INFORMATIQUE SCIENTIFIQUE

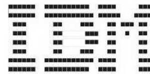
IUT ORSAY

Mesures Physiques - 1^re^année
Tome 1 - TP 1 à 4

version 3.0

http://www.iut-orsay.fr/dptmphy/Pedagogie/Welcome.html

Bob CORDEAU



Avant propos

Lorsque vous avez éliminé l'impossible, ce qui reste, même si c'est improbable, doit être la vérité.
A. Conan Doyle (*Le signe des quatre*)

Prérequis

Avant de commencer cette série de TP, il est important d'avoir lu le poly de conseil qui vous a été distribué.

Structure des TP

Les TP qui accompagnent le cours d'« informatique scientifique » se composent de huit séances de 4h. Ils sont tous organisés de la même façon :

1. un en-tête ;
2. le texte du TP ;
3. un résumé des acquis ;
4. un exercice à préparer.

1 En-tête du TP

Chaque TP définit des objectifs, c'est-à-dire des points particuliers à étudier dans la séance. L'en-tête définit également les fichiers à produire. Leurs noms sont normalisés afin d'être aisément reconnus : ils sont ainsi plus faciles à stocker, à consulter et... à corriger ! Le texte du premier TP précise ce nommage.

2 Texte du TP

Le texte est minutieusement écrit et typographié, il est à lire attentivement. Le texte précise les algorithmes à écrire (des « feuilles d'algorithme » sont disponibles dans la salle) et les sources à développer. On y trouve aussi des copies d'écran illustrant les sorties des programmes, des notes de bas de pages, des encarts, des rappels mathémathiques...

3 Résumé du TP

Un encadré grisé, bien visible sous le titre « Ce qu'il faut retenir », résume les connaissances acquises au cours du TP. Ce résumé doit être acquis pour aborder la séance suivante.

4 Exercice à préparer

Enfin chaque TP (sauf le dernier) se termine par un chapitre « À préparer pour le prochain TP ». Il s'agit le plus souvent d'un algorithme à préparer (et à rendre) sur feuille destiné à faciliter l'étude du prochain TP et à consolider vos acquis.

Informatique :

Rencontre de la logique formelle et du fer à souder.

Maurice NIVAT

SOMMAIRE

- TP 1 : EDI et ARITHMÉTIQUE ÉLÉMENTAIRE
- TP 2 : BOUCLES et TABLEAUX
- TP 3 : PREMIÈRES FONCTIONS
- TP 4 : ENCORE DES FONCTIONS…

EDI & ARITHMÉTIQUE ÉLÉMENTAIRE

Le plus long des voyages commence par le premier pas.
Lao-Tseu

Objectifs :
– utilisation de l'Environnement de Développement Intégré Dev-C++ ;
– utilisation des entrées/sorties standards ;
– arithmétique des entiers et des flottants.

Fichiers à produire : `TP1e0.cpp TP1e1.cpp TP1e2.cpp TP1e3.cpp`

---

1 Chargement, compilation, exécution

Utilisation d'un fichier préparé (`TP1x.cpp`) : cf. figure 1

– Lancez **Dev-C**++ en cliquant l'icône « DevCpp » dans le bureau Windows ;
– chargez le fichier `TP1x.cpp` qui se trouve dans le répertoire `C:\InfoScient` (utilisez l'icône *Ouvrir Projet ou Fichier* de la barre d'outils ou le raccourci `Ctrl-o`) ;
– le lire et comprendre ce qu'il doit faire ;
– le compiler (icône *Compiler* ou `Ctrl-F9`) pour s'assurer qu'il ne contient pas de faute de syntaxe ;
– le sauver dans le répertoire `C:\InfoScient` sous le nom `TP1e0.cpp` (icône *Sauvegarder Sous* ou `Ctrl-F12`) ;
– l'exécuter (icône *Exécuter* ou `Ctrl-F10`) ;

2 Conventions de nommage

Attention

☞ Pour stocker leurs programmes, tous les étudiants utilisent le même répertoire (`InfoScient`), ce qui impose des contraintes. On demande donc que vos fichiers aient des noms normalisés : `TP`**`nep`**`.cpp`, où **`n`** est le numéro du TP (1...8), **`e`** comme étudiant et **`p`** est le numéro du programme du TP. Le premier programme de ce TP doit donc s'appeler : `TP1e1.cpp`, le deuxième `TP1e2.cpp`, etc.

---

3 Votre premier programme (fichier `TP1e1.cpp`)

3.1 Méthode

Vous allez maintenant écrire seul votre premier programme en langage C. Pour cela, vous allez utiliser le modèle qui se trouve dans votre poly « Algorithmes et Conseils de programmation » sous le nom « Exemple de structure type d'un programme C en quatre parties ». Sauf que pour le moment nous n'allons utiliser que les trois premières parties. Alors, à quoi servent-elles ?

Identification Cette partie documente votre programme. On y trouve au moins le nom du programme et de son auteur, mais on peut aussi y faire figurer la date, la version et un descriptif du programme.

En-tête Pour le moment, cette partie ne contiendra que les directives de votre programme, c'est-à-dire des lignes qui commencent par `#define <fichier.h>`. Nous avons besoin au minimum d'inclure le fichier : `iostream` et d'ajouter l'instruction : `using namespace std;` (cette instruction, typique du langage

