a1/a2 << std::endl;
15
16   return 0;
17
18 }
```

Lorsqu'on introduit les valeurs 16 et 3, le programme montre la valeur 5, ce qui correspond à la division entière de 16 par 3 (C++ traite systématiquement la division entre deux entiers comme une division entière. Le résultat est donc la valeur plancher). Si l'on veut avoir une division réelle, il faut changer au moins un des deux nombres en `float` ou `double` : `(double) a1/a2` ou `a1/(double) a2`.

Exercice 9.

Programme 4 – Makefile

```
1  FLAGS = -std=c++11 -Wpedantic -Wall -Wextra -Winit-self -Winline
           -Wconversion -Weffc++ -Wstrict-null-sentinel -Wold-style
           -cast -Wnoexcept -Wctor-dtor-privacy -Woverloaded-
           virtual -Wconversion -Wsign-promo -Wzero-as-null-pointer
           -constant
2
```

```
# FLAGS est une variable avec différentes options de compilation

# structure:
# <nom du programe> : fichiers nécessaires pour créer le
        programme
# <TAB> <ligne de commande pour créer le programme>

all: hello

hello: hello.cpp
  g++ $(FLAGS) hello.cpp -o hello
```

Lorsqu'on lance `make` sans aucun argument, il considérer la première cible comme la cible principale (Notez que la cible - ou *target* en anglais - est ce qui se trouve au début de la ligne, devant les double points, par exemple `all`). C'est pour cette raison que la cible `all` est utile au début du fichier si on veut compiler plusieurs programmes en même temps.

Dans l'énoncé, on voit l'expression `<nom du programme>`. Cela signifie que vous devez remplacer l'expression par le nom du programme, c-à-d le nom de l'exécutable ou du fichier objet créé lors de l'exécution de la ligne de commande. Notez que vous ne devez bien évidemment pas garder les crochets !

Lorsqu'on exécute `make` une première fois, cela compile le fichier source (c-à-d, `hello.cpp`) et l'exécutable est mis à jour si tout se passe correctement. Lorsqu'on exécute `make` une seconde fois, sans avoir modifié le fichier source, on voit le message : `make: Nothing to be done for 'all'`. En effet, la commande `make` voit que le fichier source n'a pas changé et qu'il n'est donc pas nécessaire de le recompiler.

Exercice 10.

Programme 5 – Makefile

```
FLAGS = -std=c++11 -Wpedantic -Wall -Wextra -Winit-self -Winline
        -Wconversion -Weffc++ -Wstrict-null-sentinel -Wold-style
        -cast -Wnoexcept -Wctor-dtor-privacy -Woverloaded-
        virtual -Wconversion -Wsign-promo -Wzero-as-null-pointer
        -constant

# FLAGS est une variable avec différentes options de compilation

# structure:
# <nom du programe> : fichiers nécessaires pour créer le
        programme
# <TAB> <ligne de commande pour créer le programme>

all: hello sum

hello: hello.cpp
  g++ $(FLAGS) hello.cpp -o hello

sum: sum.cpp
  g++ $(FLAGS) sum.cpp -o sum
```

Lorsqu'on ne modifie qu'un seul des fichiers source, seule la commande de compilation liée à ce fichier est exécutée. Lorsqu'on modifie les deux fichiers source, les deux commandes de compilation sont exécutées.

Séance 2 — Expressions et instructions

Exercice 11. Essayons d'abord de comprendre ce que fait le programme. Deux entiers sont demandés à l'utilisateur, et certaines conditions sont vérifiées sur ces deux valeurs. Nous constatons que les deux conditions sont l'une dans l'autre. Tout d'abord, nous vérifions que le deuxième nombre est non nul, et seulement si cette condition est vraie, nous utilisons alors le résultat du module entre les deux nombres comme deuxième condition. Nous vérifions donc que le premier entier a est divisible par le second entier b, en considérant également le cas où b est égal à zéro. En Python, les `cin` sont remplacées par des `input()` et les `cout` par des `print()`. Nous notons que le résultat de `input()` est une chaîne de caractères, nous devons donc préciser (*cast*) qu'il s'agit d'un nombre entier avec `int()`. Les conditions et la structure du code ne changent pas. Le résultat est le suivant :

Programme 6 – Code en Python

```python
a = int(input("introduisez un entier a: \n"))
b = int(input("introduisez un entier b: \n"))

if b!=0:
    if (a%b):
        print("la division de a par b donne un reste non nul")
    else:
        print("a est divisible par b")
else:
    print("Impossible de diviser par 0")
```

Exercice 12. Le texte du problème nous suggère d'utiliser `else if` et de prêter attention à la priorité des opérateurs. En fait, en C++, l'opérateur `not` serait considéré avant l'opérateur `modulo`, ce qui entraînerait une évaluation incorrecte de l'expression. Nous devons donc utiliser des parenthèses. Après avoir également modifié le `print` et le `input`, nous obtenons :

Programme 7 – Code en C++

```cpp
#include <iostream>
using namespace std;

int main() {
    int a, b;

    cout << "introduisez un entier a: " << endl;
    cin >> a;
    cout << "introduisez un entier b: " << endl;
    cin >> b;

    if (b!=0 and not (a%b)) {
        cout << a << " est divisible par " << b
             << " et cela vaut : " << a/b << endl;
    } else if (b!=0) {
        cout << a << " n'est pas un multiple de " << b << endl;
    } else {
        cout << "Impossible de diviser par 0." << endl;
    }
    return 0;
```

```
}
```

Exercice 13. Les deux languages suivent le principe de la *lazy evaluation* : une chaîne de conditions est évaluée de la première à la dernière, une à la fois. Dès qu'il existe une condition qui rend toute l'expression vraie ou fausse, l'évaluation des conditions suivantes n'est pas effectuée. Cela permet de gagner en temps, mais donne au programmeur la responsabilité de l'ordre des conditions. En fait, dans notre cas, en inversant les deux conditions, si nous attribuons à `b` la valeur 0, nous obtenons une erreur de division par 0 dans les deux languages, puisque la première condition est évaluée avant la seconde.

En C++ :

Programme 8 – Code en C++ en inversant les deux conditions

```cpp
#include <iostream>

using namespace std;

int main() {
    int a, b;

    cout << "introduisez un entier a: " << endl;
    cin >> a;
    cout << "introduisez un entier b: " << endl;
    cin >> b;

    if (not (a%b) and b!=0) {
        cout << a << " est divisible par " << b
             << " et cela vaut : " << a/b << endl;
    } else if (b!=0) {
        cout << a << " n'est pas un multiple de " << b << endl;
    } else {
        cout << "Impossible de diviser par 0." << endl;
    }
    return 0;
}
```

En Python :

Programme 9 – Code en Python en inversant les deux conditions

```python
a = int(input("introduisez un entier a: \n"))
b = int(input("introduisez un entier b: \n"))

if not a%b and b!=0:
    print(a, "est divisible par", b, "et cela vaut :", a//b)
elif b!=0:
    print(a, "n'est pas un multiple de", b)
else:
    print("Impossible de diviser par 0.")
```

Exercice 14. Les opérateurs booléens, comme les opérateurs arithmétiques, sont régis par un ordre de priorité. Pour découvrir l'odre de priorité, nous allons écrire la table de vérité de trois variables booléennes et de leurs combinaisons en utilisant explicitement des parenthèses pour forcer la priorité des opérations.

A	B	C	$(A \wedge B) \vee C$	$A \wedge (B \vee C)$
0	0	0	0	0
0	0	1	1	0
0	1	0	0	0
0	1	1	1	0
1	0	0	0	0
1	0	1	1	1
1	1	0	1	1
1	1	1	1	1

On constate que deux lignes diffèrent. Ainsi, $(0 \wedge 0) \vee 1 = 1$, et est différent de $0 \wedge (0 \vee 1) = 0$.

Donc, en écrivant un programme qui effectue cette opération sans parenthèses, on peut découvrir quelle opération est prioritaire.

Programme 10 – Pour découvrir quelle opération est prioritaire

```
#include <iostream>
using namespace std;
int main() {
   if(false and false or true) {
        std::cout << "Le AND a priorité sur le OR" << std::endl;
    } else {
        std::cout << "Le OR a priorité sur le AND" << std::endl;
    }
    return 0;
}
```

Exercice 15. Non le compilateur n'a pas cette liberté. En effet, si les opérandes d'un opérateur booléen sont de simples variables, alors – tant que la priorité des opérations est respectée – les opérandes peuvent être évalués dans n'importe quel ordre. Cependant, les opérandes sont des *expressions* quelconques et pas nécessairement des variables.

Considérons le code suivant :

```
#include <iostream>
using namespace std;

int main() {
    int x = 1, y = 1;
    bool cond1 = false and (y--);
    bool cond2 = (x--) and false;
    cout << x << '\t' << y << endl;
    return 0;
}
```

Les deux booléens `cond1` et `cond2` seront tous les deux évalués à `false` (puisqu'une conjonction contenant au moins un faux sera toujours fausse), mais par le principe de *lazy evaluation* mentionné ci-dessus, lors de l'évaluation de `cond1`, l'expression `(y--)` ne sera pas évaluée, alors que dans l'évaluation de `cond2`, `(x--)` est exécuté (et évalué à `true`). Dès lors, en fonction de l'ordre d'évaluation des opérandes, le résultat peut être différent.

Bien entendu, cela repose *uniquement* sur la *lazy evaluation* qui fait que certains opérandes peuvent ne pas être évalués.

Exercice 16. Voici les réponses numérotées :

1. Le code évalue si deux entiers entrés par l'utilisateur sont opposés, en vérifiant que leur somme est ou n'est pas 0. La version Python correspondante est facile à écrire (ci-dessous). A l'exécution, cependant, on remarque que la version Python ne fonctionne pas et génère une `SyntaxError`.

   Notons tout de même que Python permet (depuis Python 3.8, c.f. la PEP-572) un tel comportement mais impose une syntaxe différente. Pour cela il faut noter l'assignation avec `:=` et non avec `=`. Dès lors `if a := a+b: ...` est tout à fait correct. Cette notation porte le nom informel de *walrus operator* car `:=` peut faire penser aux défenses d'un morse (*walrus* en anglais). D'ailleurs cet opérateur peut être utilisé dans bien plus de contextes que simplement dans l'évaluation d'une condition. Plus d'informations se trouvent dans la PEP.

2. La compilation sans l'option `-Wall` fonctionne avec succès. Avec l'option `-Wall`, qui permet d'afficher des *warnings*, le compilateur nous suggère d'utiliser des parenthèses dans la condition `if (a=a+b)` : `warning: suggest parentheses around assignment used as truth value [-Wparentheses]`. La fin de ce message (i.e. `[-Wparentheses]`) nous signale que c'est le flag `-Wparentheses` qui a généré ce *warning*. `-Wparentheses` est un des nombreux *warnings* qui sont activés par `-Wall`. Ce warning dit que nous n'avons pas *explicité* le fait que nous voulions bien une assignation et non une comparaion dans la condition et nous dit que c'est peut-être une erreur de notre part. Le fait de mettre des parenthèses supplémentaires autour de cette assignation permet de signaler au compilateur qu'une assignation est bien ce qu'on cherchait à faire et que nous ne nous sommes pas trompés. Notons toujours que ce *warning* est là pour s'assurer que le code fonctionne bien comme on voudrait qu'il fonctionne, mais les parenthèses ne sont en ien obligatoire (la preuve étant que le code compile très bien sans).

3. En remplaçant l'expression `if (a=a+b)` par `if (a+=b)` le *warning* n'apparaît plus. En effet le compilateur ne se dit pas que nous aurions possiblement voulu faire une comparaison ici car `+=` est suffisamment différent de `==` pour que le compilateur nous fasse (et est-ce une bonne chose) confiance.

4. Comme toujorus quand il est question de bonnes pratiques, la réponse est très discutable (et discutée). En effet, cela permet d'écrire un code plus court, considérons par exemple ce code en Python qui récupère des nombres sur l'entrée standard jusqu'à ce qu'un -1 soit rentré et puis en affiche la somme :

```python
somme = 0
while (n := int(input('Entrez un nombre: '))) != -1:
    somme += n
print('Somme :', somme)
```

   Sans le *walrus operator*, nous pouvons écrire le code comme ceci :

```python
somme = 0
loop = True
while loop:
    n = int(input('Entrez un nombre : '))
    if n == -1:
        loop = False
    else:
        somme += n
print('Somme')
```

   mais le code est assez long, ou encore comme ceci :

```python
somme = 0
n = int(input('Entrez un nombre : '))
while n != -1:
    somme += n
```

```
    n = int(input('Entrez un nombre : '))
print('Somme :', somme)
```

mais nous avons une duplication de la ligne d'entrée du nombre, ce qui n'est pas terrible.

Il faut cependant toujours faire bien attention lorsqu'une telle assignation est faite pendant l'évaluation d'une expression car la lisibilité peut en être impactée.

Version Python du code d'exercice : il faut utiliser l'opérateur `:=` pour mettre à jour la valeur de `a` comme dans le code C++ :

Programme 11 – Code en Python

```
if __name__ == "__main__":
    print("Programme qui affiche un message si deux nombres
          entiers ne sont pas opposés.")
    a = int(input("Introduisez un premier entier: "))
    b = int(input("Introduisez un second entier: "))
    if a = a+b:  # Syntax error !: Il faudrait écrire if a := a+b
        print ("Erreur : les deux nombres ne sont pas opposés")
```

Exercice 17. Nous allons donc écrire et exécuter deux programmes qui ont la même sémantique mais dans deux langages différents. Ensuite, nous utiliserons l'utilitaire `time` pour comparer les temps d'exécution.

Programme 12 – cpp_benchmark.cpp

```
#include <iostream>
using namespace std;

int main() {
    float a = 2;
    float b = 5;
    float c = 0;
    for (int i = 0; i < 100'000'000; i++) {
        c = (a + b + c) / a;
    }
    cout << "Le résultat est " << c << endl;
    return 0;
}
```

Programme 13 – python_benchmark.cpp

```
if __name__ == "__main__":
    a = 2
    b = 5
    c = 0
    for _ in range(100_000_000):
        c = (a + b + c) / a
    print(f"Le résultat est {c}")
```

Par facilié, nous utiliserons aussi ce `Makefile` où la commande `make run` nous permet de compiler et d'exécuter les deux programmes dans la foulée.

Programme 14 – Makefile

```
cpp_benchmark: cpp_benchmark.cpp
  g++ cpp_benchmark.cpp -o cpp_benchmark

run: cpp_benchmark
  time -p ./cpp_benchmark
  time -p python3 python_benchmark.py
```

Lançons les deux programmes.