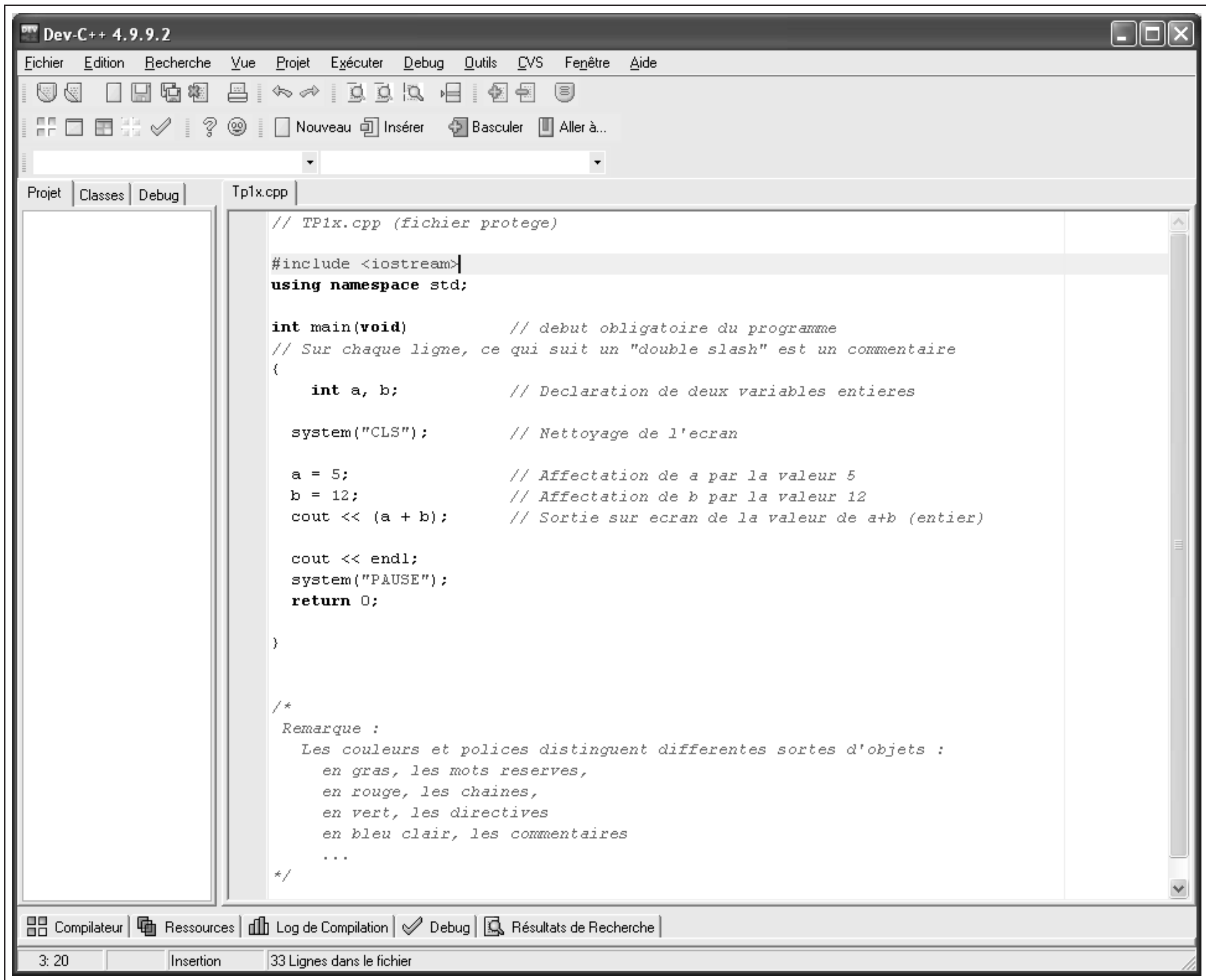


FIG. 1 – Environnement de Développement Intégré (EDI)

C++, permet d'identifier toutes les commandes appartenant à un *espace de noms* standard ; elle nous permet d'éviter une écriture lourde, par exemple l'affichage s'écrira simplement `cout` et non `std::cout`).

Programme principal En langage C, le programme principal est une fonction particulière appelée `main()`. La première ligne de tous vos programmes sera `int main(void)`. Puis suivra une liste d'instructions encadrées d'une paire d'accolades.

Remarque

✎ Pour vous simplifier l'écriture de vos premiers programme, un modèle minimal vous est fourni sous le nom : `_model.cpp`

3.2 À faire :

– ouvrir un nouveau fichier source (`Ctrl-n`) ;
– le nommer `TP1e1.cpp` ;
– écrire tout d'abord les deux premières parties ;
– le programme principal, c'est-à-dire le `main()`, doit déclarer quatre variables :

  - une chaîne de caractère, contenant votre nom, par exemple : `char nom[] = "Dennis Ritchie";`
  - et trois variables entières qui vont contenir le jour, le mois et l'année de la date du TP (par exemple : `int jour = 30;`)
- enfin vous devez affichez le message suivant :
  `Bonjour ! Je m'appelle Dennis Ritchie. Nous sommes le 30/2/2013.`
- Compilez et exécutez votre programme.

Remarque

✎ Quand on écrit des programmes, on réutilise très souvent d'anciens programmes, les siens ou d'autres... Cela implique de maîtriser le *copier/coller*. Par exemple, récupérez la déclaration, l'affectation et l'affichage de `a` et `b` du fichier `TP1x.cpp` et intégrer-les dans votre programme en cours. Compilez et exécutez.

4 Entrées-sorties formatées

4.1 Entrées-sorties

En notation algorithmique, on utilise les instructions `Afficher()` et `Saisir()` pour afficher ou saisir des valeurs. Les instructions équivalentes en C se nomment `cout <<` et `cin >>` .

Elles sont très générales et puissantes car elles fonctionnent pour tous les types.

Exemples :
- affichage d'un message :
  en algo : `Afficher("Entrez une valeur : ")`
  en C : `cout << "Entrez une valeur : ";`
- affichage d'une variable :
  en algo : `Afficher("t[0] vaut : ", t[0])`
  en C : `cout << "t[0] vaut : " << t[0];`
- une saisie :
  en algo : `Saisir(t[0])`
  en C : `cin >> t[0];`

Attention

☞ N'oubliez pas de mettre les deux lignes :

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

en début de vos programme, avant d'utiliser les instructions : `cout` et `cin`

4.2 Entrées-sorties formatées

Le problème à résoudre maintenant est d'être capable de *formater* un affichage, c'est-à-dire d'aligner des colonnes de nombres ou d'imposer deux chiffres après la virgule, etc...

Pour cela, il est nécessaire d'inclure l'en-tête `<iomanip>`. Examinons l'exemple suivant :

```
#include <iostream>
using namespace std;
#include <iomanip>
#include <math.h>

int main(void)
{
   int n = 100;

  cout << "\n ---------------- BASES ------------------\n";
  cout << setiosflags(ios::showbase)
      << "\n " << n << " en base 16 vaut : "
      << hex << n << "\n "
      << dec << n << " en base 8 vaut : "
```

```
     << oct << n << " \n "
     << setbase(10) << n << " en base 10 vaut : "
     << n << "\n\n";

   double x = 0.001234567;

  cout << " ---------------- FORMAT ------------------\n";
  cout << " format par defaut : " << x << "\n"
     << " format scientifique : "
     << setiosflags(ios::scientific) << x << "\n\n";

   double e = exp(1.0);

   cout << " --------------- PRECISION -----------------\n";
   cout << setiosflags(ios::fixed | ios::showpoint)
      << setprecision(2)
      << " e arrondi a 2 decimales : " << e << "\n"
      << setprecision(4)
      << " e arrondi a 4 decimales : " << e << "\n\n";

   cout << " ------------ LARGEUR DE CHAMP --------------\n";
   cout << " idem, mais sur un champ de 10 caracteres avec remplissage\n";
   cout << setprecision(2)
      << " e arrondi a 2 decimales : "
      << setw(10) << setfill('^') << e << "\n"
      << setprecision(4)
      << " e arrondi a 4 decimales : "
      << setw(10) << setfill('^') << e << "\n";

  cout << endl;
  system("PAUSE");
  return 0;
}
```

À l'exécution, le programme `manip.cpp` produit l'affichage de la figure 2 :

```
---------------- BASES ------------------

100 en base 16 vaut : 0x64
100 en base 8 vaut : 0144
100 en base 10 vaut : 100

---------------- FORMAT ------------------
format par defaut : 0.00123457
format scientifique : 1.234567e-03

--------------- PRECISION -----------------
e arrondi a 2 decimales : 2.72
e arrondi a 4 decimales : 2.7183

------------ LARGEUR DE CHAMP --------------
idem, mais sur un champ de 10 caracteres avec remplissage
e arrondi a 2 decimales : ^^^^^^2.72
e arrondi a 4 decimales : ^^^^2.7183
```

FIG. 2 – Exemple d'exécution des manipulateurs de formatage

4.3 À faire :

On va modifier le programme préparé `TP1e0.cpp`. Rechargez-le.

On voudrait que les valeurs a et b ne soient plus figées dans le programme mais choisies par l'utilisateur… Comment faire ? Effectuez les modifications puis compilez, rectifiez les erreurs éventuelles et exécutez.

On voudrait maintenant que ce même programme affiche non seulement la somme de a et b mais aussi leur différence et leur produit… Modifiez, compilez, exécutez.

Attention

☞ Ne pas oublier de sauver votre programme (toujours sous `InfoScient`).

5 Arithmétique, saisies et affichages (fichier `TP1e2.cpp`)

Toujours en suivant le même modèle, vous allez maintenant écrire seul votre deuxième programme décrit ci-dessous.

Division entre flottants ou entre entiers ? Écrire le programme suivant, nommé `TP1e2.cpp`

- déclarez deux entiers `long`, `a` et `b` ;
- déclarez deux flottants `double`, `x` et `y` ;
- Affectez `a` par 5, `b` par 2, `x` par `a` et `y` par `b` ;
- Affichez le message ”Variables entières” ;
- Affichez les valeurs de `a`, `b` et `a/b` ;
- Affichez le message ”Variables flottantes” ;
- Affichez les valeurs de `x`, `y` et `x/y` ;
- N'oubliez pas de terminer votre programme par `system("PAUSE");` puis par `return 0;`.
- compilez, exécutez... et comprenez l'exécution. Qu'en concluez-vous ?

Modifiez le programme pour que l'utilisateur puisse *saisir* les valeurs de a, b, x et y. Faire en sorte que chaque saisie soit précédée d'un message expliquant ce que l'ordinateur attend. Recompilez, exécuter.

6 Division *euclidienne* (fichier `TP1e3.cpp`)

On souhaite écrire un programme qui demande à l'utilisateur la valeur de deux entiers a et b et affiche le résultat de leur division euclidienne sous la forme $a = bq + r$ (q et r sont des entiers et vérifient : $r < b$). Comment calculer q et r ?

Avant de coder le programme, écrire son algorithme. Ensuite seulement, commencez à écrire le programme `TP1e3.cpp` (éventuellement, pour gagner du temps, vous pouvez repartir du programme précédent, le modifier et le sauver sous ce nouveau nom).

Essayez d'obtenir un affichage proche de celui de la figure 3 (où les valeurs 37 et 11 ne sont que des exemples).

```
Entrez un premier entier : 37
Entrez un second entier  : 11

Division euclidienne : 37 = 11*3 + 4
```

FIG. 3 – Division euclidienne

7 Récupération de votre travail sur disquette

Attention

☞ Pour sauvegarder votre travail tout au long des TP, il est **fortement conseillé** de posséder une disquette **individuelle marquée à votre nom**, sur laquelle vous enregistrerez tous vos sources, c'est-à-dire uniquement les fichiers avec l'extension « `.cpp` ».

Ce qu'il faut retenir :

- ☞ comment mettre en route et arrêter une machine ;
- ☞ comment charger, compiler et exécuter un fichier ;
- ☞ la différence entre une affectation par le programmeur et par l'utilisateur ;
- ☞ les entrées-sorties ;
- ☞ la différence entre quotient euclidien et flottant ;
- ☞ enfin qu'il n'y a pas de variable *vide* en informatique.

8 À préparer pour le prochain TP

Écrire l'algorithme correspondant au programme `TP2e3.cpp`.

9 Exercices supplémentaires

Remarque

✎ Ces exercices sont destinés aux étudiants qui connaissent déjà le langage C et qui ont fini de traiter (correctement) les exercices prévus.

1. Écrire un programme (fichier `TP1Se1.cpp`) qui affecte deux flottants a et b puis qui affiche d'une part le résultat de $(a+b)^2$ et d'autre part celui de $a^2 + 2ab + b^2$.

2. Écrire un programme (fichier `TP1Se2.cpp`) qui demande les valeurs de deux résistances R_1 et R_2 et qui donne la résistance équivalente dans le cas où R_1 et R_2 sont en série puis dans le cas où R_1 et R_2 sont en parallèle. Indiquez les unités à la saisie et à l'affichage.

3. Dans un circuit électrique on trouve en série : un générateur alternatif de tension efficace constante, un ampèremètre, une résistance R, une bobine d'inductance L, et un condensateur de capacité C.
   Écrire un programme (fichier `TP1Se3.cpp`) qui demande les valeurs des trois grandeurs R, L et C (saisissez les grandeurs en unités pratiques et convertissez-les pour effectuer les calculs en SI) et qui calcule et affiche (avec l'unité et une précision de deux chiffre après le point décimal) l'impédance de ce circuit pour une fréquence $f = 100$ kHz.
   On rappelle que :

$$Z = \sqrt{R^2 + (L\omega + \frac{1}{C\omega})^2}$$

BOUCLES ET TABLEAUX

Une fois ma décision prise, je m'y conformais strictement.
Harry TRUMAN

Objectifs :
– utilisation des boucles et des tableaux.

Fichiers à produire : `TP2e1.cpp TP2e2.cpp TP2e3.cpp TP2e4.cpp`

---

1 Utilisation simple des boucles

On vous demande d'écrire un programme (fichier `TP2e1.cpp`) dans lequel on va écrire utiliser les trois boucles à connaître.

Sans oublier les conseils du TP précédent (parties 1 et 2 d'un programme type), déclarez dans le `main()`, deux entiers : `i`, un compteur de boucle et `s` une somme.

Dans un premier temps écrire une boucle `Faire .. TantQue` qui calcule la somme des 10 premiers entiers (en commençant à 1). Après l'avoir exécutée sans erreur (vous connaissez la formule qui donne cette somme. . .), réécrivez la même boucle en utilisant la construction `TantQue .. FinTantQue`, puis la construction `Pour`.

```
        Somme des 10 premiers entiers :


Somme (boucle "Faire .. TantQue") : 55
Somme (boucle "TantQue .. FinTantQue") : 55
Somme (boucle "Pour") : 55
```

FIG. 4 – Les trois boucles

2 Affectation partielle d'un tableau et calcul de la moyenne

On veut créer un programme (fichier `TP2e2.cpp`) qui réalise les actions suivantes :
- déclarer un tableau de 100 entiers de type `long` ;
- demander à l'utilisateur combien de variables il souhaite affecter dans le tableau, sa réponse devant être entre 0 et 99 (on supposera que l'utilisateur entre correctement cette valeur, on ne la teste pas) ;
- affecter le nombre de variables souhaité par l'utilisateur par des entiers au hasard entre -10 (compris) et 10 (compris) ;
- afficher les variables du tableau qui sont affectées. . . (pas les autres !) ;
- calculer et afficher la moyenne des contenus des variables affectées du tableau (sachant que cette moyenne doit être proche de 0) ;
- écrire l'algorithme correspondant à ce programme ;
- écrire le programme.