```
$ make run
g++ cpp_benchmark.cpp -o cpp_benchmark
time -p ./cpp_benchmark
Le résultat est 7
real 0.83
user 0.83
sys 0.00

time python3 python_benchmark.py
Le résultat est 7.0
real 21.89
user 21.89
sys 0.00
```

Regardons tout d'abord le résultat affiché par les deux programmes : on voit bel et bien que les deux programmes affichent le résultat 7. Ouf !

Nous constatons que le programme écrit en C++ est largement plus rapide que le programme écrit en Python. On peut même dire que le programme C++ est $\frac{21.89}{0.83} \approx 26$ fois plus rapide que le programme en Python sur la machine de test.

Séance 3 — Modes de stockage et portée

Exercice 18. Qu'est-ce que `gdb` ?

Il s'agit du debugger du projet `GNU` (cfr aussi `GNU Linux`, le système d'exploitation). L'intérêt de `gdb`, c'est qu'il peut vous aider à voir ce qui se passe dans votre programme durant son exécution. C'est particulièrement utile pour mieux comprendre ce qui a posé problème en cas de crash.
(source : https ://www.gnu.org/software/gdb/).

Si l'on entre dans notre exemple les valeurs 6, 2 et 0, on reçoit une `floating point exception` car `2%0` n'est pas permis (dans `myMod`). Si l'on entre les valeurs 6, 4, 2, on reçoit ici encore une `floating point exception`.

Qu'est-ce qui a bien pu se passer ? Pour le savoir, recompilons le programme avec le flag `-ggdb`. Ainsi, les noms de fonctions et de variables seront conservés.

Lançons le programme avec le débugger (`gdb ./<program_name>` suivi de `run`). Si l'on introduit par exemple les valeurs 6, 4, 2, nous recevons un message qui indique qu'une exception arithmétique a été levée et que la cause se situe dans `myDiv(x=6, y=0)` au moment du `return x/y;` (`y` vaut 0 car c'est le résultat de `4%2`).

Il est également possible d'utiliser le debugger pour consulter l'état de la pile. Pour rappel, la pile (stack) est une section de la mémoire `RAM` utilisée notamment pour le stockage des variables locales d'une fonction.

Consultons donc l'état de la pile avec la commande `bt`. Nous voyons pour notre exemple qu'à la base, dans la fonction `main()`, ligne 24, la fonction `myDiv()` a été appelée avec les valeurs suivantes pour les arguments : `x=6`, `y=0`. Ici, nous ne voyons pas `myMod()` car cette fonction a déjà fini de s'exécuter au moment du crash et a permis d'obtenir la valeur de `y=0`. La stackframe de `myMod()` n'est donc plus dans la pile.

A quoi sert la pile ?

Elle stocke les valeurs des variables locales (par exemple dans une stackframe qui existe momentanément durant l'exécution de la fonction) mais pas seulement. En fait, à chaque appel de fonction, différentes choses devront être sauvegardées sur la stack, comme par exemple le contenu des registres de la fonction appelante (afin que la fonction appelée puisse disposer des registres pour sa propre exécution), les valeurs des arguments que va recevoir la fonction appelée, ainsi que l'adresse de la prochaine instruction à exécuter dans le code de la fonction appelante car, lorsque la fonction appelée aura fini de s'exécuter, il faudra savoir où retourner dans le code de la fonction appelante pour exécuter la prochaine instruction. Ainsi, à chaque nouvel appel de fonction, toute une série d'informations sera ajoutée sur la pile (et réciproquement, à la sortie de la fonction, avec le `return`, la pile diminuera). En Assembleur Intel, c'est par exemple le registre `ESP` qui contiendra l'adresse indiquant le sommet de la pile. Donc, dans les grandes lignes, à chaque appel de fonction, toute une série d'informations (comme les variables locales) seront temporairement placées au sommet de la pile et seront conservées jusqu'à la fin de l'exécution de la fonction.

Dans notre exemple, en utilisant le debugger, on peut constater que l'état de la pile n'est pas le même au moment de l'exception lorsque les valeurs 6, 2 et 0 ont été introduites ou lorsque les valeurs 6, 4, 2 ont été introduites.

Exercice 19.

- La valeur de `i` après la boucle `for` (l. 23) vaut 42 ;
- la valeur de `j` après la boucle `for` (l. 24) vaut 58 ;
- la valeur de `k` après la boucle `for` (l. 25) vaut 666 ;
- la valeur de `l` après la boucle `for` (l. 26) vaut 56 ;
- la valeur de `i` après la condition `if` (l. 33) vaut 666 ;
- la valeur de `j` après la condition `if` (l. 34) vaut 58.

La variable `i` utilisée dans la boucle `for` est une variable qui n'existe que dans le *bloc* d'instructions de la boucle. La variable référencée dans l'incrément de la ligne 21 et dans le `cout` de la ligne 23 est donc la

variable `i` déclarée à la ligne 6 et pas celle déclarée à la ligne 11. La valeur affichée est donc 42, peu importe les itérations de la boucle `for` (l. 11-20).

La variable `j` référencée dans le `cout` de la ligne 24 est celle définie à la ligne 7 qui est modifiée dans la boucle `for` (à la ligne 16) en prenant la valeur de la variable `i` définie à la ligne 11. Comme cette variable `i` est initialisée à 40, est augmentée de 2 dans la boucle (l. 15) et est réaugmentée de 2 dans le pas de la boucle (l. 11), lors de la réassignation de `j`, les valeurs de `i` sont (dans l'ordre) : 42, 46, 50, 54, 58. Lors de la sortie de la boucle, la dernière valeur assignée à `j` est donc 58.

La variable `k` référencée dans le `cout` de la ligne 25 est celle définie à la ligne 8, qui est totalement indépendante de celle déclarée à la ligne 12. Sa valeur est donc 666 comme lors de son initialisation.

De manière similaire à la variable `j`, la variable `l` référencée dans le `cout` de la ligne 26 est la variable définie à la ligne 9 et qui est réassignée à la ligne 13. La différence est la suivante : contrairement à `j`, la variable `l` est réassignée *avant* l'instruction `i+=2;`, et donc les valeurs prises par cette variable sont (dans l'ordre) : 40, 44, 48, 52, 56.

Comme `i` et `j` n'ont pas la même valeur lors de l'évaluation de la condition (l. 29), le bloc des lignes 30-31 sera obligatoirement exécuté. La variable `i` est donc réassignée, et une *nouvelle* variable `j` est déclarée, bien que cette dernière n'existe que dans ce même bloc. La variable `j` déclarée à la ligne 7 n'est donc pas modifiée dans ce bloc, ce qui explique pourquoi les valeurs affichées sont respectivement 666 et 58.

Afin de lever toute amibiguïté concernant la variable référencée lors de chaque instruction, on peut se convaincre que le code suivant est strictement équivalent à celui donné dans l'énoncé. La seule différence est que chaque variable a un nom unique :

```
1  int main() {
2    int i=41;
3    int j=i;
4    int k=666;
5    int l=0;
6
7    for (int i2=40; i2<60; i2+=2) {
8      int k2=i2;
9      l=i2;
10
11     i2+=2;
12     j=i2;
13     i2=j;
14     cout << i2 << ' ';
15     cout << j2 << endl;
16   }
17   i++;
18
19   cout << "après la boucle for, i vaut : " << i << endl;
20   cout << "après la boucle for, j vaut : " << j << endl;
21   cout << "après la boucle for, k vaut : " << k << endl;
22   cout << "après la boucle for, l vaut : " << k << endl;
23
24
25   if (i!=j) {
26     i=666;
27     int j2=666;
28   }
```

```
29
30    cout << "après la condition if, i vaut : " << i << endl;
31    cout << "après la condition if, j vaut : " << j << endl;
32
33    return 0;
34 }
```

Exercice 20. Parmi les différences majeures entre C++ et Python, nous pouvons noter que le système de portée de Python est dynamique alors que celui de C++ est statique. Cela veut dire qu'en C++, toute variable est déclarée dans un bloc et est accessibles dans tous les *sous-blocs* de ce même bloc mais n'est pas accessible dans les blocs parents du bloc de déclaration ; alors qu'en Python, dans une fonction, peu importe le bloc dans lequel une variable est déclarée elle sera accessible partout dans le reste de la fonction. En effet, le code Python ci-dessous est tout à fait valide alors que le code C++ ne compilera pas.

```python
if __name__ == '__main__':
    if True:
        x = 0
    cond = x > 0
```

```cpp
int main() {
    if(true) {
        int x = 0;
    }
    bool cond = x > 0;
    return 0;
}
```

Dans le code C++ ci-dessus, même si d'un point de vue conceptuel, la variable `x` sera toujours déclarée (puisque la condition `true` est *par définition* toujours vraie), cette dernière n'existera que dans le bloc défini par les accolades (l. 2-4). Dès lors, lors de l'initialisation de la variable `cond`, il est fait référence à une variable `x` n'existant pas dans le *scope* actuel, ce qui mènera à une erreur de compilation.

Ce problème n'est pas rencontré en Python puisqu'une variable existe à partir du moment où elle est définie. Il découle directement de cela qu'une erreur dans le nom d'une variable ne pourra pas être découverte par Python avant que la ligne la contenant ne soit exécutée par l'interpréteur.

Cette propriété sur le système de portée de Python et C++ rend possible le *shadowing* de variable (comme montré dans le code de l'exercice précédent) en C++, c'est-à-dire la déclaration d'une variable dans un sous-bloc ayant le même nom qu'une variable déclarée dans un bloc parent sans ambiguïté (aux yeux du compilateur) :

```cpp
int main() {
    int var;
    {
        int var;  // nouvelle variable -> shadowing
    }
    return 0;
}
```

Ce qui est impossible en Python : lors d'une écriture dans une variable en Python, l'interpréteur regarde d'abord s'il existe une variable de ce nom dans le scope actuel et écrit dedans si c'est le cas, et la crée si ce n'est pas le cas.

Il faut tout de même noter que le shadowing en C++ requiert que les deux variables de même nom soient déclarées dans des blocs différents ! Le code suivant n'est pas valide et le compilateur donnera une erreur :

```
1 int main() {
2     int var;
3     int var;
4     return 0;
5 }
```

Les différences entre le code Python et le code C++ sont les suivantes : (i) le déroulement de la boucle `for` (à cause de l'itérateur) et (ii) l'absence de shadowing.

En effet, la boucle `for` en Python correspond à une boucle *for each* (qui *itère* donc sur tous les éléments d'un conteneur) et pas à une boucle *for* au sens de C(++). Notons qu'une boucle *for each* existe également en C++ mais dépasse le cadre de ce cours. En C++, la boucle `for` travaille sur une variable bien définie et la *modifie* à chaque itération alors que la boucle `for` en Python *réassigne* une variable bien définie à chaque itération.

`range` en Python est une classe qui définit une méthode `__iter__` qui renvoie un itérateur de type `range_iterator`, qui quant à lui définit une méthode `__next__` permettant de passer à l'élément suivant du *range* (c.f. INFO-F103). Dès lors, `for i in range(start, stop, step)` en Python va créer une variable `i` s'il n'en existe pas déjà une, et puis lui *assigner* successivement les valeurs `start`, et puis `start+step`, et puis `start+2*step`, etc. jusqu'à la plus grande valeur de la forme `start+k*step` qui est plus petite ou égale à `stop`. Peu importe les modifications faites sur `i` au sein de la boucle, à chaque nouvelle itération, la variable `i` prend une nouvelle valeur dépendant de l'itérateur qui, lui, n'a pas été modifié.

```
1 # La boucle for suivante :
2 for i in range(start, stop, end):
3     # ...
4 # est équivalente au code suivant :
5 container = range(start, stop, end):
6 iterator = iter(container)
7 try:
8     while True:
9         i = next(iterator)
10        # ...
11 except StopIteration:
12     pass
```

Dès lors, la boucle `for` dans le code de l'énoncé va itérer sur les valeurs suivantes de `i` : 40, 42, 44, 46, 48, 50, 52, 54, 56, 58 ; ce qui donne 10 tours de boucle, contre 5 dans le code C++ de l'exercice précédent.

La deuxième différence étant l'absence de shadowing :

- la variable `i` utilisée pour la boucle (l. 7) est la même que celle déclarée à l'entrée de la fonction `main` (l. 2) ;
- la variable `k` modifiée dans la boucle (l. 8) est la même que celle déclarée à l'entrée de la fonction `main` (l. 4) ;
- la variable `j` modifiée dans le `if` (l. 26) est la même que celle déclarée à l'entrée de la fonction `main` (l. 3) et modifiée dans la boucle (l. 12).

Finalement, afin de comparer l'état des variables par rapport au code C++ de l'exercice précédent :

- la valeur de `i` après la boucle `for` (l. 19) vaut 61 ;
- la valeur de `j` après la boucle `for` (l. 20) vaut 60 ;
- la valeur de `k` après la boucle `for` (l. 22) vaut 58 ;

— la valeur de `l` après la boucle `for` (l. 22) vaut 58 ;
— la valeur de `i` après la condition `if` (l. 28) vaut 666 ;
— la valeur de `j` après la condition `if` (l. 29) vaut 666.

Exercice 21.

```
1 int main() {
2     int x = 0;
3     if(x == 0) {
4         int x = 1;
5         return x;  // la variable définie à la ligne 4
6     }
7     return x;  // la variable définie à la ligne 2
8 }
```

Exercice 22. Lors de la déclaration d'une variable, le mot-clef `static` peut désigner deux choses différentes :
— sur une variable globale, `static` signifie que la variable ne peut être accédée que depuis le fichier dans lequel elle a été déclarée ;
— sur une variable locale à un bloc, `static` signifie que la variable a la *durée de vie* d'une variable globale mais a une portée limitée à son bloc de définition (et ses sous-blocs).

Le mot-clef `extern` signifie que la variable déclarée existe mais est définie dans une autre unité de compilation (dans un autre fichier).

Ces deux mots-clefs sont donc exclusifs : si une variable globale est déclarée `static` dans un premier fichier, et qu'une autre variable du même nom est déclarée `extern` dans un second fichier, la compilation des deux fichiers en fichiers objets se passera correctement, mais lors de l'*édition des liens*, la variable `extern` ne sera jamais trouvée :

main.cpp

```
1 int main() {
2     extern int x;
3     int y = x;
4     return 0;
5 }
```

x.cpp

```
1 static int x;
```

```
g++ main.cpp -c -o main.o
g++ x.cpp -c -o x.o
g++ main.o x.o -o ex37
```

Il faut donc bien comprendre que la déclaration d'une variable `extern` signifie au compilateur que cette variable existe quelque part et qu'il ne doit pas s'en occuper dans l'immédiat, mais du coup aucune allocation n'est faite car aucune nouvelle variable n'est *créée*.

L'output du programme est :

```
1
2
1
1
```

Le premier appel à `func2` va incrémenter la variable statique `c` qui aura été initialisée à 0 et ensuite la retourner (donc va retourner la valeur 1), et le deuxième appel à `func2` va à nouveau incrémenter cette variable et retourner la valeur (donc retourner 2). En effet, la variable `c` n'est pas remise à 0 lors du second appel à `func2` puisque la variable est *statique*, c.-à-d. sa durée de vie est celle d'une variable globale. La variable n'ayant pas été détruite/désallouée, elle existe toujours lors du deuxième appel et son contenu n'a pas été écrasé.

Exercice 23. Pour la fonction `f1()` nous voulons que la variable declarée continue à exister après que la fonction ait terminé son exécution. Cette dernière partie suggère la nécessité du mot-clé `static`.