3 Affectation partielle d'un tableau et calcul de la moyenne courante

Dans ce programme (fichier `TP2e3.cpp`) on réalise les actions suivantes :

- déclarer un tableau de 100 entiers de type `long` ;
- puis écrire la boucle suivante :

Faire
<saisir un entier val >
Si $(val \neq 0)$
<ranger val dans le tableau>
<calculer le nombre de valeurs saisies, leur somme et leur moyenne>
FinSi
<afficher le nombre de valeurs saisies non nulles, leur somme et leur moyenne>
TantQue$(val \neq 0)$

Attention

☞ Écrire *complètement* l'algorithme *puis* le programme.

4 L'approximation des nombres réels

4.1 Le problème de la précision

Voici un algorithme connu d'échange de la valeur de deux variables. Nous allons voir que ce n'est pas un *bon* algorithme. En effet son comportement n'est pas *stable*, il dépend de la valeur des variables !

DébutProgramme
DébutDéclaration
a, b : entier
FinDéclaration

$a \leftarrow A$
$b \leftarrow B$

$a \leftarrow b - a$
$b \leftarrow b - a$
$a \leftarrow a + b$
FinProgramme

Cet algorithme est mis en œuvre dans le programme suivant (Voir aussi fig.5) :

```
// precision.cpp
// Bob

#include <iostream>
using namespace std;

int main(void)
{
   float x, dx;
   double xx, dxx;

  cout << " -------- float -------- ";
  x = 1.;
  dx = 1e-10;
  cout << "\n Avant echange : \n\t x = " << x << "\t dx = " << dx;
  x = dx - x;
  dx = dx - x;
```

```
  x = x + dx;
  cout << "\n Apres echange : \n\t x = " << x << "\t dx = " << dx;

  cout << "\n\n -------- double ------- ";
  xx = 1.;
  dxx = 1e-10;
  cout << "\n Avant echange : \n\t xx = " << xx << "\t dxx = " << dxx;
  xx = dxx - xx;
  dxx = dxx - xx;
  xx = xx + dxx;
  cout << "\n Apres echange : \n\t xx = " << xx << "\t dxx = " << dxx;

  cout << endl << endl;
  system("PAUSE");
  return 0;
}
```

```
-------- float --------
Avant echange :
        x = 1   dx = 1e-10
Apres echange :
        x = 0   dx = 1

-------- double -------
Avant echange :
        xx = 1  dxx = 1e-10
Apres echange :
        xx = 1e-10      dxx = 1_
```

FIG. 5 – Problème de précision...

Surprise ! En effet, si la valeur de dx est trop petite, après échange, x vaut 0. Qu'en déduisez-vous ? Le type informatique est-il en cause ou bien est-ce l'algorithme ?

4.2 Calcul de l'*epsilon-machine*

Il est maintenant évident que l'on ne peut espérer effectuer des calculs avec une précision aussi grande que voulue. Combien peut-on espérer ?

On va définir l'*epsilon-machine* comme la plus grande valeur ε telle que[1] :

$$1 + \varepsilon = 1$$

Pour cela, dans votre dernier programme (fichier `TP2e4.cpp`), déclarez une variable `dx` de type `double` initialisée à la valeur `1.0`. Puis dans une boucle `TantQue .. FinTantQue` divisez `dx` par `2.0` tant que la condition `(1.0 + dx > 1.0)` est vraie.

Combien trouvez-vous ?

Ce qu'il faut retenir :

[1] Cette définition est *conventionnelle* car si on définit l'epsilon machine autour d'une valeur différente de 1 on trouve des valeurs légèrement différentes

- ☞ bien connaître les trois boucles et leur équivalence ;
- ☞ les trois champs de la boucle `for` sont séparés par des point-virgules ;
- ☞ un tableau de n cases d'un type donné est équivalent à n variables de ce type ;
- ☞ les indices d'un tableau de dimension n sont notés entre crochets et vont de 0 à $n-1$;
- ☞ la fonction `rand()%n` donne un `int` aléatoire entre 0 et $n-1$. Pour éviter de générer la *même* séquence aléatoire entre deux exécutions, il faut utiliser la fonction `srand()`.
- ☞ Les *flottants* informatiques ne sont qu'une approximation des *réels* mathématiques ; leur précision est *finie* et dépend du type choisi.

5 À préparer pour le prochain TP

5.1 Saisie conditionnelle

On dispose de deux dés ordinaires (à six faces) d_1 et d_2. On choisit l'un de ces dés, on le lance et on enregistre le nombre de points portés sur sa face supérieure.

Un programme est destiné à enregistrer (et *uniquement* à enregistrer) les scores de ces deux dés dans deux tableaux différents appelés `d1` et `d2`. À chaque lancer d'un dé l'utilisateur annonce au programme s'il s'agit du dé n° 1 ou n° 2 puis il donne le nombre de points.

On voudrait que le programme sorte de la boucle de saisie si le numéro de dé indiqué est `0` (saisie *conditionnelle*) et qu'alors il annonce pour chacun des deux dés le nombre de valeurs saisies et la moyenne des valeurs saisies... si cette moyenne est possible à calculer. On utilise la méthode suivante :

- quelles variables ce programme va-t-il utiliser ? Donner leurs types, leurs noms, leurs significations ;
- écrire *l'algorithme* du programme en respectant les niveaux d'indentation (un codage en C de cet algorithme donnerait les figures 6(a) et 6(b)).

```
Quel de ? 2
Quelle valeur ? 4

Quel de ? 1
Quelle valeur ? 3

Quel de ? 1
Quelle valeur ? 4

Quel de ? 2
Quelle valeur ? 6

Quel de ? 0


Nombre de valeurs pour le de numero 1 : 2
Moyenne pour le de numero 1 : 3.500

Nombre de valeurs pour le de numero 2 : 2
Moyenne pour le de numero 2 : 5.000
```

(a) Tirage avec deux dés

```
Quel dé ? 1
Quelle valeur ? 5

Quel dé ? 1
Quelle valeur ? 3

Quel dé ? 0


Nombre de valeurs pour le dé n°1 : 2
Moyenne pour le dé n°1 : 4

Nombre de valeurs pour le dé n°2 : 0
Pas de moyenne pour le dé n°2
```

(b) Tirage d'un seul dé

FIG. 6 – Saisie conditionnelle

5.2 Introduction aux fonctions

Faire les « exercices d'application » du TP n° 3.

6 Exercices supplémentaires

Ces exercices sont destinés aux étudiants qui connaissent déjà le langage C et qui ont fini de traiter (correctement) les exercices prévus.