```
int f1(int a) {
    static int x1 = a;
    x1--;
    return x1;
}
```

La fonction `f2()` déclare une variable également appelée `x1` en l'initialisant à 0 puis l'incrémente de 1. Cependant, nous ne voulons pas changer la valeur de la variable `x1` stockée avec le mot-clé `static`.

```
int f2() {
    int x1 = 0;
    x1++;
    return x1;
}
```

Le code dans son intégralité devient le suivant :

```
#include <iostream>
using namespace std;

int f1(int a) {
    static int x1 = a;
    x1--;
    return x1;
}
int f2() {
    int x1 = 0;
    x1++;
    return x1;
}

int main() {
    f1(3);
    int x2 = f1(f2());
    cout << x2 << endl;
    return 0;
}
```

Il semblerait intuitif de s'attendre à ce que `x2` prenne la dernière valeur passée en argument à `f1()` mais, comme la variable dans `f1()` est statique et sa valeur existe déjà, elle n'est pas réassignée. La valeur affichée sera donc 1 (c.-à-d., $3 - 1 - 1$) et pas 0.

Séance 4 — Les conditions et les boucles

Exercice 24. Nous allons simplement gérer les différents cas à l'aide de `if`, `else if` et `else` :

Programme 15 – tri.cpp

```
#include <iostream>

int main(int argc, char ** argv) {
  int a, b, c;
  std::cin >> a;
  std::cin >> b;
  std::cin >> c;
  if (a <= b) {
    if (a <= c) {
      std::cout << a << " ";
      if (b <= c) {
        std::cout << b << " " << c << std::endl;
      } else {
        std::cout << c << " " << b << std::endl;
      }
    } else {
      std::cout << c << " " << a << " " << b << std:::endl;
    }
  } else {
    if (b <= c) {
      std::cout << b << " ";
      if (a <= c) {
        std::cout << a << " " << c << std::endl;
      } else {
        std::cout << c << " " << a << std::endl;
      }
    } else {
        std::cout << c << " " << b << " " <<  a << std::endl;
    }
  }
  return 0;
}
```

Si nous voulons utiliser les paramètres donnés au programme, nous devons utiliser les paramètres de la fonction `main`. Celle-ci peut s'écrire de deux façons différentes, soit :

```
int main(int argc, char ** argv)
```

soit

```
int main(int argc, char * argv[])
```

Le premier paramètre de la fonction main (`argc`) consiste en le nombre de paramètres passés programme tandis que le deuxième paramètre (`argv`) consiste en un tableau contenant les paramètres passés au programme - `argv[0]` étant le nom du programme et `argv[1]` à `argv[argc-1]` étant les véritables paramètres du programme. Ceux-ci sont des chaînes de charactères, ç-à-d des pointeurs vers des charactères, comme

vous le verrez dans le chapître sur les pointeurs (le pointeur étant représenté par une astérisque). Elles sont converties en entiers grâce à la fonction `atoi`. De même, `argv` est un tableau de `char *`, et peut être représenté par un pointeur vers `char *` (c-à-d `char **`) ou à l'aide de crochets (voir chapître sur les tableaux et les pointeurs).

Le programme `tri.cpp` ci-dessous est compilé à l'aide de la commande :

```
g++ tri.cpp -o tri
```

et est appelé, par exemple de la manière suivante, pour trier les trois nombres 6, 2 et 3 :

```
./tri 6 2 3
```

ce qui donne en sortie :

```
2 3 6
```

Nous avons besoin de la fonction `int atoi(const char*);` qui prend une chaîne de caractères en paramètre et renvoie l'enter qui y est représenté :

Programme 16 – tri.cpp

```
#include <iostream>
#include <cstdlib>  /*atoi*/

using namespace std;

int main(int argc, char ** argv) {
  int a = atoi(argv[1]);
  int b = atoi(argv[2]);
  int c = atoi(argv[3]);
  if (a <= b) {
    if (a <= c) {
      cout << a << " ";
      if (b <= c) {
        cout << b << " " << c << endl;
      } else {
        cout << c << " " << b << endl;
      }
    } else {
      cout << c << " " << a << " " << b << endl;
    }
  } else {
    if (b <= c) {
      cout << b << " ";
      if (a <= c) {
        cout << a << " " << c << endl;
      } else {
        cout << c << " " << a << endl;
      }
    } else {
        cout << c << " " << b << " " <<  a << endl;
    }
```

```
  }
  return 0;
}
```

Notez que nous avons utilisé l'espace de nommage `std` (grâce à l'instruction `using namespace std`) afin de ne pas devoir le répéter devant tous les appels à `cin`, `cout` et `endl`.

Exercice 25. Nous avons raccourci le programme ici, de sorte qu'il ne montre un prénom que pour les lettres A et B.

Programme 17 – prenom.cpp

```
#include <iostream>

using namespace std;

int main() {
  char c;
  cout << "Veuillez entrer un caractère (majuscule): ";
  cin >> c;
  switch(c) {
    case 'A':
      cout << "Aline" << endl;
      break;
    case 'B':
      cout << "Boris" << endl;
      break;
    default:
      cout << "Désolé le programme ne marche que pour A et B" <<
          endl;
  }
  return 0;
}
```

Notons qu'il n'y a pas besoin d'ajouter un `break` dans le `default`, mais que si le `break` est omis dans un des `case`, alors à la fin du bloc d'exécution, le `case` suivant sera exécuté. La raison est qu'ainsi, plusieurs `case` peuvent donner lieu au même code exécuté sans devoir le réécrire. Par exemple, le code suivant permet de faire la même chose que le programme précédent mais en gérant également les lettres minuscules.

```
switch(c) {
  case 'a':
  case 'A':
    cout << "Aline" << endl;
    break;
  case 'b':
  case 'B':
    cout << "Boris" << endl;
    break;
  default:
    cout << "Désolé le programme ne marche que pour A et B" <<
         endl;
}
```

Exercice 26.

Programme 18 – fibonacci_while.cpp

```cpp
#include <iostream>

int main() {
    unsigned int prevprev = 0;
    unsigned int prev = 1;
    unsigned int next;
    std::cout << 0 << " " << 1 << " ";
    unsigned int n = 2;
    while(n < 16) {
        next = prev + prevprev;
        std::cout << next << " ";
        prevprev = prev;
        prev = next;
        n++;
    }
    std::cout << std::endl;
    return 0;
}
```

Programme 19 – fibonacci_do_while.cpp

```cpp
#include <iostream>

int main() {
    unsigned int prevprev = 0;
    unsigned int prev = 1;
    unsigned int next;
    std::cout << 0 << " " << 1 << " ";
    unsigned int n = 2;
    do {
        next = prev + prevprev;
        std::cout << next << " ";
        prevprev = prev;
        prev = next;
        n++;
    } while(n < 16);
    std::cout << std::endl;
    return 0;
}
```

Programme 20 – fibonacci_for.cpp

```cpp
#include <iostream>

int main() {
    unsigned int prevprev = 0;
    unsigned int prev = 1;
    unsigned int next;
```

```
    std::cout << 0 << " " << 1 << " ";
    for(unsigned int n = 2; n < 16; n++) {
        next = prev + prevprev;
        std::cout << next << " ";
        prevprev = prev;
        prev = next;
    }
    std::cout << std::endl;
    return 0;
}
```

Exercice 27. Voici la traduction en C++ du programme en Python :

Programme 21 – traduction_python.cpp

```
#include <iostream>
using namespace std;

int main() {
    int n;
    int count = 0;
    cout << "entier positif : ";
    cin >> n;
    while(n != 1) {
        cout << n << " ";
        if (n % 2 == 0)
            n = n / 2;
        else
            n = n*3 + 1;
        count++;
        cout << n << endl;
    }
    cout << count << "itérations" << endl;
    return 0;
}
```

Notez que en C++, contrairement à Python, les `while` ou `if` doivent être suivies d'une condition entre *parenthèses*. La condition – ou le mot clé dans le cas d'un `else` – doit être suivie d'une instruction ou d'un bloc de une ou plusieurs instructions entre accolades.

Exercice 28. Si la fonction $f(n)$ est définie par :

$$f(n) = 1 + \frac{1}{1!} + \frac{1}{2!} + ... + \frac{1}{n!}, \tag{1}$$

on peut écrire :

$$e - f(n) \leq \frac{2}{(n+1)!} \leq \epsilon, \tag{2}$$

où ϵ (epsilon) est la différence maximale autorisée entre le nombre e et son approximation. Il faut donc trouver le plus petit entier n tel que :

$$(n+1)! \geq \frac{2}{\epsilon}. \tag{3}$$

La fonction suivante renvoie l'approximation du nombre e, correcte à un nombre ϵ près, donné en argument :

```
double approxE(double epsilon) {
    double limit = 2./epsilon;
    double e = 1;
    unsigned int n = 0;
    unsigned int nFact = 1;
    while (nFact * (n+1) < limit) {
       n++;
       nFact *= n;
       e += 1./nFact;
    }
    return e;
}
```

Exercice 29. Le programme ci-dessous utilise des générateurs aléatoires de type hardware et software, ici respectivement les variables `rd` et `gen`. Un générateur aléatoire de type hardware (ici via le type `std::random_device`) génère des nombres aléatoires via un processus physique plutôt qu'à l'aide d'un algorithme. Il est généralement basé sur des phénomènes microscopiques qui génèrent des signaux de "bruit" aléatoire dans le hardware de l'ordinateur. En C++, le générateur aléatoire hardware de type `std::random_device` est un objet dont l'opérateur `()` (par exemple, à la ligne 6, `rd()`) retourne un nombre aléatoire de type `unsigned int`. Le problème de ce type de générateur est qu'il ne peut générer des nombres qu'à une fréquence limitée. Pour augmenter la rapidité à laquelle les nombres sont générés, on utilise des générateurs aléatoires de type software (ici via le type `std::mt19937`). Ceux-ci sont également appelés *pseudo*-aléatoires car ils ne sont pas tout à fait non-déterministes. Le générateur aléatoire hardware reste toutefois utilisé pour générer la graine (ou *seed* en anglais) lors de l'initialisation du générateur aléatoire de type software (voir ligne 6). Finalement, un objet de type `std::uniform_int_distribution` permet de générer des entiers aléatoires selon une distribution uniforme entre deux valeurs limites (ici 0 et 100). Pour générer un entier, l'objet de ce type doit être appelé avec un objet de type générateur en argument (voir lignes 8 et 12).

Programme 22 – deviner_nombre.cpp

```
#include <iostream>
#include <random>

using namespace std;

int main() {
    random_device rd;  // générateur aléatoire hardware
    mt19937 gen(rd());  // générateur aléatoire software
    // distribution uniforme
    uniform_int_distribution<> distr(0, 100);
    // nombre aléatoire entre 0 et 100
    unsigned int n = distr(gen);
    unsigned int nombreEssais = 0;
    bool reussi = false;
    while(nombreEssais < 6) {
        cout << "entrez un nombre entre 0 et 100: ";
        int essai;
        cin >> essai;
        nombreEssais++;
        if(essai == n) {
            cout << "Bravo ! Vous avez réussi en " <<
         nombreEssais << " essais!" << endl;
```

```
22              reussi = true;
23              break;
24          } else if (essai < n)
25              cout << "Le nombre n est plus grand" << endl;
26          else
27              cout << "Le nombre n est plus petit" << endl;
28      }
29      if(not reussi)
30          cout << "Vous avez perdu!" << endl;
31  }
```

Séance 5 — Les tableaux et les pointeurs

Exercice 30. Vous savez maintenant enregistrer un entier à l'aide de `std::cin`. Pour récupérer l'adresse d'une variable, la syntaxe en C++ est la suivante `T *ptr = &var;` (où `T` désigne un type quelconque).

```cpp
#include <iostream>

int main() {
  int value;
  int* ptr;

  std::cout << "Entrez un entier: ";
  std::cin >> value;
  ptr = &value;
  std::cout << "La valeur " << value
            << " est stockée à l'adresse: "
            << ptr << std::endl;
  return 0;
}
```

Exercice 31. La syntaxe pour la création d'un tableau est la suivante `T var[size]` où à nouveau `T` désigne un type quelconque et où `size` doit être une valeur entière. [1]

Dans cet exercice, le fait d'avoir une variable constante (ici même `constexpr` car sa valeur est connue à la compilation) pour la déclaration du tableau et pour la borne des boucles permet de n'avoir qu'une unique ligne à changer pour changer la taille du tableau. Si par contre une telle valeur n'était pas définie, il aurait fallu changer la déclaration du tableau, et changer les bornes dans les boucles `for`.

```cpp
#include <iostream>

int main() {
  constexpr unsigned short size = 10;
  int vec[size];

  vec[0] = 0;
  vec[1] = 1;

  for(unsigned short i = 2; i < size; ++i)
    vec[i] = vec[i-1] + vec[i-2];

  std::cout << "Premiers nombres de Fibonacci :" << std::endl;
  for(unsigned short i = 0; i < size; ++i)
    std::cout << vec[i] << " ";
  std::cout << std::endl;
  return 0;
}
```

Exercice 32. En exécutant le code ci-dessous, vous risquez une erreur de violation d'accès mémoire (un SEGFAULT dans le cas des systèmes paginés). Le fonctionnement précis de la mémoire et de la gestion de

1. En C, il faut distinguer les tableaux alloués à la compilation et les tableaux alloués pendant l'exécution : si `size` est *connu* à la compilation, alors le tableau peut être alloué directement sur la pile, alors que si `size` n'est pas connu, il n'est pas possible d'allouer la taille nécessaire sur la pile. On appelle ces tableau des VLAs (pour Variable Length Array).

celle-ci sera vue en bloc 2 dans le cours *Systèmes d'exploitation*. Le résultat est un plantage net du programme car l'OS remarque que le programme essaye d'accéder à une zone mémoire qui ne lui est pas allouée, et sa manière de le gérer est de stopper le programme dissident.

```cpp
#include <iostream>

int main() {
  int vector[5];
  for(unsigned short i=0; i < 5; ++i)
    vector[i] = i;

  for(unsigned short i=0; i < 10000; ++i)
    std::cout << "vector[" << i << "] = "
              << vector[i] << std::endl;

  return 0;
}
```

Cette erreur est ici causée par un mauvais traitement du tableau `vector`, mais peut également être causée par une mauvaise utilisation d'un pointeur, par exemple déréférencer un pointeur qui n'a pas été initialisé ou qui est à `nullptr`.

Exercice 33.

```cpp
#include <iostream>

int main() {
  char text[42];
  unsigned short i = 0;

  std::cout << "Entrez une phrase: ";
  std::cin >> text;

  std::cout << "Print 1 caractère à la fois: " << endl;
  while(text[i] != '\0') {
    std::cout << text[i];
    ++i;
  }
  /* ou de manière plus courte :
  while(text[i] != '\0')
    std::cout << text[i++];
  */
  std::cout << std::endl;

  std::cout << "Print vector: " << std::endl;
  std::cout << text << std::endl;

  cout<<"Print avec arithmétique de pointeurs: "<<endl;
  i = 0;
  while(*(text+i) != '\0') {
    std::cout << *(text+i);
    ++i;
  }
```

```
30    std::cout << std::endl;
31    return 0;
32  }
```

Exercice 34. L'aspect imoprtant de cet exercice est l'allocation dynamique du tableau `phis` contenant les approximations successives du nombre d'or φ. Cette dernière se réalise par l'opérateur `new []` avec la syntaxe suivante : `T *tab = new T[n];` où `T` est une type quelconque, et `n` est la taille du tableau à allouer.

Contrairement à l'allocation *statique* d'un tableau (comme dans l'exercice précédent) où la taille du tableau doit être connue à la compilation, lors d'une allocation *dynamique* d'un tableau, la taille ne doit pas nécessairement être connue à la compilation. C'est d'ailleurs son intérêt principal : lorsque la taille du tableau n'est pas connue à la compilation mais est garantie de ne pas changer pendant l'exécution, il est préférable de passer par une telle allocation plutôt que de passer par une liste chaînée (car l'aspect contingu en mémoire du tableau permet d'être plus rapide).

Attention : contrairement à d'autres langages (tels que Python ou Java par exemple), lorsqu'un espace mémoire est alloué dynamiquement, il faut *désallouer* cet espace lorsqu'il n'est plus utilisé (avec l'opérateur `delete` si l'allocation a été faite par `new` et avec l'opérateur `delete[]` si l'allocation a été faite par `new[]`) afin d'éviter les fuites de mémoire.

```
1  #include <iostream>
2
3  int main() {
4    unsigned size, tmp, fib0 = 1, fib1 = 1;
5    double* phis;
6
7    std::cout << "Entrez une valeur : ";
8    std::cin >> size;
9    phis = new double[size];
10
11   for(unsigned i = 0; i < size; ++i) {
12     phis[i] = static_cast<double>(fib1) / fib0;
13     tmp = fib1;
14     fib1 = fib1 + fib0;
15     fib0 = tmp;
16     std::cout << phis[i] << " " ;
17   }
18   std::cout << std::endl;
19
20   delete[] phis;
21   return 0;
22 }
```

Exercice 35. Désallouer deux fois une seule adresse mémoire (ou désallouer à l'aide de `delete` un pointeur qui n'a pas été alloué avec `new`) est un comportement indéfini selon la norme C++. De manière générale, cela résulte également en un plantage du programme. Il faut donc à la fois faire attention à désallouer la mémoire allouée quand elle n'est plus utilisée, mais également à ne pas désallouer plusieurs fois un pointeur alloué dynamiquement.

Un pointeur ne pointant pas vers une adresse valide est appelée *dangling pointer*. En particulier, après avoir été libéré (avec `delete` en C++ ou avec `free` en C), un pointeur est *dangling* et ne doit pas être lu ou déréférencé car l'adresse qu'il contient ne correspond à rien.

Exercice 36. Le code suivant est simplement une boucle infinie avec une réallocation dynamique d'un pointeur :

```
int main() {
  int* ptr;
  while(true)
    ptr = new int[100];
  return 0;
}
```

Le résultat est que votre OS va au final allouer autant de mémoire pour ce processus qu'il le peut (donc soit autant de mémoire que vous avez de disponible, soit autant de mémoire qui peut être allouée à un unique processus si votre OS y met une limite).

Au contraire, si le `new[]` est directement suivi d'un `delete[]`, la mémoire n'est réallouée qu'après avoir été libérée, et peut donc avoir lieu un nombre arbitraire de fois sans que la mémoire vive ne se retrouve monopolisée.

Séance 6 — Pointeurs, références, const, tableaux et fonctions : aller plus loin

Exercice 37. Les instructions suivantes sont illégales :

1. `j++;` est interdit car `j` est une variable *constante* et ne peut donc être modifiée ;
2. `int &k;` est interdit car une référence doit toujours être initialisée lors de sa déclaration. Cela vient du fait qu'une référence ne peut pas être réassignée ;
3. `int &l = j;` est interdit car `l` est une référence vers un `int` alors que `j` est un `const int`, et donc on perd le qualificateur `const` en faisant cela ;
4. `o[2] = 15` est interdit car `o[2]` est de type *pointeur sur* `int`, et un entier ne peut pas être converti implicitement en pointeur ;
5. `int *const s;` est interdit car `s` est un pointeur constant vers un entier, il ne peut donc pas être réassigné, et doit donc être initialisé lors de sa déclaration ;
6. `r = p;` est interdit car `r` est un pointeur constant, il ne peut donc pas être réassigné.

Les autres instructions sont syntaxiquement et sémantiquement correctes en C++.

Exercice 38. On déclare d'abord les entiers `x`, `y`, `z` pour pouvoir déclarer `t`. Puisqu'un pointeur vers un entier constant se déclare avec `int *const ptr`, un tableau de pointeurs vers des entiers constantes se déclare par `int *const t[]`. Un pointeur vers un pointeur constant vers un entier est donc `int *const *ptr` ;

Les pointeurs vers fonctions ont une syntaxe particulière qui ressemble très fort à un prototype de fonction. Il est à noter que *pointeur vers fonction* n'est pas un type en soi. Il faut préciser le type de retour de la fonction et le type de tous les paramètres pour définir un type.

```
int x, y, z;
int *const t[] = {&x, &y, &z};
int *const *ptr_vers_t2 = &t[2];
double (*ptr_vers_fct)(int *);
```

Il est commun d'utiliser `typedef` pour donner un alias à un type s'il est un peu compliqué ou long à écrire. Typiquement pour définir `ptr_vers_fct`, on aurait pu utiliser :

```
typedef double (*ptr_t)(int *); // on définit le type "ptr_t"
double une_fonction_quelconque(int *p) {
  // ...
}
int main() {
  ptr_t ptr_vers_fct = &une_fonction_quelconque;
  return 0;
}
```

Exercice 39. La fonction `f` prend une *copie* d'un entier en paramètre. Le paramètre `i` est donc une variable locale au sens où toute modification de ce paramètre n'aura aucune incidence sur la variable initiale.

```
void f(int i) { i = 10; }
int main() {
  int var = 5;
  f(var);
  // ici var == 5 et pas 10
  return 0;
}
```

La fonction `g` prend une *référence* vers un entier en paramètre, donc le paramètre `i` **est** la variable qui est donnée en paramètre. Donc toute modification sur le paramètre `i` correspond à cette même modification sur la variable donnée en paramètre. Cependant dans ce cas, on ne peut pas donner une r-value à la fonction.

```
void g(int& i) { i = 10; }
int main() {
  int var = 5;
  g(var);
  // ici var == 10 et plus 5
  g(5);  // provoque une erreur
  return 0;
}
```

La fonction `h` prend également une référence vers un entier, mais un entier *constant*. Ainsi il n'est pas permis de modifier cette variable. La fonction `h` provoque donc une erreur de compilation puisque la référence constante `i` est modifiée alors qu'elle est constante.

Exercice 40. En supposant que `h` est une fonction correcte prenant une référence d'un entier constant en paramètre :

1. `g(3)` ne fonctionne pas car la fonction `g` prend en paramètre une référence vers un entier, ce qui signifie que la fonction est suscieptible de modifier la variable référencée. Or, lorsque l'on appelle la fonction avec la valeur `3`, cette valeur est en fait *hardcodée* dans le binaire qui résulte de la compilation. Par conséquent, il est impossible de le modifier et donc on ne peut pas prendre en paramètre une référence vers cette valeur.
2. `g(b)` car `b` est une référence vers un entier constant, il ne peut donc être modifié. Or, la signature de la fonction `g` suggère que la valeur de la référence peut potentiellement changer.

Attention : `h(3)` est tout à fait valide en C++ ! Il n'est pas possible de passer une r-value à une fonction qui attend une référence non-constante mais il est tout à fait autorisé de le faire si la fonction prend une référence constante en paramètre. En effet, aucune modification de la variable ne peut être faite.

Atention : `h(a)` est aussi valide même si les types ne correspondent pas car la fonction `h` indique simplement via `const` que la variable référencée ne va pas changer.

Exercice 41. La première implémentation prend donc deux pointeurs et doit être appelée en donnant les adresses des variables. La seconde prend des références, donc elle doit être appelée en donnant simplement les variables.

```
#include <iostream>

void swap_ptr(int *a, int *b) {
  int tmp = *a;
  *a = *b;
  *b = tmp;
}

void swap_ref(int& a, int& b) {
  int tmp = a;
  a = b;
  b = tmp;
}

int main() {
  int v1 = 0;
```

```
  int v2 = 1;
  std::cout << "v1: " << v1 << " et v2: " << v2 << std::endl;
  swap_ptr(&v1, &v2);
  std::cout << "v1: " << v1 << " et v2: " << v2 << std::endl;
  swap_ref(v1, v2);
  std::cout << "v1: " << v1 << " et v2: " << v2 << std::endl;
  return 0;
}
```

Exercice 42.

```
#include <iostream>
using namespace std;

int main() {
  int ints[20] = {10, 20, 30, 40, 50, 60, 70, 80, 90, 100, 110,
         120, 130, 140, 150, 160, 170, 180, 190, 200};
  int *intp = ints + 3;
  cout << endl;
  cout << ints << endl;           // 100
  cout << ints[4] << endl;        // 50
  cout << ints+4 << endl;         // 116 (100 + 4*4)
  cout << *ints+4 << endl;        // 14
  cout << *(ints+4) << endl;      // 50
  cout << ints[-2] << endl;       // une valeur non initialisée
  cout << &ints << endl;          // 100
  cout << &ints[4] << endl;       // 116 (100 + 4*4)
  cout << &ints+4 << endl;        // 420 (100 + 4*80)
  cout << sizeof(ints) << endl;   // 80 (4*20)
  cout << (&ints)+4 << endl;      // 420 (100 + 4*80)
  cout << &ints[-2] << endl;      // 92  (100 - 2*4)
  cout << endl;
  cout << sizeof(intp) << endl;   // 4
  cout << intp << endl;           // 112 (100 + 4*3)
  cout << intp[4] << endl;        // 80
  cout << intp+4 << endl;         // 128 (112 + 4*4)
  cout << *intp+4 << endl;        // 44
  cout << *(intp+4) << endl;      // 80
  cout << intp[-2] << endl;       // 20
  cout << &intp << endl;          // 96 (100-4)
  cout << &intp[4] << endl;       // 128 (112 + 4*4)
  cout << &intp+4 << endl;        // 112 (96 + 16)
  cout << &intp[-2] << endl;      // 104 (112 - 8)
  cout << endl;
  return 0;
}
```

Séance 7 — Les fonctions simples et la découpe d'un programme

Exercice 43. la fonction `sq` prend un entier signé en paramètre et renvoie un entier non signé car un carré ne peut être négatif.

```
unsigned int sq(int n) {
  return n*n;
}

int main() {
  int n = -12;
  unsigned int nSq = sq(n);
  std::cout << n << "^2 = " << nSq << std::endl;
  return 0;
}
```

Déclarer la fonction `sq` veut dire *ajouter un prototype contenant la signature de la fonction*. Un prototype contient le type de retour de la fonction suivi du nom de la fonction, et le type de tous les paramètres entre parenthèse. Notons que le nom des paramètres peut également être ajouté mais ne sert pas au compilateur.

```
unsigned int sq(int);

int main() {
  int n = -12;
  unsigned int nSq = sq(n);
  std::cout << n << "^2 = " << nSq << std::endl;
  return 0;
}

unsigned int sq(int n) {
    return n*n;
}
```

Si on se contente uniquement de mettre la fonction `sq` après le `main` sans mettre un prototype, `g++` donnera l'erreur suivante (ou équivalent) : `erreur: « sq » n'a pas été déclaré dans cette portée` précisant qu'il a rencontré le nom `sq` mais que ce nom ne correspond à rien de connu. Le fait de déclarer la fonction avant `main` permet justement de signaler au compilateur qu'une fonction nommée `sq` prenant un unique paramètre de type `int` existe et qu'elle sera définie plus tard.

À l'inverse, si on donne le prototype au compilateur mais que l'on ne lui donne pas le code de la fonction `sq`, alors `g++` donnera un autre message d'erreur : `référence indéfinie vers « sq(int) »`. Ce message-ci dit que le compilateur a été averti de l'existence d'une fonction `sq` prenant un unique paramètre de type `int`, mais qu'il ne le trouve pas.

Exercice 44. le fichier `fctAdd.hpp` est le *header* qui contient la déclaration de la fonction `fctAdd`, et le fichier `fctAdd.cpp` est le fichier source contenant la *définition* de la fonction `fctAdd`.

Programme 23 – fctAdd.cpp

```
int fctAdd(int a, int b) {
  return a+b;
}
```

Programme 24 – fctAdd.hpp

```
int fctAdd(int a, int b);
```

Programme 25 – main.cpp

```
#include <iostream>
#include "fctAdd.hpp"

int main() {
  int a = 14, b = 30, c = fctAdd(a, b);
  std::cout << "14 + 30 = " << c << std::endl;
  return 0;
}
```

Notons qu'il est nécessaire d'inclure `fctAdd.hpp` dans `main.cpp` sinon les mêmes erreurs qu'à l'exercice précédent apparaîtront. Voici le Makefile permettant de compiler ces fichiers. Vous y trouverez la compilation en fichiers objets `*.o` à l'aide du paramètre `-c` donné à `g++` :

```
FLAGS=-Wall -Wextra

all: ex53

ex53: fctAdd.o main.o
  g++ fctAdd.o main.o -o ex53

fctAdd.o: fctAdd.cpp
  g++ $(FLAGS) -c fctAdd.cpp -o fctAdd.o

main.o: main.cpp
  g++ $(FLAGS) -c main.cpp -o main.o
```

Notons que ce Makefile peut largement être amélioré. En effet le nom d'une cible et les dépendances fournies peuvent être récupérées à l'aide des variables spéciales suivantes : `$@`, `$<`, et `$^`. La première correspond au nom de la cible, le second à la première dépendance de la cible, et le dernier à l'ensemble des dépendances, séparées par un espace. Dès lors `g++ -c $< -o $@` correspond à la compilation du fichier `$<` donné en dépendance en fichier objet `$@` donné en nom de cible.