1. Écrire un programme (fichier `TP2Se1.cpp`) qui demande un entier de type `long` n positif non nul et qui affiche la somme des inverses des carrés des entiers de 1 à n. En déduire une valeur approchée de π, sachant que :
$$\sum_{n=1}^{\infty} \frac{1}{n^2} = \frac{\pi^2}{6}$$
2. Écrire un programme (fichier `TP2Se2.cpp`) qui saisit un entier `n` de type `unsigned long` n dans l'intervalle $[1..12]$ et qui affiche la valeur de $n!$ (pourquoi ne peut-on prendre $n > 12$?).
3. Retrouvez π par la formule de Wallis :
$$\lim_{p \to \infty} \frac{(2 \times 4 \times ... \times 2p)^2}{(1 \times 3 \times ... \times (2p-1))^2 \times (2p+1)} = \frac{\pi}{2}$$
Attention : ne calculez pas séparément le numérateur puis le dénominateur, vous risqueriez un dépassement de limite des entiers. À chaque tour de boucle, calculez le rapport $(\frac{2p}{2p-1})^2$ à multiplier par le produit courant, puis en sortie de boucle, divisez le résultat par la quantité qui convient. Pour faire des essais significatifs, utilisez des flottants `double` et des entiers `long`. $\triangleleft$

PREMIÈRES FONCTIONS EN C

La grammaire, qui sait régenter jusqu'aux rois...
MOLIÈRE (Les femmes savantes, Acte II, scène 6)

Objectifs :
– utilisations des fonctions de la bibliothèque standard ;
– utilisations de fonctions personnelles ;
– prototypage des fonctions et passage des arguments.

Fichiers à produire : `TP3e1.cpp TP3e2.cpp TP3e3.cpp TP3e4.cpp`

---

1 Appels de fonctions

L'emploi des *fonctions* augmente largement ses possibilités d'écriture *structurée* d'un programme. On a toujours avantage à découper un gros programme en plusieurs fonctions : le programme principal est plus simple et plus clair, les actions effectuées par les fonctions restent limitées, faciles à écrire et à maintenir, enfin seules leurs interfaces (*prototypes*) doivent être connues du développeur, ce qui permet de *masquer* les détails du codage.

Pour cette initiation, nous allons distinguer deux types de fonctions : celles qui sont déjà fournies par tout compilateur standard, puis celles que l'on écrit pour nos propres besoins.

1.1 Fonctions de la bibliothèque standard (fichier `TP3e1.cpp`)

Avant d'utiliser une fonction il faut la *déclarer*, c'est-à-dire donner son prototype. Les prototypes de toutes les fonctions de la bibliothèque standard sont disponibles dans des fichiers d'en-tête[1] que l'on doit inclure en début de programme. Par exemple, avant d'utiliser la fonction `isdigit()`, vous devez inclure le fichier `<ctype.h>` car c'est lui qui déclare cette fonction[2].

À titre d'exemple, on demande de déclarer dans le programme principal (fichier `TP3e1.cpp`) deux `double` a et b et deux seulement, de saisir leur valeur, de calculer la longueur de l'hypoténuse d'un triangle rectangle de côtés $|a|$ et $|b|$ en employant la fonction `sqrt()`, puis de l'afficher.

1.2 Fonctions fournies (fichier `TP3e1.cpp` *suite*)

On va continuer à enrichir le programme `TP3e1.cpp`.

Le fichier `TP3.cpp`, à inclure et à **ne pas** modifier, contient plusieurs fonctions déjà écrites et cet exercice consiste à les utiliser, par des appels, en ne connaissant que leurs « modes d'emploi[3] » :

- Procédure `Centrer()`.

  Prototype : `void Centrer(char s[]);`

  Cette fonction reçoit en argument une chaîne de caractères (disons s) contenant au maximum 80 caractères ; elle ne retourne rien ; elle affiche la chaîne s en la centrant dans la ligne où se trouve le curseur, puis elle place le curseur au début de la ligne suivante ;

---

[1] en anglais *header*
[2] Pensez à utiliser l'aide en ligne pour savoir dans quel en-tête est déclarée une fonction que vous voulez utiliser.
[3] On distingue le *prototype* qui donne les conditions d'emploi (c'est-à-dire le nombre et le type de chaque argument) du *mode d'emploi* qui, lui, indique la signification des paramètres.

- Procédure `Ecritxy()`.
  Prototype : `void Ecritxy(char s[], int x, int y);`
  Cette procédure reçoit trois arguments : une chaîne de caractères (disons s) contenant un nombre quelconque de caractères, un entier x et un autre entier y ; elle ne retourne rien ; elle affiche la chaîne s à partir de la position $(x; y)$ c'est-à-dire colonne x et ligne y pourvu que x soit entre 1 et 80 et y entre 1 et 25 ;
- Procédure `AfficheTab()`.
  Prototype : `void AfficheTab(int tab[], int max, int n);`
  Cette fonction reçoit trois arguments : un tableau de MAX entiers, la valeur de MAX, le nombre n de variables réellement utilisées dans le tableau ; elle ne retourne rien ; elle affiche successivement les variables réellement utilisées dans le tableau ;
- Fonction `MoyTab()`.
  Prototype : `double MoyTab(int tab[], int max, int n);`
  Cette fonction reçoit trois arguments : un tableau t de MAX entiers, la valeur de MAX, le nombre n de variables réellement utilisées dans le tableau (on suppose ce nombre strictement positif). Elle retourne la valeur moyenne des n premières variables de t. Si n est supérieur à MAX (ce qui serait une faute de programmation) la fonction ne calcule la moyenne que des MAX premières variables de t.

Affichez au centre de la première ligne le message « IUT ORSAY » et au centre de la deuxième ligne le message « MESURES PHYSIQUES ». N'oubliez pas, bien sûr, d'inclure le fichier de définition de ces fonctions !

Remarque

✎ Lorsqu'on inclut un fichier *système* (i-e fourni dès l'installation), on l'indique entre chevrons : `#include <math.h>`. Le compilateur sait alors où il se trouve. Mais quand il s'agit d'un fichier *personnel* qui se trouve au même endroit que le fichier source que l'on développe, alors on l'indique entre guillemets : `#include "perso.h"`.

1.3 Encore des fonctions fournies (fichier `TP3e2.cpp`)

Dans un deuxième programme (fichier `TP3e2.cpp`), déclarez un tableau nommé *tab* de 50 entiers. Remplir *tab* par des entiers au hasard entre -100 et +100 (bornes comprises). Faire afficher la moyenne des 10 premières variables de *tab* en utilisant la fonction `MoyTab()` puis faire afficher les 10 premières variables de *tab* en utilisant `AfficheTab()`. Faire afficher en colonne 50 et ligne 25 la question « On recommence ? » et faire en sorte que le programme recommence tant que la réponse n'est pas le caractère `n`.

Remarque

✎ Vous pouvez aussi utiliser le fonction `getch()`. Voir l'exemple fourni : `getch.cpp`

2 Prototypage des fonctions et passage des arguments (fichier `TP3e3.cpp`)

Il est maintenant temps non plus d'*utiliser* des fonctions écrites par d'autres mais de développer vos *propres* fonctions. De plus, ce programme va permettre d'approfondir différents points.

2.1 Le prototypage

Nous avons déjà vu que le prototype d'une fonction correspondait à sa déclaration. En langage C, il faut déclarer localement tout ce qu'on utilise, les variables *et* les fonctions.