```
FLAGS=-Wall -Wextra

all: ex53

ex53: fctAdd.o main.o
  g++ $^ -o $@

%.o: %.cpp
  g++ $(FLAGS) -c $< -o $@
```

Remarque sur les headers

Notons également que `#include` n'est pas une instruction C++ qui sera exécutée pendant l'exécution du programme. C'est une directive de pré-processing, i.e. une instruction qui sera exécutée *pendant* la compilation. Le résultat de cette directive est l'inclusion *directe* du fichier demandé *dans le fichier source*. Donc

quand votre fichier `main.cpp` commence par `#include <iostream>`, le fichier `iostream` est *copié/collé dans son intégralité* au sein de votre fichier `main.cpp`. C'est la raison pour laquelle les prototypes sont mis dans les headers, et non dans les fichiers sources : on n'inclut **jamais** un fichier source, mais toujours un header. Les fichiers source sont compilés séparément en fichiers objets, mais ne sont jamais inclus. À l'inverse, les headers sont toujours inclus et **jamais** compilés. C'est pour cette raison qu'ils n'apparaissent pas dans les Makefiles.

Remarque sur les Makefiles et les headers

Rappelons ici l'objectif des dépendances dans la définition d'une cible au sein d'un Makefile. Dans une cible `x.o: x.cpp`, on définit que pour réaliser la cible `x.o` (qui est un fichier dans le cas présent), le fichier `x.cpp` est requis. Cela permet également de définir que dès que le fichier `x.cpp` est modifié, alors la cible `x.o` doit être reconstruite (i.e. le code de compilation doit être réexécuté). Cela permet que lors de la recompilation d'un gros projet, seuls les fichiers ayant été modifiés ne soient recompilés, et ainsi cela permet de gagner du temps en recompilations inutiles.

Cependant, si les headers ne sont pas ajoutés dans les dépendances des cibles, lorsque celles-ci sont modifiées, les fichiers objets ne sont pas nécessairement recompilés. Pour vous en convaincre, compilez les fichiers de cet exercice à l'aide du Makefile. Si vous retapez la commande `$ make`, vous serez avertis par : `make: rien à faire pour « all »`. Si maintenant vous allez modifier le fichier `fctAdd.cpp` (par exemple en ajoutant un commentaire), et qu'ensuite vous retapez `$ make`, alors le fichier `fctAdd.o` sera recréé, et l'exécutable `ex53` sera également reconstruit à l'aide du nouveau fichier `fctAdd.o`. Si maintenant vous modifiez le fichier `fctAdd.hpp` et retapez `$ make`, alors à nouveau `make` vous signalera qu'il n'a rien à faire.

Deux solutions s'offrent à nous pour pallier ce problème : la première est d'aller ajouter manuellement les headers inclus dans chaque fichier source dans les dépendances du Makefile. Cette approche est fastidieuse (imaginez si votre fichier source inclut plus de 10 headers de votre projet, la perte de temps se fait vite sentir) et peut mener à des oublis. Il faudrait donc automatiser cela. C'est pour cela que `g++` propose l'option `-MMD` qui génère un fichier `.d` pour chaque fichier `.cpp` compilé au format `cibles: dépendances` listant l'ensemble des dépendances du fichier en question. Il faut alors ajouter tous ces fichiers `.d` au Makefile à l'aide de l'instruction `-include *.d`. Ainsi, dès qu'un fichier source est compilé, la liste de ses dépendances est déterminée automatiquement par `g++`, et peut ensuite être utilisée dans le Makefile qui recompilera correctement les fichiers source en cas de modification des headers. Le Makefile final est donc le suivant :

```
FLAGS=-Wall -Wextra

all: ex53

-include *.d  # on inclut les fichiers de dépendances

ex53: fctAdd.o main.o
  g++ $^ -o $@

%.o: %.cpp
  g++ $(FLAGS) -MMD -c $< -o $@  # Notez bien le -MMD
```

Exercice 45. Dans la résolution de cet exercice, nous utilisons une condition ternaire. En Python, vous avez vu la notation `x = a if cond else b` qui permet d'assigner soit la valeur `a`, soit la valeur `b` à la variable `x` en fonction de la condition booléenne `cond`. Les langages C et C++ permettent le même type d'instruction. La syntaxe est la suivante : `x = cond ? a : b;`. On place donc la condition en premier, suivie d'un point d'interrogation ; on met ensuite la valeur à utiliser si la condition est évaluée à `true`, suivi de deux points, et de la valeur à utiliser si la condition est évaluée à à `false`.

Il est à noter que dans le cas présent, les fonctions sont systématiquement définies avant d'être utilisées. Il n'est donc pas nécessaire de les déclarer au préalable à l'aide de prototypes. Cependant, si vous décidez d'écrire ce code dans plusieurs fichiers, il vous faudra nécessairement faire un header contenant les prototypes.

```
double fabs(double x) {
  return (x<0) ? -x : x;
}

bool almostEqual(double x, double y, double eps) {
  return fabs(x-y) < eps;
}

double squareRoot(double a, double epsilon) {
  double x = 0., y = 1.;
  while(not almostEqual(x, y, epsilon)) {
    x = y;
    y = (y + a/y) / 2;
  }
  return y;
}
```

Exercice 46. Pour transposer la matrice, il faut une double boucle `for` pour itérer sur tous les éléments, avec comme subtilité que `j` (dans la seconde boucle) est initialisé à `i+1` au lieu de 0. En effet, sinon on swap d'abord `M[i][j]` avec `M[j][i]` pour `j<i`, et puis on reswap `M[i][j]` et `M[j][i]` pour `j>i`, i.e. on transpose deux fois la matrice.

notons que l'on définit la taille de la matrice dans une constante `SIZE` pour éviter les constantes *magiques* qui se baladent librement dans le code.

```
#include <iostream>

constexpr unsigned int SIZE = 4;

void transpose(int M[SIZE][SIZE]) {
  int tmp;
  for(unsigned int i = 0; i < SIZE; ++i) {
    for(unsigned int j = i+1; j < SIZE; ++j) {
      tmp = M[i][j];
      M[i][j] = M[j][i];
      M[j][i] = tmp;
    }
  }
}

void printMatrix(int M[SIZE][SIZE]) {
  for(unsigned int i = 0; i < SIZE; ++i) {
    for(unsigned int j = 0; j < SIZE; ++j)
      std::cout << M[i][j] << " ";
    std::cout << std::endl;
  }
}

int main() {
```

```
25   int M[SIZE][SIZE] = {{ 1,  2,  3,  4}, { 5,  6,  7,  8},
26                        {-1, -2, -3, -4}, {-5, -6, -7, -8}};
27   printMatrix(M);
28   transpose(M);
29   printMatrix(M);
30 }
```

C++ propose également la fonction `std::swap` pour échanger les valeurs de deux variables. Cela permet de réécrire la fonction `transpose` comme suit :

```
1 void transpose(int M[SIZE][SIZE]) {
2   for(unsigned int i = 0; i < SIZE; ++i)
3     for(unsigned int j = i+1; j < SIZE; ++j)
4       std::swap(M[i][j], M[j][i]);
5 }
```

Séance 8 — POD

Exercice 47.

```cpp
#include <string>

struct Etudiant {
    std::string prenom, nom;
    unsigned matricule;
    float notes[6];
};

Etudiant creer_sebesta() {
    Etudiant sebesta = {
        "Robert", "Sebesta",
        123456,
        {16.5, 16.5, 16.5, 16.5, 20, 16.5}
    };
    return sebesta;
}
```

Nous pouvons également expliciter quelle valeur correspond à quelle variable :

```cpp
Etudiant creer_sebesta() {
    Etudiant sebesta = {
        .prenom="Robert",
        .nom="Sebesta",
        .matricule=123456,
        .notes={16.5, 16.5, 16.5, 16.5, 20, 16.5}
    };
    return sebesta;
}
```

Et nous pouvons également écrire le tout en un *one-liner* :

```cpp
Etudiant creer_sebesta() {
    return {"Robert", "Sebesta", 123456, {16.5, 16.5, 16.5, 16.5,
          20, 16.5}};
}
```

Exercice 48.

```cpp
void creer_sebesta_ptr(Etudiant* e) {
    e->prenom = "Robert";
    e->nom = "Sebesta";
    e->matricule = 123456;
    for(int i=0; i < 6; ++i)
        if(i != 4)
            e->notes[i] = 16.5;
    e->notes[4] = 20;  // 20/20 en INFO-F105
}

```

```
int main() {
    Etudiant sebesta;
    creer_sebesta(&sebesta);
    return 0;
}
```

Il est également correct de remplacer la notation -> par un déréférencement explicite :

```
void creer_sebesta_ptr(Etudiant* e) {
    (*e).prenom = "Robert";
    (*e).nom = "Sebesta";
    (*e).matricule = 123456;
    for(int i=0; i < 6; ++i)
        if(i != 4)
            (*e).notes[i] = 16.5;
    (*e).notes[4] = 20;  // 20/20 en INFO-F105
}
```

Nous pouvons également faire une assignation sur un type non-trivial (tel qu'une `struct`) :

```
void creer_sebesta_ptr(Etudiant* e) {
    *e = creer_sebesta();
}
```

Exercice 49.

```
struct Personne {
    std::string prenom;
    std::string nom;
    unsigned matricule;
};

struct Etudiant {
    Personne p;
    float notes[6];
};

Etudiant creer_sebesta() {
    Etudiant e = {
        .p={
            .prenom="Robert",
            .nom="Sebesta",
            .matricule=123456
        },
        .notes={16.5, 16.5, 16.5, 16.5, 20, 16.5}
    };
  return e;
}

void creer_sebesta_ptr(Etudiant *e) {
    e->p = {
        .prenom="Robert",
```

```
        .nom="Sebesta",
        .matricule=123456
    };
    for(int i = 0; i < 6; ++i)
        if(i != 4)
            e->notes[i] = 16.5;
    e->notes[4] = 20;
}
```

Exercice 50.

```
struct Personne {
    std::string prenom;
    std::string nom;
};

struct Assistant;

struct Professeur {
    Personne p;
    Assistant** doctorants;
};

struct Assistant {
    Personne p;
    Professeur* superviseur;
};

int main() {
    Professeur superviseurs[] = {
        {"Gianluca", "Bontempi", nullptr},
        {"Tom", "Lenaerts", nullptr},
    };
    Assistant assistants[] = {
        {"Yannick", "Molinghen", &superviseurs[1]},
        {"Gian Marco", "Paldino", &superviseurs[0]},
        {"Robin", "Petit", &superviseurs[1]},
        {"Arnaud", "Pollaris", &superviseurs[0]},
        {"Nassim", "Versbraegen", &superviseurs[1]}
    };
    superviseurs[0].doctorants = new Assistant*[2];
    superviseurs[0].doctorants[0] = &assistants[1];
    superviseurs[0].doctorants[1] = &assistants[3];
    superviseurs[1].doctorants = new Assistant*[3];
    superviseurs[1].doctorants[0] = &assistants[0];
    superviseurs[1].doctorants[1] = &assistants[2];
    superviseurs[1].doctorants[2] = &assistants[4];
    for(int i = 0; i < 2; ++i)
        delete[] superviseurs[i].doctorants;
    return 0;
}
```

Exercice 51.

- `E1 i = c;` n'est pas correct car `c` fait référence à `E2` (qui est une *simple* `enum`, et donc les constantes `a`, `b` et `c` existent dans le *scope* global). Il aurait fallu écrire `E1 i = E1::c;` pour utiliser la constante définie dans `E1` car les `enum class` ne placent les constantes que dans le scope de cette classe.
- `E2 j = c;` est ok.
- `int k = a;` n'est pas correct car `a` n'existe pas dans le scope global, à nouveau il faut utiliser `E1::a`.
- `int l = e;` est ok.
- `E1 x = i;` est ok : une `enum class` est copiable.
- `x = j;` est incorrect car `E1` et `E2` sont deux types différents qui ne sont pas convertibles l'un en l'autre, bien qu'ils soient tous les deux représentés par un entier.
- `E2 y = j;` est ok pour la même raison que ci-dessus.
- `y = i;` est incorrect pour la même raison que ci-dessus.

Exercice 52. L'objectif avec `union`, c'est d'économiser de la mémoire en utilisant la même région de mémoire pour stocker différents objets. Si vous savez que plusieurs objets de votre programme contiennent des valeurs dont les durées de vie ne se chevauchent pas, vous pouvez "fusionner" ces objets dans une `union`.

Lorsque le compilateur alloue de la mémoire pour les unions, il réserve toujours l'espace pour le membre le plus grand. Dans l'exercice, le compilateur alloue par exemple 4 bytes pour *int*, même si la taille d'un *char* est de 1 byte.

L'entier `x` est initialisé à la valeur `321`, ce qui correspond pour les 8 bits de poids faibles (et pour rappel, nous sommes en "little endian") à la valeur `65`. Cette valeur `65` elle-meme correspond en code ASCII au caractère `A`. Lorsque l'on retranche `256` à l'entier `x`, ses 8 bits de poids faibles ne sont pas modifiés (mais bien ses bits de poids forts). Ainsi, la séquence binaire correspondant au caractère `A` est préservée. En revanche, lorsque l'entier `x` est incrémenté (de 1), cela affecte les bits de poids faibles qui représenteront alors la valeur `66` ce qui correspond en code ASCII au caractère `B`.

Exercice 53. Il s'agit d'un programme demandant à l'utilisateur d'introduire successivement deux nombres entiers non signés, le premier nombre devant être strictement inférieur à 64 et le deuxième devant être strictement inférieur à 32.

Ensuite, une `struct S` est déclarée avec deux variables non signées `bf_a` et `bf_b` respectivement sur 6 et 5 bits. Ces variables sont séparées en mémoire par 21 bits de bourrage. Dans la foulée, une instance de cette `struct S` est initialisée à partir des deux nombres entrés précédemment par l'utilisateur.

Ensuite, un masque est appliqué sur le champ de bits de chacun de ces deux nombres pour calculer le reste d'une division par 4.

Enfin, une `union` est utilisée avant d'afficher, sous forme hexadécimale, la séquence binaire correspondant à l'instance de `S`. (Pour l'interprétation, attention à l'endian utilisé !)

En ce qui concerne les valeurs assignables :

- Pour `bf_a`, un entier de 0 à 63 (bornes incluses).
- Pour `bf_b`, un entier de 0 à 31 (bornes incluses).

Séance 9 — Les ADT

Exercice 54. Il est important que les attributs de la classe `Personne` soient privés. Il faut alors un constructeur permettant de donner les informations nécessaires à la création d'un objet, et un destructeur. Étant donné qu'aucune allocation explicite de mémoire n'est réalisée, le destructeur n'a rien d'explicite à faire et peut alors être déclaré `default`.

```cpp
// Personne.hpp

#include <string>

class Personne {
public:
  /**< constructor >**/
  Personne(const std::string&, const std::string&,
           const std::string&, short unsigned,
           const std::string&);
  ~Personne() = default;  /**< destructor */
  void printDetails() const;
private:
  /**< string contenant le titres d'appel*/
  const std::string titre;
  /**< string contenant le nom de famille*/
  const std::string nom;
  /**< string contenant le prenom*/
  const std::string prenom;
  /**< entier contenant l'année de naissance*/
  short unsigned date;
  /**< initialisation du titre d'appel par le string t */
  const std::string etat;
};
```

```cpp
// Personne.cpp

#include <iostream>
#include "Personne.hpp"

Personne::Personne(const std::string& t, const std::string& n,
                   const std::string& p, short unsigned d,
                   const std::string& e):
    titre(t), nom(n), prenom(p), date(d), etat(e) {
}

void Personne::printDetails() const {
  std::cout << titre << ' ' << prenom << ' '
            << nom << " est né en " << date
            << ", il est " << etat << std::endl;
}
```

Voici un exemple d'utilisation de la classe :

```cpp
// main.cpp

#include "Personne.hpp"

int main(){
  Personne p("M.", "Abitbol", "George", 1993, "célibataire");
  p.printDetails();
  return 0;
}
```

Notez que le passage de paramètres pour le constructeur se fait par copie et non par références constantes afin de pouvoir donner directement du texte au constructeur sans devoir passer par des variables supplémentaires.

Notez également que la méthode `printDetails` est déclarée `const`. Cela veut dire qu'elle ne modifie pas l'objet référencé par le pointeur `this`. Puisqu'en réalité, ce `this` est un paramètre *implicite* de votre méthode (i.e. vous ne le passez pas *explicitement* à la méthode, mais le compilateur s'en charge pour vous) est donc de type *pointeur constant* sur `Personne`. Il est une bonne habitude à prendre que de systématiquement déclarer `const` les méthodes qui ne modifient pas l'objet. En particulier tous vos getters doivent être déclarés `const` (à moins qu'ils ne nécessitent une mise à jour de certains attributs, comme un compteur par exemple).

Notez également que `const` fait partie du type de `this`. Dès lors, bien qu'il soit tout à fait posible d'appeler une méthode `const` depuis une méthode non-`const`, l'inverse l'est pas possible. le code suivant ne compilera pas :

```cpp
class C {
public:
  C() = default;
  void foo1()       { foo2(); }
  void foo2() const {}
  void bar1() const { bar2(); }
  void bar2()       {}
};

int main() {
    C c;
    return 0;
}
```

à cause de l'appel à `bar2` dans `bar1`, alors que l'appel à `foo2` depuis `foo1` ne pose aucun problème.

Exercice 55. Ici, seul un constructeur par défaut est nécessaire puisque les objets sont initialisés par la méthode `Etudiant::saisie`. Le destructeur peut être déclaré `default` pour la même raison qu'à l'exercice précédent.

```cpp
// Etudiant.hpp

#include <string>

class Etudiant {
public:
  Etudiant();  /**< constructor */
  ~Etudiant() = default;  /**< destructor */
  void saisie();
```

```
10    void affichage() const;
11    float moyenne() const;
12    bool ex_aequo(const Etudiant&) const;
13  private:
14    std::string nom;
15    std::string prenom;
16    float tabnotes[10];
17  };
```

Notez cependant que dans `Etudiant::Etudiant: nom(""), prenom("") {}`, les initialisation des `std::string`s `nom` et `prenom` correspondent aux constructeurs par défaut. Le constructeur de `Etudiant` peut dès lors, lui aussi, être déclaré `default`.

```
1  // Etudiant.cpp
2
3  #include <iostream>
4  #include "Etudiant.hpp"
5
6  Etudiant::Etudiant(): nom(""), prenom("") {
7  }
8
9  void Etudiant::saisie() {
10   std::cout << "Nom : ";
11   std::cin >> nom;
12   std::cout << "Prenom : ";
13   std::cin >> prenom;
14   for(int i = 0; i < 10; ++i) {
15     std::cout << i << "eme note : ";
16     std::cin >> tabnotes[i];
17   }
18 }
19
20 void Etudiant::affichage() const {
21   std::cout << prenom << ' ' << nom;
22 }
23
24 float Etudiant::moyenne() const {
25   float sum = 0;
26   for(int i = 0; i < 10; ++i)
27     sum += tabnotes[i];
28   return sum / 10;
29 }
30
31 bool Etudiant::ex_aequo(const Etudiant& e) const {
32   return this->moyenne() == e.moyenne();
33 }
```

La méthode `ex_aequo` ici peut prendre une référence constante vers un objet de type `Etudiant` plutôt qu'une copie d'un tel objet car l'objet n'est pas modifié (une copie n'est donc pas nécessaire).

Voici un code permettant de tester cette classe :

```cpp
// main.cpp

#include "Etudiant.hpp"

int main() {
  Etudiant e1, e2;
  std::cout << "Saisie du premier étudiant :\n";
  e1.saisie();
  std::cout << "Saisie du deuxième étudiant :\n";
  e2.saisie();
  std::cout << "Les étudiants ";
  e1.affichage();
  std::cout << " et ";
  e2.affichage();
  std::cout << (e1.ex_aequo(e2) ? " sont " : " ne sont pas ")
            << "ex aequo." << std::endl;
  return 0;
}
```

Exercice 56. L'objectif de cet exercice est de vous faire coder des surcharge d'opérateurs. La classe `Complex` doit alors avoir un constructeur et un destructeur (comme toute classe), un getter et un setter pour chacun de ses deux attributs, et ensuite les opérateurs demandés.

Chacun de ces opérateurs prend en paramètre une référence constante vers un objet de type `Complex`, et renvoie un objet de type `Complex` (ou `bool` pour l'opérateur de comparaison).

```cpp
// Complex.hpp

#include <iostream>

class Complex {
public:
  Complex(float r=0, float i=0);     //*< Construcor
  Complex operator+(const Complex&) const;  //*< Plus operator
  Complex operator-(const Complex&) const;  //*< Minus operator
  Complex operator*(const Complex&) const;  //*< Product operator
  Complex operator/(const Complex&) const;  //*< Division
         operator
  bool operator==(const Complex&) const;    //*< Comparison
         operator
  void setdata(float, float);
  float getreal() const;
  float getimaginary() const;
private:
  float real; //*< Real Part
  float imag; //*< Imaginary Part
};

std::ostream& operator <<(std::ostream &s, const Complex &c);
```

Remarquez également la surcharge de l'opérateur `<<` sur un `std::ostream` permettant l'affichage d'un nombre complexe.

```cpp
// Complex.cpp

#include "Complex.hpp"

#include <cmath>
#include <iomanip>

Complex::Complex(float r, float i): real(r), imag(i) {
}

// (a+bi)+(c+di) = (a+c) + (b+d)i
Complex Complex::operator+(const Complex& c) const {
  Complex tmp;
  tmp.real = this->real + c.real;
  tmp.imag = this->imag + c.imag;
  return tmp;
}

// (a+bi)+(c+di) = (a-c) + (b-d)i
Complex Complex::operator-(const Complex& c) const {
  Complex tmp;
  tmp.real = this->real - c.real;
  tmp.imag = this->imag - c.imag;
  return tmp;
}

// (a+bi)*(c+di) = (ac-bd) + (ad+bc)i
Complex Complex::operator*(const Complex& c) const{
  Complex tmp;
  tmp.real = (this->real * c.real) - (this->imag * c.imag);
  tmp.imag = (this->real * c.imag) + (this->imag * c.real);
  return tmp;
}

// (a+bi)/(c+di) = [(a+bi)/(c+di)] * [(c-di)/(c-di)]
//               = [(a+bi)*(c-di)] / |c+di|
Complex Complex::operator/(const Complex& c) const{
  float div = (c.real * c.real) + (c.imag * c.imag);
  Complex tmp;
  tmp.real = (this->real * c.real) + (this->imag * c.imag);
  tmp.imag = (this->imag * c.real) - (this->real * c.imag);
  tmp.real /= div;
  tmp.imag /= div;
  return tmp;
}

void Complex::setdata(float r, float i) {
  real = r;
  imag = i;
}
float Complex::getreal() const {
```

```
52   return real;
53 }
54
55 float Complex::getimaginary() const {
56   return imag;
57 }
58
59 bool Complex::operator==(const Complex& c) const {
60     return (real == c.real) and (imag == c.imag);
61 }
62
63 std::ostream& operator<<(std::ostream &s, const Complex &c) {
64     s << c.getreal() << '+' << c.getimaginary() << 'i';
65     return s;
66 }
```

Une remarque purement syntaxique : le constructeur proposé prend deux paramètres (respectivement la partie réelle et la partie imaginaire) ayant une valeur par défaut. Remarquez que les valeurs par défaut sont données uniquement dans le header, et non dans le fichier source. Il faut en effet que les valeurs par défaut soient données dans le header, puisque c'est ce dernier qui est importé dans un fichier source ayant besoin de la classe `Complex`. Si les valeurs ne sont pas précisées dans le header, il ne sera pas possible de créer un objet sans préciser les deux paramètres puisque le compilateur ne pourra pas savoir qu'il y a des valeurs par défaut.

De plus, une fois ces valeurs par défaut écrites dans la *déclaration* du constructeur, il ne faut plus les re-préciser dans la *définition* du constructeur.

```
1  // main.cpp
2
3  #include "Complex.hpp"
4
5  int main() {
6    Complex a(7.0f, 5.f); //< Calls Constructor
7    Complex b(8.0f, -3.f);  //< Calls Constructor
8    Complex c(8.0f, -5.f);  //< Calls Constructor
9    //< Calls the overloaded operator<<
10   std::cout << "a = " << a << std::endl;
11   std::cout << "b = " << b << std::endl;
12   std::cout << "c = " << c << std::endl;
13   if(b == c)                //< calls overloaded operator==
14     std::cout << "b == c" << std::endl;
15   else
16     std::cout << "b != c" << std::endl;
17   Complex d;
18   d = a+b;  //< calls overloaded + operator
19   std::cout << "a + b = " << d << std::endl;
20   d = a-b;  //< calls overloaded + operator
21   std::cout << "a - b = " << d << std::endl;
22   d = a*b;  //< calls overloaded * operator
23   std::cout << "a * b = " << d << std::endl;
24   d = a/b;  //< calls overloaded / operator
25   std::cout << "a / b = " << d << std::endl;
```

```
  d = d*b;  // verify that * and / are inverse operators
  std::cout << "a / b * b = " << d << std::endl;
  return 0;
}
```

Exercice 57. Étant donné que cet exercice est assez long, la définition des méthodes est écrite directement avec leur déclaration dans les headers. Bien que cela ne soit pas une bonne pratique en général, cela nous permet de ne pas allonger inutilement ce document et de le garder suffisamment lisible.

1. La classe `Point` ne contient rien d'extravagant : un attribut `_x` pour la coordonnée horizontale et un attribut `_y` pour la coordonnée verticale, ainsi que les getters/setters associés :

```
// Point.hpp

#include <string>
#include <iostream>

class Point {
private:
  double _x;
  double _y;

public:
  // Constructor
  Point(double x=0, double y=0): _x(x), _y(y) {}
  ~Point() = default;   //*< Destructor
  // Copy constructor
  Point(const Point& other): _x(other._x), _y(other._y) {}
  // getters
  double getX() const {
    return _x;
  }
  double getY() const {
    return _y;
  }
  // setters
  void setX(double x) {
    this->_x = x;
  }
  void setY(double y) {
    this->_y = y;
  }
  // overload operator+
  Point operator+(const Point& p) const {
    return Point(this->_x + p._x, this->_y + p._y);
  }
};

std::ostream& operator<<(std::ostream &strm, const Point &p) {
  return strm << "("<< p.getX() << ", " << p.getY() << ")";
}
```

2. La classe `Rectangle` est encore plus simple avec ses 4 attributs et uniquement un constructeur et un destructeur.

```cpp
// Rectangle.hpp

class Rectangle {
public:
  // default constructor
  Rectangle(double x1=0, double y1=0,
            double x2=0, double y2=0):
      _x1(x1), _y1(y1), _x2(x2), _y2(y2) {}
  // default destructor
  ~Rectangle() = default;
private:
  double _x1, _y1,_x2, _y2;
};
```

3. La méthode `ordonner` vérifie d'abord que la coordonnée `x` du premier point est bien inférieure à la coordonnée `x` du second point, et échange ces deux valeurs si ce n'est pas le cas. Ensuite, elle fait la même chose sur la coordonnée `y`.

```cpp
// Rectangle.hpp

class Rectangle {
public:
  // default constructor
  Rectangle(double x1=0, double y1=0,
            double x2=0, double y2=0):
      _x1(x1), _y1(y1), _x2(x2), _y2(y2) {}
  // default distructor
  ~Rectangle() = default;
  void ordonner() {
    if(_x1 > _x2)
      std::swap(_x1, _x2);
    if(_y1 > _y2)
      std::swap(_y1, _y2);
  }
private:
  double _x1, _y1,_x2, _y2;
};
```

4. Afin d'avoir un rectangle bien ordonné dès la construction, il nous suffit d'ajouter un appel à la méthode `ordonner` dans le corps du constructeur :

```cpp
// Rectangle.hpp

class Rectangle {
public:
  // default constructor
  Rectangle(double x1=0, double y1=0,
            double x2=0, double y2=0):
      _x1(x1), _y1(y1), _x2(x2), _y2(y2) {
    ordonner();
```

```
  }
  // default distructor
  ~Rectangle() = default;
  void ordonner() {
    if(_x1 > _x2)
      std::swap(_x1, _x2);
    if(_y1 > _y2)
      std::swap(_y1, _y2);
  }
private:
  double _x1, _y1,_x2, _y2;
};
```

5. Le constructeur prenant deux instances de `Point` peut tout simplement faire appel au premier constructeur (qui se charge déjà d'appeler la méthode `ordonner`) :

```
// Rectangle.hpp

#include "Point.hpp"

class Rectangle {
public:
  // default constructor
  Rectangle(double x1=0, double y1=0,
            double x2=0, double y2=0):
      _x1(x1), _y1(y1), _x2(x2), _y2(y2) {
    ordonner();
  }
  // alternative constructor
  Rectangle(const Point& p1, const Point& p2):
      Rectangle(p1.getX(), p1.getY(), p2.getX(), p2.getY()) {}
  // default distructor
  ~Rectangle() = default;
  void ordonner() {
    if(_x1 > _x2)
      std::swap(_x1, _x2);
    if(_y1 > _y2)
      std::swap(_y1, _y2);
  }
private:
  double _x1, _y1,_x2, _y2;
};
```

6. Finalement, la méthode `translation` doit uniquement ajouter la composante `x` du point P à ses coordonnées `x` et la composante `y` du point P à ses coordonnées `y` :

```
// Rectangle2.hpp

#include "Point.hpp"

class Rectangle {
public:
  // default constructor
```

```
8   Rectangle(double x1=0, double y1=0,
9             double x2=0, double y2=0):
10      _x1(x1), _y1(y1), _x2(x2), _y2(y2) {
11    ordonner();
12  }
13  // alternative constructor
14  Rectangle(const Point& p1, const Point& p2):
15      Rectangle(p1.getX(), p1.getY(), p2.getX(), p2.getY()) {}
16  // default distructor
17  ~Rectangle() = default;
18  void ordonner() {
19    if(_x1 > _x2)
20      std::swap(_x1, _x2);
21    if(_y1 > _y2)
22      std::swap(_y1, _y2);
23  }
24  void translation(const Point& p) {
25    _x1 += p.getX();
26    _x2 += p.getX();
27    _y1 += p.getY();
28    _y2 += p.getY();
29  }
30 private:
31  double _x1, _y1,_x2, _y2;
32 };
```

Séance 10 — Compilation et exécution de programmes en assembleur 80386

Exercice 59. Le registre `EBX` contient la valeur 14 suite à l'instruction `MOV`. Mais les registres `EAX`, `ECX`, et `EDX` contiennent une valeur d'apparence *aléatoire*. En effet, à défaut de mettre une valeur explicite dans un registre, il contient la dernière valeur qui y a été mise. Au lancement d'un programme, les valeurs présentes dans les registres doivent être considérées comme aléatoires. Il est donc tout à fait probable que les valeurs que vous observiez soit différentes de celles de la figure ci-dessous.

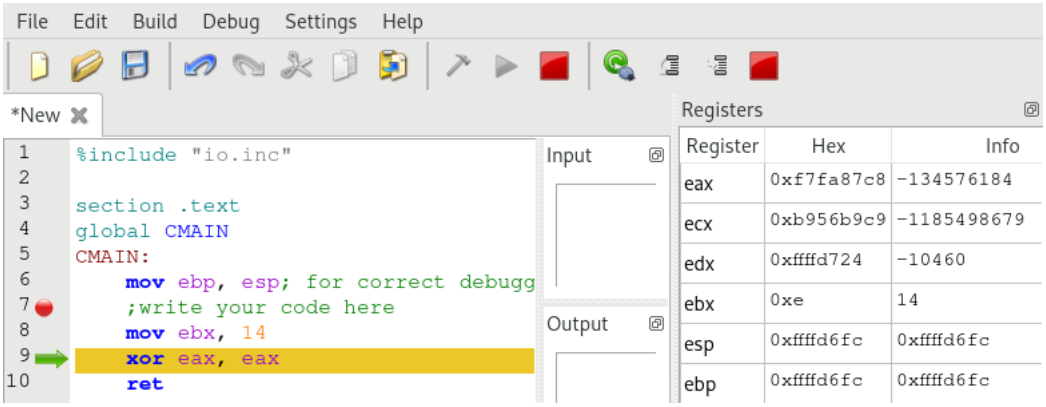

FIGURE 1 – Valeurs dans les registres.

Exercice 60.