Mais au sujet du prototypage, ajoutons deux remarques importantes :

la portée du prototype : Prenons l'exemple d'un programme principal qui appelle une fonction `F()`, on *doit* la déclarer avant de l'utiliser. Il faut donc la prototyper *dans* le programme principal. Mais si la fonction `F()` appelle dans sa définition une fonction `G()`, où doit-on la déclarer ? Sûrement pas dans le `main()` qui ne l'utilise pas. On doit toujours, c'est un bon principe du génie logiciel, déclarer au plus près ce qu'on utilise, que ce soit des variables ou des fonctions. C'est le problème de la *visibilité* (ou de la *portée*) des objets utilisés. On parle de *déclarations locales* par opposition à *déclarations globales*.

le style du prototype : On demande que vous suiviez le modèle suivant, dans lequel la fonction est déclarée avec un prototype **muet et commenté**. En effet, le compilateur n'a pas besoin du nom des paramètres *formels* (utilisés localement dans la définition de la fonction) mais seulement de leur type. Par contre, vous avez besoin de savoir ce que représentent ces paramètres, et s'ils sont uniquement en entrée ou bien dans d'autres modes que l'on verra plus tard. Pour le moment indiquez systématiquement le mode `(entree)` dans vos prototypes, sauf pour les tableaux pour lesquels vous noterez `(entree/sortie)`.

Exemples :

```
// prototypes muets et commentes
void Fonction_1(char); // caractère 'O' ou 'N' (entrée)

float Fonction_2(double,  // l'abscisse (entrée)
             unsigned short, // le nombre de pas (entrée)
             bool);       // l'état du systeme (entrée)

int Fonction_3(int [], // tableau d'entier (entrée/sortie)
            int);  // taille du tableau (entrée)
```

Dans le programme `TP3e3.cpp`, vous allez déclarer et saisir une variable `x` de type `double`, puis vous afficherez le résultat des trois fonctions suivantes :

- `F()`, fonction qui reçoit un paramètre de type `double` et qui retourne 5 fois son paramètre ;
- `G()`, fonction qui reçoit un paramètre de type `double` et qui retourne son paramètre moins 2 ;
- `H()`, fonction qui reçoit un paramètre de type `double` et qui retourne 5 fois son paramètre moins 2. On demande, bien sûr, que la définition de `H()` fasse appel aux fonctions `F()` et `G()`, et que, de plus, on utilise la *composition* des fonctions (c'est le mécanisme par lequel une fonction appelle une autre fonction).

2.2 Le passage des arguments

Nous venons de voir qu'un programme non trivial est organisé hiérarchiquement : le programme principal appelle des fonctions de niveau inférieur pour sous-traiter des tâches spécialisées. Cette *communication* entre fonctions est réalisée par l'action d'*appel*. Quel que soit le langage utilisé, se pose le problème de la transmission des paramètres entre la fonction appelante et la définition de la fonction appelée. On distingue généralement trois modes.

2.2.1 Passage par valeur (argument en entrée)

Dans ce mode la fonction appelante, par exemple le `main()`, appelle une fonction avec un paramètre `p` de type `int` : `F(p);`, dont la définition modifie son paramètre. Ce qu'il faut comprendre, c'est que lors d'un passage par valeur, le **compilateur créé une copie du paramètre** pour la définition de la fonction. Celle-ci, donc, ne travaille pas directement sur le paramètre mais sur sa copie qui, de plus, sera détruite en fin de définition. Bref, **l'appel n'a pas changé la valeur du paramètre !**

2.2.2 Passage par référence (argument en entrée/sortie)

Dans ce mode, avec les mêmes notations que précédemment, on indique au compilateur de ne pas travailler sur une copie mais sur un *alias* du paramètre. Cela signifie que, bien que le paramètre de la définition puisse avoir un autre nom que lors de son appel, la fonction va directement travailler sur lui. Donc, quand on fait un appel par référence, **on change la valeur du paramètre !**.

2.2.3 Passage des tableaux (argument en entrée/sortie)

Le troisième mode sera étudié en détail lors des TP sur les pointeurs. C'est le passage par adresse. Le passage des tableaux en argument fait partie de ce type. Pour le moment, il est suffisant de savoir qu'il se traite de la façon indiquée par l'exemple suivant qui synthétise ce que l'on vient de voir :

```
{
   int a = 3, b;
   int tab[] = {-1, 0, 1}; // la dimension de tab vaut celle de la liste (3)
   // prototypes locaux
   int F_ParValeur(int);  // paramètre par valeur (entrée)
```

```
   int F_ParReference((int &); // paramètre par référence (entrée)
   int F_ParAdresse(int []); // un tableau d'int (entrée/sortie)

  // les appels se ressemblent :
  b = F_ParValeur(a);
  b = F_ParReference(a);
  // Notez bien l'appel des tableaux : pas de type ni de crochets !
  b = F_ParAdresse(tab);
}

//définitions :
int F_ParValeur(int x)
{
   x = ... // x ne change pas dans l'appelant
}

int F_ParReference(int &y)
{
   y = ... // y change dans l'appelant
}

int F_ParAdresse(int t[])
{
   t[1] = ... // t (donc tab) change dans l'appelant
}
```

2.3 Exercices d'application

1. On donne la fonction de prototype :

   `int ZeFonction(double, int);`

   qui retourne la partie entière du produit de ses deux arguments.

   (a) Combien vaut : `ZeFonction(5.4, 3)` et `ZeFonction(M_PI, 2)` ?

   (b) Que voit-on sur l'écran si un programme contient l'instruction :

   `ZeFonction(5.4, 3);`

   Quel est le problème ?

   (c) On veut voir sur l'écran la partie entière de 5×2.718. Quelle instruction faut-il écrire dans le programme ?

2. Une fonction `Mystere()` reçoit en entrée deux arguments de type `int` et retourne la somme de leur PPCM et de leur PGCD.

   (a) Quel est le prototype de la fonction `Mystere()` ?

   (b) Comment obtenir sur l'écran l'affichage :

   ```
   Mystere(9, 4) = 37
   Mystere(10, 5) = 15
   ```

   où les valeurs 37 et 15 sont calculées par l'ordinateur !

3. Une fonction `Truc()` reçoit deux arguments, le premier en entrée de type `int`, le second en sortie est un booléen, et elle retourne un `int`.
   Le booléen en sortie vaut `true` si le premier argument est pair et `false` sinon. La valeur de la fonction est l'exposant maximum dans 2^n tel que l'argument d'entrée soit soit divisible par 2^n.