```
%include "io.inc"
CPU 386

SECTION .text
GLOBAL CMAIN
CMAIN:
  MOV EAX, 1
  MOV EBX, 2
  MOV ECX, 3
  MOV EDX, 4
  PRINT_UDEC 4, EAX
  NEWLINE
  PRINT_UDEC 4, EBX
  NEWLINE
  PRINT_UDEC 4, ECX
  NEWLINE
  PRINT_UDEC 4, EDX
  XOR EAX, EAX
  RET
```

Exercice 61. Afin de multiplier un nombre par deux, plusieurs solutions sont possibles : (i) `ADD EAX, EAX` ajoute `EAX` à la valeur actuelle de `EAX`, (ii) `MOV EBX, 2` suivi de `MUL EBX` multiplie `EAX` par `EBX` (qui vaut 2), et met le résultat dans `EDX:EAX`, (iii) `SHL EAX, 1` opère un *shift left* (décalage vers la gauche de tous les bits) sur le registre `EAX` qui a pour effet de le multiplier par 2.

La première solution est fonctionnelle mais pas particulièrement optimisée pour la situation : il faut en effet gérer tous les reports (*carry*) des bits à 1. La seconde est encore pire : elel nécessite le registre `EBX` et va écraser ce qui était contenu précédemment dans `EDX`. La dernière solution est la plus efficace.

```
CMAIN:
  MOV EAX, 2020
  SHL EAX, 1
  XOR EAX, EAX
  RET
```

Exercice 62. Réfléchissons de manière binaire : à quoi correspond le fait de prendre un nombre modulo une puissance de 2 ? Le modulo est le reste de la division entière, dès lors n modulo 2^k est la valeur $r < 2^k$ telle que $n = 2^k m + r$ pour un certain m entier. Puisque $2^k m$ correspond aux bits de poids fort jusqu'au bit k, r est en réalité le nombre composé des k bits de poids faible.

par analogie, prendre un certain nombre n modulo 10^k revient à considérer uniquement ses k derniers chiffres.

Dans ce cas-ci, on cherche à prendre `EAX` modulo $32 = 2^5$, donc on veut conserver ses 5 derniers bits. Afin de modifier `EAX` de manière à ce que ses 27 bits de poids fort soient mis à 0, il suffit d'appliquer un masque à l'aide de l'instruction `AND` qui effectue un `ET` logique bit à bit sur un registre. Ce masque doit donc être `0b11111`, i.e. 31.

```
CMAIN:
  MOV EAX, 365
  AND EAX, 31
  XOR EAX, EAX
  RET
```

Exercice 63. La parité d'un nombre est donné par le LSB (*Least Significant Bit*, i.e. bit de poids le plus faible) de sa représentation binaire. Afin de déterminer, il faut donc à nouveau appliquer un masque. Cependant, on veut que `EAX` soit mis à 1 si `EAX` est pair et à 0 s'il est impair. Dès lors récupérer uniquement le dernier bit fera exactement l'inverse. Il faut donc tout d'abord inverser tous les bits de `EAX`, et ensuite appliquer un masque pour récupérer le dernier bit.

En effet, si la valeur initiale de `EAX` est paire, alors inverser tous les bits rendra `EAX` impair, et donc isoler le dernier bit donnera bien 1. À l'inverse, si la valeur initiale de `EAX` ests impaire, alors inverser tous ses bits la rendra paire, en isoler le dernier bit donnera 0.

```
CMAIN:
  MOV EAX, 19   ; on met une valeur arbitraire dans EAX
  NOT EAX       ; on inverse tous les bits de EAX
  AND EAX, 1    ; on applique le masque 1 donc on met les 31
          premiers bits à 0
  XOR EAX, EAX
  RET
```

Exercice 64. Les variables sont déclarées dans la section `.data`. La syntaxe est la suivante : `<nom> Dx [valeur]`, où `Dx` contient deux lettres : la première est le `D` de *Declare*, et la seconde est un indicateur de la taille réservée pour la variable. Les principaux sont décrits ci-dessous :

Lettre	Signification	Taille (bits)
B	Byte	8
W	Word	16
D	Double Word (ou DWord)	32
Q	Quad Word (ou QWord)	64

Pour lire la valeur contenue à une adresse, il faut impérativement utiliser des crochets autour du nom de la variable : `MOV EAX, var` mettra *l'adresse* de `var` dans `EAX`, alors que `MOV EAX, [var]` (en supposant que `var` est déclaré comme étant un DWORD) ira lire la valeur de `var` et la mettra dans `EAX`.

Pour pouvoir échanger les bits de `var`, il va falloir maintenir plusieurs copies de travail, 3 pour être précis : la première va contenir les 5 premiers bits, la seconde les 6 bits du milieu, et la dernière contiendra les 5 derniers bits. À l'aide de rotations, les bits de la première et de la 3e copie de travail seront déplacés à la bonne place. À la fin, il suffit d'appliquer un `OR` pour mettre les bits ensemble.

```
1  CPU 386
2  SECTION .data
3  var DW 1111100011100000b  ; les 5 premiers bits sont à 1, les 6
           bits centraux sont 000 111, et les 5 derniers bits sont
           à 0
4  SECTION .text
5  GLOBAL CMAIN
6  CMAIN:
7    ; on crée les 3 copies de travail dans AX, BX, et CX
8    MOV AX, [var]
9    MOV BX, AX
10   MOV CX, AX
11   ; on isole les bits dans chacune des copies de travail
12   AND AX, 1111100000000000b
13   AND BX, 0000011111100000b
14   AND CX, 0000000000011111b
15   ; on applique les rotations sur AX et CX pour inverser les 5
           premiers et 5 derniers bits
16   ROL AX, 5
17   ROR CX, 5
18   ; on remet les 3 copies ensemble
19   OR BX, AX
20   OR BX, CX
21   ; on remplace la valeur modifiée à l'adresse var
22   MOV [var], BX
23   XOR EAX, EAX
24   RET
```

Séance 11 — Instructions arithmétiques

Exercice 65. On veut additionner deux nombres sur 64 bits. Mais les registres ne sont qu'en 32bits. Il faut donc procéder par étapes :

1. additionner les bits de poids faible
2. conserver le carry
3. additionner les bits de poids fort en ajoutant le carry.

On utilisera l'instruction `ADC` qui s'utilise comme `ADD` à la différence que le contenu du carry flag (`CF`) est également ajouté à la première opérande.

Afin d'accéder aux MSB, il faut aller lire à l'adresse de `N1` décalée de 4 bytes : l'adresse `N1` fait référence au dernier byte, et donc mettre les MSB de `N1` dans le registre `EAX` se fait par `MOV EAX, [N1+4]`.

```
CPU 386
GLOBAL CMAIN
SECTION .data
  N1 DQ 126348 ; sur 64 bits
  N2 DQ 172879 ; sur 64 bits
  N3 DQ 000000 ; sur 64 bits
SECTION .text
CMAIN:
  MOV EAX, [N1]     ; on lit les 32 LSB de N1
  MOV EBX, [N2]     ; on lit les 32 LSB de N2
  ADD EAX, EBX      ; et on les additionne
  MOV [N3], EAX     ; on stocke (N1+N2)l dans (N3)l
  MOV EAX, [N1+4]   ; on lit les 32 MSB de N1
  MOV EBX, [N2+4]   ; on lit les 32 MBS de N2
  ADC EAX, EBX      ; on les additionne et on ajoute le CF
  MOV [N3+4], EAX   ; on stocke (N1+N2)h dans (N3)h
  XOR EAX, EAX
  RET
```

Exercice 66. Pour soustraire `N2` de `N1`, on procède de la même manière sauf qu'on utilise `SUB` et `SBB` à la place de `ADD` et `ADC`.

```
CMAIN:
  MOV EAX, [N1]     ; on lit les 32 LSB de N1
  MOV EBX, [N2]     ; on lit les 32 LSB de N2
  SUB EAX, EBX      ; et on les soustrait
  MOV [N3], EAX     ; on stocke (N1-N2)l dans (N3)l
  MOV EAX, [N1+4]   ; on lit les 32 MSB de N1
  MOV EBX, [N2+4]   ; on lit les 32 MSB de N2
  SBB EAX, EBX      ; on soustrait avec l'emprunt
  MOV [N3+4], EAX   ; on stocke (N1-N2)h dans (N3)h
```

Exercice 67. L'assembleur x86 permet d'aller faire des shifts et rotations directement en mémoire. On fait donc un SHL des LSB, ce qui place le bit sortant dans le `CF`, et puis on fait une rotation avec carry qui donc place le carry dans le bit rentrant et le bit sortant dans le carry.

Notons que lorsque l'on utilise une instruction directement sur la mémoire de la sorte, l'assembleur ne peut déduire la taille de la variable à aller chercher, et il faut l'expliciter : `SHL DWORD [N1]`. À la différence d'un

`MOV EAX, [N1]` dans lequel le nombre de bits qu'il faut aller lire à l'adresse `N1` est donné implicitement par la taille du registre ; sans point de repère, il faut systématiquement expliciter la taille de la lecture/écriture.

```
CMAIN:
  SHL DWORD [N1], 1     ; on left shift les 32 LSB de 1 (ce qui
        met le CF à jour)
  RCL DWORD [N1+4], 1   ; on left rotate de 1 les 32 MSB et le
        carry
```

Exercice 68. Pour faire une rotation, on procède exactement de la même manière, à la différence qu'il faut ensuite ajouter le `CF` aux LSB. Pour ce faire, on emploie l'instruction `ADC [N1], 0` : en effet en procédant de la sorte, on ajoute aux 32 LSB de `N1` la valeur 0 et le `CF`. Et puisque le `CF` contient le bit sortant du `RCL` précédent (i.e. le MSB initial de la valeur à l'adresse `N1`), et que le LSB est mis à 0 par le `SHL`, alors `ADC` permet de mettre le fameux bit sortant tout à droite de la représentation binaire de la valeur à l'adresse `N1`.

```
CMAIN:
  ; SHL des 64 bits de [N1]
  SHL DWORD [N1], 1
  RCL DWORD [N1+4], 1
  ; et on place le bit sortant dans le LSB
  ADC DWORD [N1], 0
```

Exercice 69. Il faut se rendre compte que $\lfloor N2/2^{32} \rfloor$ correspond en réalité aux 32 MSB de N2. Il faut donc aller lire à l'adresse `[N2+4]` et multiplier cette quantité par 237, et ensuite y ajouter N1.

Pour rappel : `IMUL` multiplie la valeur actuelle de `EAX` par l'opérande donnée, et stocke le résultat dans `EDX:EAX`. Il faut en effet se rappeler que le résultat d'une multiplication de deux nombres sur 32 bits a besoin de 64 bits pour être stocké. Par analogie en base 10, le résultat du produit de deux nombres à 3 chiffres a besoin de jusqu'à 6 chiffres pour être écrit.

On place alors 237 dans `EAX`, on multiplie cette valeur par le nombre encodé par les 32 MSB de N2 (notons qu'il faut à nouveau préciser la taille de la lecture à `IMUL` puisqu'aucune référence n'est donnée). Ce résultat est encodé dans `EDX:EAX`. On ajoute donc les 32 LSB de N1 à `EAX`, et puis on ajoute les 32 MSB de N1 à `EDX` (en n'oubliant pas le `CF` !), et on sauve le résultat à l'adresse `N3`.

```
CMAIN:
  MOV  EAX, 237
  IMUL DWORD [N2+4]
  ; ici, EDX:EAX contient floor(N2/2^32)*237
  ADD  EAX, [N1]
  ADC  EDX, [N1+4]
  ; ici, EDX:EAX contient floor(N2/2^32)*237 + N1
  MOV  [N3], EAX
  MOV  [N3+4], EDX
```

Exercice 70. On déclare `N4` comme étant un nombre sur 128 bits comme suit : `N4 DQ <64 LSB>, <64 MSB>`. En effet, un nombre sur 128 bits n'est autre que deux nombres sur 64 bits.

Pour multiplier N5 et N6, on procède comme dans la figure du recueil d'exercices : on calcule d'abord N5 * (N6)l, puis N5 * (N6)h et on additionne les résultats. Il ne faut pas oublier de mettre les 32 MSB de N4 à 0 puisque le produit d'un nombre sur 32 bits avec un nombre sur 64 bits ne prend que 96 bits. Il ne faut donc pas oublier les bits 97 :128.