   (a) Écrire le prototype de `Truc()`

   (b) `b` étant un booléen, que donne sur l'écran :
   `cout << Truc(19, b);`

   (c) `n` étant un `int`, que donne sur l'écran :
   ```
   n = Truc(18, b);
   cout << "n = " << n << " et b = " << b;
   ```

2.4 Dernier programme (fichier `TP3e4.cpp`)

Pour terminer ce TP, écrire le programme `TP3e4.cpp` qui effectue les actions suivantes :
- le programme principal déclare une variable x affectée à 5 et deux tableaux de 3 `int` appelés `tabPair` et `tabImpair` respectivement affectés par les trois premiers entiers pairs et impairs. Le `main()` déclare aussi trois prototypes : une fonction `ParValeur()` qui possède un paramètre `long` et ne retourne rien, une fonction `ParReference()` qui possède un paramètre référence sur `long` et ne retourne rien et une fonction `PourLesTableaux()` qui possède deux paramètres sur des tableaux de type `int` ;
- Appelez `ParValeur(x)`, puis affichez la valeur de `x` ;
- Appelez `ParReference(x)`, puis affichez la valeur de x. Que constatez-vous ?
- Appelez `PourLesTableaux()` avec nos deux tableaux, puis affichez leurs valeurs ;
- Enfin écrivez les définitions des fonctions qui accomplissent les actions suivantes :
  - `ParValeur()` additionne 2 à son argument[1] ;
  - `ParReference()` additionne 2 à son argument[2] ;
  - `PourLesTableaux()` affecte 55 à la première case du premier tableau et 66 à la troisième case du second tableau.

Ce qu'il faut retenir :

☞ une fonction peut posséder zéro, un ou plusieurs arguments ;

☞ ces arguments ne doivent pas être modifiés par la fonction ;

☞ une fonction retourne un résultat *et un seul* ;

☞ une fonction possède trois « aspects », suivant qu'on la déclare (c'est le *prototype*), qu'on la définit (c'est la *définition*) ou qu'on s'en sert (c'est l'*appel*) ;

☞ une fonction peut être déclarée *dans* une fonction, ce qui doit être fait quand elle est utilisée *localement* ;

☞ les noms des arguments *formels* d'une fonction (ceux utilisés dans la définition) n'ont rien à voir avec les noms des arguments *réels* (ceux utilisés lors de l'appel) ;

☞ les arguments réels peuvent être des constantes ou des variables ;

☞ bien faire la différence entre passage par valeur et par référence ;

☞ les tableaux sont toujours passés par adresse ;

☞ attention à ne pas confondre « nommer une fonction » c'est-à-dire lui choisir un nom et « appeler une fonction » c'est-à-dire s'en servir.

☞ lorsqu'une fonction `F()` utilise une fonction `G()` on dit que `F()` est la fonction *appelante* et `G()` la fonction *appelée* ;

☞ en C, toute fonction prototypée et définie peut être appelée et/ou appelante.

☞ comme le programme principal n'a pas de prototype, la fonction `main()` ne peut être qu'appelante ;

[1] Ici le compilateur doit émettre un *warning*, il se plaint que l'argument de la fonction soit affecté mais pas utilisé. Pouvez-vous le justifier ?
[2] Ici le compilateur n'émet pas de *warning*. Pouvez-vous le justifier ?

3 À préparer pour le prochain TP

Écrire le prototype C et l'algorithme de la fonction `IndDuMax()` du TP n° 4.

ENCORE DES FONCTIONS…

L'amour humain ne se distingue du rut stupide des animaux
que par deux fonctions divines : la caresse et le baiser.

Pierre LOUYS

Objectifs :
– bien écrire, utiliser et réutiliser des fonctions ;
– notion de récursivité.

Fichiers à produire : `TP4e1.cpp TP4e2.cpp TP4e3.cpp TP4e4.cpp`

---

1 Un algorithme de fonction

La fonction `MinTab()` calcule le minimum des n premières valeurs contenues dans un tableau t d'entiers de taille MAX.

Elle reçoit trois arguments : le tableau tab d'entiers dans lequel elle cherche le minimum, la valeur de MAX et le nombre n de variables réellement affectées dans ce tableau. Elle retourne la valeur minimale des n premières variables de t.

Voici son algorithme :

Algorithm 1 Fonction `MinTab()` d'un tableau d'entiers

```
DébutFonction MinTab(↔ tab : entier[N], → M : entier, → n : entier) : entier
    DébutDéclaration
        cpt, iMin, top : entier
    FinDéclaration

    Si (n > M)
        top ← M
    Sinon
        top ← n
    FinSi

    iMin ← 0

    Pour cpt Dans [1..top[ ParPasDe 1
        Si (tab[cpt] < tab[iMin])
            iMin ← cpt
        FinSi
    FinPour

    Retourner (tab[iMin])
FinFonction
```

Attention

☞ On demande d'écrire l'algorithme d'une fonction `IndDuMax()` recevant deux arguments : un tableau d'entiers et sa taille, et retournant l'indice de la valeur maximale de ce tableau.

2 Suites de Syracuse

2.1 Ce que c'est…

On appelle suite de Syracuse toute suite d'entiers naturels vérifiant :

$$u_{n+1} = \begin{cases} \dfrac{u_n}{2} & \text{si } u_n \text{ est pair} \\ 3\,u_n + 1 & \text{si } u_n \text{ est impair} \end{cases}$$

Par exemple, la suite de Syracuse partant de $u_0 = 11$ est :

$$11, 34, 17, 52, 26, 13, 40, 20, 10, 5, 16, 8, 4, 2, 1, 4, 2, 1, 4, 2, 1$$

En se bornant[1] à 1, on appelle cette suite finie d'entiers le **vol** de 11. On appelle **étape** un nombre quelconque de cette suite finie. Par exemple 17 est une étape du vol de 11. On remarque que la suite atteint une étape maximale, appelée **altitude maximale** du vol (52 pour le vol de 11).

Attention

☞ Dans tout ce qui suit les entiers seront de type **long**.

2.2 … et ce qu'il faut faire

Premier pas… (fichier : `TP4e1.cpp`)

Écrire en C une fonction `EtapeSuivante()` qui reçoit un entier et qui produit l'entier suivant dans la suite de Syracuse. Par exemple `EtapeSuivante(5)` doit produire 16 et `EtapeSuivante(16)` doit faire 8, etc.

Écrire un programme principal qui demande un entier initial supérieur à 1[2] à l'utilisateur puis qui, en utilisant la fonction `EtapeSuivante()`, calcule et affiche les termes de la suite de Syracuse jusqu'à ce qu'on obtienne 1.

… Pas suivant… (fichier : `TP4e2.cpp`)

Renommez le fichier précédent pour le compléter par une fonction `AltMaxi()` qui reçoit un entier $init$ et qui produise l'altitude maximale de la suite de Syracuse commençant par $u_0 = init$. Par exemple, en partant de $u_0 = 5$ la suite de Syracuse est : 16, 8, 4, 2, 1, 4, 2, 1. Donc `AltMaxi(5)` vaut 16. En partant de $u_0 = 7$, la suite de Syracuse est : 22, 11, 34, 17, 52, 26, 13, 40, 20, 10, 5, 16, 8, 4, 2, 1 et par conséquent `AltMaxi(7)` vaut 52.

Tester cette fonction `AltMaxi()` par quelques appels depuis le programme principal.

… Dernier pas. (fichier : `TP4e3.cpp`)

Écrire un dernier programme qui cherche parmi les entiers de 1 à N, où N est choisi par l'utilisateur, quel est celui qui produit la suite de Syracuse atteignant la plus grande altitude maximale (voir figure 7). Pour cela on pourra soit utiliser dans le programme principal les fonctions précédentes, soit développer une nouvelle fonction.

```
        Entrez un entier positif : 32

Pour les vols de 1 a 32, celui partant de 27
produit la plus grande altitude maximale : 9232
```

FIG. 7 – Plus grande *altitude maximale* de suites de Syracuse

[1] On a constaté depuis longtemps que pour tous les entiers de départ testés, la suite atteint après un nombre de termes plus ou moins grand, la période 4, 2, 1, 4, 2, 1… Hélas ! pour l'instant tous ceux qui ont essayé de prouver que c'est ainsi pour tout entier de départ ont échoué : c'est un « problème ouvert » (une *conjecture*).

[2] Ne pas oubler de faire une saisie filtée…

3 Notion de récursivité

3.1 Introduction

La notion de récursivité en programmation peut être définie en une phrase :

une fonction récursive peut s'appeler elle-même.

Une définition est récursive lorsqu'une partie du tout fait appel au tout. Par exemple, trier un tableau de N éléments c'est extraire le minimum puis trier le tableau restant à $N-1$ éléments.

Bien que cette méthode soit souvent plus difficile à comprendre et à coder que la méthode classique dite *itérative*, elle est dans certains cas l'application la plus directe de sa définition mathématique.

Voici un exemple de définition par récurrence de la fonction factorielle :

$$n! \begin{cases} 1 & \text{si} \quad n = 0 \\ n\,(n-1)! & \text{si} \quad n \geqslant 1 \end{cases}$$

Une définition peut éventuellement faire appel plusieurs fois à elle-même. Ainsi, les coefficients C_n^p du triangle de PASCAL peuvent être définis par une double récurrence :

$$C_n^p \begin{cases} C_n^0 = 1 & \text{pour tout} \quad n \geqslant 0 \\ C_n^p = 0 & \text{pour} \quad p > n \geqslant 0 \\ C_n^p = C_{n-1}^{p-1} + C_{n-1}^p & \text{pour} \quad n \geqslant m > 0 \end{cases}$$

3.2 Dérouler un programme récursif (exemple avec $n!$)

Nous allons détailler le problème de la factorielle. Tout d'abord, remarquons que sa définition algorithmique est très exactement calquée sur sa définition mathématique :

```
DébutFonction FACTORIELLE(→ n : entier) : entier
    Si (n == 0)
        Retourner 1
    Sinon
        Retourner (n * factorielle(n − 1))
    FinSi
FinFonction
```

Dans cette définition, la valeur de $n!$ n'est pas connue tant que l'on n'a pas atteint la condition terminale (ici $n == 0$). Le programme empile les appels récursifs jusqu'à atteindre la condition terminale puis dépile les valeurs. Ce mécanisme est illustré par les figures 8 et 9.

Pour dérouler un programme récursif, il suffit de dérouler tous les appels dans un tableau comme ci-dessous, jusqu'à tomber sur la condition terminale sans appel (par exemple ici quand $n == 0$). Ensuite on fait remonter à partir de la ligne du bas (sans appel) les valeurs de retour trouvées (3[e]colonne) jusqu'en haut du tableau.

Fonction(n)	Appels	Valeur de retour
factorielle(n)	**n*factorielle(n-1)**	**n*factorielle(n-1) = 24**
factorielle(4)	4*factorielle(3) ↓	24
factorielle(3)	3*factorielle(2) ↓	6 ↑
factorielle(2)	2*factorielle(1) ↓	2 ↑
factorielle(1)	1*factorielle(0) ↓	1 ↑
factorielle(0)	sans appel	1 ↑

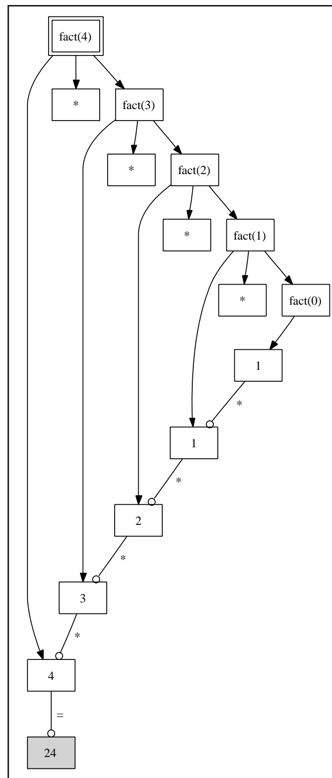


FIG. 8 – Empilage/dépilage de $4!$

3.3 Ce qu'il faut faire (fichier `TP4e4.cpp`)

On définit la suite de FIBONACCI de la façon suivante :[1]

$$\begin{cases} fib_0 = 0 \\ fib_1 = 1 \\ fib_{n+1} = fib_n + fib_{n-1} \end{cases}$$

On demande d'effectuer les actions suivantes :
- écrire l'algorithme de la suite de FIBONACCI de manière récursive (Attention à choisir des tests de terminaison adéquates...) ;
- Dérouler le programme comme expliqué dans le paragraphe précédent avec 5 comme entrée. Remplir le tableau suivant ;
- Coder le fichier `TP4e4.cpp` et vérifier les résultats obtenus.

Fonction(n)	Appels	Valeur de retour
fib(n)	**fib(n-1)+fib(n-2)**	**fib(n-1)+fib(n-2) = ?**
fib(5)	fib(4) + fib(3) ↓	
fib(4)	fib(3) + fib(2) ↓	↑
fib(3)	fib(2) + fib(1) ↓	↑
fib(2)	fib(1) + fib(0 ↓	↑
fib(1)	sans appel	↑
fib(0)	sans appel	↑

[1] *Attention*, c'est une définition doublement récursive...

```
Entrez un entier [0 .. 13] : 9

Observez l'empilage des appels puis le depilage des valeurs :

 fact(9) appelle : 9*fact(8)
   fact(8) appelle : 8*fact(7)
     fact(7) appelle : 7*fact(6)
       fact(6) appelle : 6*fact(5)
         fact(5) appelle : 5*fact(4)
           fact(4) appelle : 4*fact(3)
             fact(3) appelle : 3*fact(2)
               fact(2) appelle : 2*fact(1)
                 fact(1) appelle : 1*fact(0)
1  2  6  24  120  720  5040  40320  362880

resultat : 9! = 362880
```

FIG. 9 – Visualisation des appels récursifs de 9!

Ce qu'il faut retenir :

☞ Lorsque l'on utilise des fonctions, bien respecter l'ordre suivant :

1. **déclaration :** écriture du *prototype* muet commenté de la fonction
2. **appel :** c'est l'utilisation de la fonction
3. **définition :** écriture du code de la fonction

☞ on peut *appeler* des fonctions dans des fonctions ;

☞ on ne peut pas *définir* des fonctions dans des fonctions.

4 À préparer pour le prochain TP

Écrire l'algorithme de la fonction `Dicho()` du TP n° 5.