```
SECTION .data
```

```
2  N4 DQ 0 , 0
3  N5 DD 0xCAFEBABE
4  N6 DQ 0x0BADCAFE
5
6  SECTION .text
7  CMAIN:
8    MOV EAX, [N5]
9    MOV EBX, EAX          ; on conserve N5 puisque EAX sera modifié
               par MUL
10   MUL DWORD [N6]        ; EDX:EAX <- N5 * (N6)l
11   MOV [N4], EAX         ; on stocke les 32 LSB du résultat
12   MOV ECX, EDX          ; ECX <- (N5 * (N6)l)h
13   MOV EAX, [N6+4]       ; on prépare N5 * (N6)h
14   MUL EBX               ; EDX:EAX <- N5 * (N6)h
15   ADD EAX, ECX          ; EAX <- (N5*(N6)h)l + (N5*(N6)l)h
16   MOV [N4+4], EAX       ; on stocke les bits 33:64 du résultat
17   ADC EDX, 0            ; EDX <- (N5*(N6)h)h + CF
18   MOV [N4+8], EDX       ; on stocke les 32 MSB du résultat
19   MOV DWORD [N4+12], 0  ; et on met les 32 MSB à 0
```

Exercice 71. L'avantage ici est que l'on peut s'en sortir avec uniquement 3 appels à `MUL` au lieu de 4 puisque `(N6)l * (N6)h == (N6)h * (N6)l`.

```
1  SECTION .data
2  N4 DQ 0 , 0
3  N6 DQ 0xFFFFFFFFFFFFFFFF
4
5  SECTION .text
6  CMAIN:
7    MOV EAX, [N6]
8    MOV EBX, EAX       ; on conserve (N6)l puisque EAX sera modifié
              par MUL
9    MUL DWORD [N6]     ; EDX:EAX <- (N6)l * (N6)l
10   MOV [N4], EAX      ; on stocke les 32 LSB du résultat
11   MOV ECX, EDX       ; ECX <- ((N6)l * (N6)l)h
12   MOV EAX, [N6+4]    ; on prépare (N6)l * (N6)h
13   MUL EBX            ; EDX:EAX <- (N6)l * (N6)h
14   SHL EAX, 1         ; on double EDX:EAX
15   RCL EDX, 1         ; puisque (N6)l * (N6)h == (N6)h * (N6)l
16   MOV EBX, 0         ; il faut que l'on considère le carry actuel
17   ADC EBX, 0         ; donc on le met dans EBX
18   ADD EAX, ECX       ; EAX <- 2*((N6)h*(N6)l)l + ((N6)l*(N6)l)h
19   MOV [N4+4], EAX    ; on stocke les bits 33:64 du résultat
20   ADC EDX, 0         ; EDX <- 2*((N6)h*(N6)l)h
21   MOV ECX, EDX       ; on sauve EDX
22   MOV EAX, [N6+4]    ; on prépare le produit (N6)h * (N6)h
23   MUL EAX            ; EDX:EAX <- (N6)h * (N6)h
24   ADD EAX, ECX       ; EAX <- ((N6)h*(N6)h)l + 2*((N6)h*(N6)l)h
25   MOV [N4+8], EAX    ; on stocke les bits 65:96 du résultat
26   ADC EDX, EBX       ; EDX <- ((N6)h*(N6)h)h
27   MOV [N4+12], EDX   ; et on stocke les 32 MSB du résultat
```

Exercice 72. On déclare `t` comme un vecteur trois words (16 bits) comme suit : `t DW <t[0]>, <t[1]>, <t[2]>`.

```
SECTION .data
  t DW 0xFFFF, 0x00, 0x00

SECTION .text
  MOV AX, [t]
  MOV DX, AX
  AND AX, 1111100000011111b
  XOR DX, AX
  SHR DX, 5
  MOV [t+2], AX
  MOV [t+4], DX
```

Séance 12 — Les choix et les boucles

Exercice 73. On utilise un *jump conditionnel* : d'abord on utilise l'instruction `CMP` qui met à jour les flags, et ensuite on saute en fonction des valeurs de ces flags. Dans ce cas, on met le maximum entre `A` et `B` dans le registre `AL`, et ensuite on affiche le coutenu de `AL`.

```
SECTION .data
  A DB 10
  B DB 12

SECTION .text
CMAIN:
  MOV AL, [A]
  CMP AL, [B]
  JGE show      ; si A >= B => on affiche AL = [A]
  MOV AL, [B]   ; si A <  B => on affiche AL = [B]
show:
  PRINT_UDEC 1, AL
```

Exercice 74. L'objectif ici est de mettre le plus grand des 3 nombres dans `AX`, le second dans `BX`, et le plus petit dans `CX`. Deux solutions sont proposées ci-après. La première est assez intuitive et teste toutes les combinaisons tandis que la seconde simule un algorithme de tri.

Dans cette première solution, on détermine d'abord lequel de `A` et `B` est le plus grand, et ensuite dans chacune de ces deux situations, on regarde dans lequel des 3 ordres possibles on se situe.

```
SECTION .data
  A DW 12
  B DW 10
  C DW 11

SECTION .text
CMAIN:
  MOV AX, [A]
  MOV BX, [B]
  CMP AX, BX
  JGE AB
BA:             ; B > A
  CMP AX, [C]
  JGE BAC
  CMP BX, [C]
  JGE BCA
  JMP CBA
AB:             ; A >= B
  CMP BX, [C]
  JGE ABC
  CMP AX, [C]
  JGE ACB
  JMP CAB
ABC:            ; A >= B >= C
  MOV AX, [A]
```

```
26    MOV BX, [B]
27    MOV CX, [C]
28    JMP show
29  ACB:              ; A >= C >= B
30    MOV AX, [A]
31    MOV BX, [C]
32    MOV CX, [B]
33    JMP show
34  BAC:              ; B >= A >= C
35    MOV AX, [B]
36    MOV BX, [A]
37    MOV CX, [C]
38    JMP show
39  BCA:              ; B >= C >= A
40    MOV AX, [B]
41    MOV BX, [C]
42    MOV CX, [A]
43    JMP show
44  CAB:              ; C >= A >= B
45    MOV AX, [C]
46    MOV BX, [A]
47    MOV CX, [B]
48    JMP show
49  CBA:              ; C >= B >= A
50    MOV AX, [C]
51    MOV BX, [B]
52    MOV CX, [A]
53    JMP show
54  show:
55    PRINT_UDEC 2, AX
56    NEWLINE
57    PRINT_UDEC 2, BX
58    NEWLINE
59    PRINT_UDEC 2, CX
60
61    XOR EAX, EAX
62    RET
```

Dans cette seconde solution, on considère les trois variables comme une liste et on va effectuer une sorte de tri par insertion. L'idée est que le résultat se trouvera dans `[AX, BX, CX]`, trié dans l'ordre croissant. Ensuite, il suffit de l'afficher d'afficher les valeurs dans l'ordre souhaité (coissant ou décroissant).

```
1  %include "io.inc"
2  section .data
3      A DW 20
4      B DW 10
5      C DW 15
6
7  section .text
8  global CMAIN
9  CMAIN:
10     mov ax, [A]
```

```
11      mov bx, [B]
12      cmp ax, bx
13      ; Si ax <= bx, on ne doit rien faire et on passe directement
            à l'insertion de C
14      jle INSERT_C
15  ; Sinon, on échange ax et bx
16      xchg ax, bx
17
18  INSERT_C:
19      mov cx, [C]
20      cmp ax, cx
21      ; Si ax <= cx, on passe directement à la comparaison avec bx
22      jle COMPARE_BX
23      ;Sinon, on échange les deux positions.
24      xchg ax, cx
25
26  COMPARE_BX:
27      ; On compare maintenant bx avec la variable temporaire cx
28      cmp bx, cx
29      jle END
30      ; Et on les échange si elles sont mal ordonnées
31      xchg bx, cx
32
33  END:
34      PRINT_UDEC 2, cx
35      NEWLINE
36      PRINT_UDEC 2, bx
37      NEWLINE
38      PRINT_UDEC 2, ax
39      NEWLINE
40      xor ax, ax
41      ret
```

N.B. : les solutions ci-dessus sont proposées à titre d'exemple. Ceci étant, des solutions plus optimisées sont également envisageables... A vous de chercher comment faire.

Exercice 75. Dans cet exercice, on va utiliser le calcul d'adresse : si on met dans `EBX` l'*indice* auquel on veut accéder, alors accéder à cet élément dans le vecteur `V` se fait à l'aide de `[V+EBX*2]` puisque chaque entrée de `V` est encodée sur 2 bytes.

Malgré que la valeur à l'adresse `N` soit encodée sur 2 bytes, il faut la mettre dans `EBX` (ou tout autre registre sur 32 bits) pour pouvoir effectuer des calculs d'adressage. On utilise donc l'instruction `MOVZX` pour mettre les 16 bits encodés à l'adresse `N` dans `ECX` (dans `CX` en mettant des 0 dans les 16 MSB).

```
1  SECTION .data
2    N DW 3
3    V DW 2, 4, 8
4
5  SECTION .text
6  CMAIN:
7    MOVZX ECX, WORD [N]
8    XOR   EAX, EAX   ; EAX <- 0
9    MOV   EBX, -1    ; EBX <- -1
```

```
bcle:
  INC   EBX
  TEST  WORD [V+EBX*2], 1
  ; ZF = 1 si [V+EBX*2] pair, 0 autrement.
  ; i.e. le resultat du AND de TEST est 0 si [V+EBX*2] est pair
  LOOPZ bcle   ; Saut à bcle si ECX != 0 et ZF = 1
  JNZ   sinon  ; Saut si ZF = 0 (on rencontre un nombre impair)
  INC   EAX    ; EAX = 1 si on complète la boucle et
               ; tous les nombres sont pairs
sinon:
    PRINT_UDEC 4, EAX
```

Exercice 76. Étant donné que les données sont stockées de manière continguë, on peut se contenter d'une unique boucle sur les $m \times n$ éléments de A et B. On met donc n dans `AX`, et on multiplie par m. On place cette valeur dans `ECX`, ce qui nous permet de faire la boucle à l'aide de l'instruction `LOOP`, et qui nous sert d'indice pour les vecteurs. Notons que l'on accède aux composantes des vecteurs avec `[A+ECX*2-2]` : on multiplie `ECX` par 2 puisque les données sont encodées sur 16 bits, et il faut soustraire 2 puisque le décrément de `ECX` se fait après les accès à `A` et `B`.

```
SECTION .data
  N DW 2
  M DW 2
  A DW 1,2,3,4
  B DW 5,6,7,8
  C DW 0,0,0,0

SECTION .text
CMAIN:
  MOV   AX, [N]
  MUL   WORD [M]
  MOVZX ECX, AX
bcle:
  MOV   AX, [A+ECX*2-2]
  ADD   AX, [B+ECX*2-2]
  MOV   [C+ECX*2-2], AX
  LOOP  bcle
```

Exercice 77. Les éléments de la trace sont espacés de $n+1$ éléments. On va donc utiliser `ESI` pour le décalage, et on va augmenter `ESI` de $n+1$ à chaque fois.

```
SECTION .data
  A DW 1,2,3,4
  N DW 2

SECTION .text
CMAIN:
  XOR   EAX, EAX
  XOR   ESI, ESI
  MOVZX ECX, WORD [N]
  MOV   EBX, ECX
  ; EBX servira d'incrément pour passer à l'élément suivant
  INC   EBX
```

```
bcle:
  ADD   AX, [A+ESI*2]
  ADD   ESI, EBX
  LOOP  bcle
```

Exercice 78. On itère sur le vecteur `V` en partant de la fin, et on met itérativement la paire actuelle (i, x) à sa bonne place à l'aide d'un `XCHG` avec le bon décalage ; et on procède de la sorte jusqu'à ce que `ECX` arrive à 0 ou jusqu'à ce que tous les swaps aient été effectués.

```
SECTION .data
  N DW 10
  V DW 3,10,5,15,4,17,1,9,2,19

SECTION .text
CMAIN:
  MOVZX  ECX, WORD [N]
  SHR    ECX, 1             ; ECX = N/2
  MOV    EDX, ECX
bcle:
  MOV    EAX, [V+ECX*4-4]   ; EAX dest => 2x16 bits déplacés
  ; si l'élément est bien placé (i-ème élément en i-ème position)
  CMP    AX, CX
  JE     fin_bcle
cycle:
  MOVZX  EBX, AX            ; AX contient l'idx du i-ème élement
  XCHG   EAX, [V+EBX*4-4]   ; Swap EAX avec AX-ème pair de V
  DEC    EDX
  CMP    AX, CX             ; si l'élément n'est pas bien placé
  JNE    cycle              ; on boucle
  ; Déplacement de la paire dans la bonne position
  MOV    [V+ECX*4-4], EAX
fin_bcle:
  DEC    EDX                ; Si EDX = 0, alors ZF <- 1
  LOOPNZ bcle               ; Saut à bcle si ECX > 0 et ZF = 0
```

Exercice 79. Afin de déterminer si la matrice est symétrique, on parcourt itérativement les lignes de la matrice, et pour chaque ligne on compare chaque élément avec son symétrique. Pour cela, on maintient les registres `ESI` et `EDI` qui correspondent aux offsets symétriques.

Pour cela, à une ligne donnée, on se place sur la diagonale en ayant `ESI` et `EDI` égaux. Ensuite, on incrémente progressivement `ESI`, et on ajoute n à `EDI` (ce qui correspond à l'élément de la même colonne mais de la ligne suivante). Si une seule des comparaisons est fausse, alors la matrice n'est pas symétrique et on jump en dehors de la boucle. Si la matrice est symétrique, alors l'élément pointé par `ESI` et celui pointé par `EDI` seront systématiquement égaux, et donc on exécute la boucle jusqu'à la fin. Il faut donc mettre `EAX` à 1 après la boucle, mais ne pas passer par cette instruction lorsque l'on quitte la boucle.

```
SECTION .data
  n DD 2
  M DD 1,2,3,4

SECTION .text
CMAIN:
  MOV  ECX, [N]         ; ECX <- n
```

```
8   DEC  ECX              ; ECX <- n-1
9   MOV  EDX, 1           ; EDX <- ligne actuelle
10  XOR  ESI, ESI         ; ESI parcourt chaque ligne
11 bcle1:
12  ; EBX contient le nombre de comp à faire sur la ligne en cours
13  MOV  EBX, ECX
14  MOV  EDI, ESI         ; EDI parcourt chaque colonne
15 bcle2:
16  ADD  EDI, [N]         ; élément suivant de la colonne
17  INC  ESI              ; élément suivant de la ligne
18  MOV  EAX, [M+ESI*4]   ; EAX <- l'élément pointé par ESI
19  CMP  EAX, [M+EDI*4]   ; on compare avec son symétrique
20  JNZ  pas_symetrique   ; si !=, la matrice n'est pas symétrique
21  DEC  EBX              ; on passe à la comparaison suivante
22  JNZ  bcle2            ; si EBX = 0, alors on a fini la ligne
23  INC  EDX              ; on passe à la ligne suivante
24  ADD  ESI, EDX         ; on positionne ESI sur la diagonale
25  LOOP bcle1            ; et on recommence
26  MOV  EAX, 1           ; la matrice est symétrique
27  JMP  fin
28 pas_symetrique:
29  XOR  EAX, EAX         ; la matrice n'est pas symétrique
30 fin:
31  ret
```