Bases pragmatiques de programmation

« Les bons mathématiciens/programmeurs sont des fainéants intelligents »

D.Grenier

Centre de Recherche en Acquisition et Traitement de l'Image pour la Santé.

CNRS UMR 5220 – INSERM U1294 – Université Lyon 1 – INSA Lyon - Université Jean Monnet Saint-Etienne.

Notions sur les langages informatiques

La langue parlée par votre ordinateur est le binaire son alphabet n'a que deux lettres 0 et 1, le moindre mot prend donc un certain temps à être écrit par un humain :
PolytechLyon=01010000011011110110110001111001011101000110010101100011011010000100110001111001011011101101110.

Le binaire est le langage naturel de l'ordinateur, car il est très facile et rapide de faire commuter des transistors à une vitesse de plusieurs GHz de l'état passant (0) à l'état bloqué (1).
Par contre ce langage est très inadapté pour **interagir** avec un humain.

L'intérêt d'un langage informatique est de servir d'intermédiaire entre la machine et l'humain : l'humain dispose alors d'un langage suffisamment proche du sien pour décrire ce qu'il veut que l'ordinateur fasse (par exemple la cuisine), et le compilateur va faire la traduction de cette description en binaire pour que l'ordinateur sache réaliser ce que l'humain lui a demandé.
La force des microprocesseurs de nos ordinateurs est leur capacité à pouvoir extrêmement facilement se reprogrammer pour effectuer des tâches très différentes : ils peuvent servir de traitement de texte, à visualiser des pages web, faire de la musique, dessiner…
(Par opposition aux microcontrôleurs que l'on destine à faire réaliser une tache précise : contrôler un ascenseur, une porte de garage, une maison…).

Ainsi si on veut que la machine soit spécialisée dans la réalisation d'une tache ou d'un groupe de taches et que l'on connaît exactement ce que ces tâches nécessitent en entrée et ce qu'elles envoient en sortie (si on peut définir tous les cas possibles de réalisation de la tache) alors la tâche serait sans doute plus appropriée à des microcontrôleurs ou FPGA dans lesquels la tâche peut être « gravée ».

Si les tâches à réaliser sont des variantes d'une tâche fondamentale dont les conditions initiales vont dépendre de l'utilisateur ou d'un contexte qui peut varier de manière très importante alors on a intérêt à utiliser un microprocesseur, car il pourra très facilement adapter les tâches qu'il doit réaliser en fonction de leur contexte, il lui suffit de pouvoir charger dans ses microprocesseurs la description générique unique de ce qu'il a à faire (le programme) et qu'il puisse adapter cette

description en fonction d'un contexte précis décrit dans un fichier de configuration (les données d'entrées)

Un fichier de type exécutable (qu'on appelle aussi programme), sert à décrire à l'ordinateur le travail qu'il doit faire, ce fichier contient des instructions exécutables écrites dans le langage des microprocesseurs (le binaire)
Un fichier de type donnés contient de quoi interagir avec un programme :

- Soit les informations que doit utiliser le programme pour assurer sa tâche.
- Soit ce qu'aura produit le programme (résultats)

Ex : Le programme (binaire) word.exe utilise des fichiers de configuration pour se spécialiser (fichiers de type donnés) et il produit des fichiers avec l'extension .docx qui sont eux aussi des fichiers de type donnés, décrivant le texte qui a été tapé
le programme binaire matlab.exe utilise des fichiers de données entrants monTP.m et sauve les résultats ensuite sous forme de fichiers de données (sortantes) par exemple resultat.bin

Qu'est-ce qu'un compilateur ? Pourquoi compiler un programme ?

Tâche de traduction

Pour qu'une tache puisse être bien exécutée par l'ordinateur il faut :
– Que l'humain puisse bien décrire ce qu'il veut faire en utilisant le moins de mots possibles (langage adapté à une utilisation donnée)
— Qu'il n'y ait pas d'ambiguïté dans la description des tâches (rajouter du sel ! 1g ou 1kg ?)
— Qu'il n'y ait pas de fautes d'orthographe ou de grammaire (rajouter du citron, rajouter du cifron)

La **chaîne de compilation** est un ensemble de programme qui vont assurer la traduction des instructions du langage humain en langage binaire, elle permet d'indiquer à l'humain s'il a écrit des instructions qui sont ambigües ou avec des fautes qui empêcheraient l'ordinateur de pouvoir réaliser la tache qui lui est demandé.
S'il n'y a aucune erreur ou ambiguïté dans la description des tâches écrites dans le programme, le compilateur va ensuite transformer ce **texte** écrit par l'humain en un **programme exécutable** par l'ordinateur.

Tâche d'accélération

une commande (un programme)
$ affiche 312 422
met 1 µs pour allumer le pixel d'abscisse 312 et d'ordonnée 422 de l'écran, ça peut sembler rapide mais si votre écran fait 1920 × 1080 (soit à peu près 2M pixels) il faut près de 2 seconds pour accéder à tous les pixels de votre écran (*Vous imaginez un jeu vidéo dont les images changent seulement toutes les 2 secondes !!!!!)*
Pourquoi ce processus est il si lent ?
Rien n'est réutilisé entre chaque lancement :

*→ la commande est **recherchée** sur le disque dur*
→ transportée en mémoire vive,
→ exécutée par le processeur, quand il a un créneau de libre
→ puis proprement déchargée du processeur
*→ et enfin **libérée** de la mémoire vive.*

Même si la partie exécution par le processeur peut être extrêmement rapide, un temps énorme est utilisé pour la préparation de l'exécution et pour le nettoyage après exécution, car l'ordinateur ne sait pas ce qu'on va lui demander de faire ensuite :
Quand vous faites la cuisine, est ce que vous nettoyez la cuisine après chaque utilisation de chaque ustensile ?
Eh bien, par défaut, votre ordi, lui, il le fait !!!

- Si vous utilisez un langage **interprété (shell, bash, perl, python…),** l'ordinateur considérera que chaque ligne que vous écrivez est une recette de cuisine différente et il fera la vaisselle, rangera la cuisine et passera l'aspirateur entre chaque recette (ligne de code)
- Si vous utilisez un langage **compilé,** l'ordinateur comprendra qu'il doit optimiser le temps qu'il passe dans chaque recette et si possible réutiliser les mêmes ustensiles ingrédients sans les ranger. Il ne fera vraiment le ménage qu'à la fin du programme.

Si le programme que vous écrivez doit être **compilé**, l'ordinateur doit être capable d'avoir une **stratégie d'exécution** :
Si vous avez besoin de faire une pâte brisée pour la quiche de l'entrée et une autre pour la tarte à la fraise, vous avez tout intérêt à faire votre pâte en une fois et la séparer en deux.
Vous pouvez aussi avoir intérêt à faire cuire votre tarte et votre quiche en même temps.

Si l'ordinateur est capable de comprendre toutes les tâches qu'il aura à faire, on peut lui demander de minimiser le nombre d'actions qu'il aura à faire pour cuisiner le repas complet.

Le programme qui permet à l'ordinateur de comprendre le programme écrit par l'humain et de l'optimiser pour l'ordinateur s'appelle un **compilateur**.

Il existe des langages pour faire la cuisine, d'autres pour faire la lessive ou sortir le chien ce qui en langage d'informaticien veut dire qu'il existe plus de 300 langages/compilateurs plus spécialement dédiés à certaines taches : Fortran, C, C++, C#, LISP, Ruby, Python, PERL :
https://en.wikipedia.org/wiki/List_of_programming_languages
Certains langages sont dédiés au calcul scientifique numérique, d'autres au calcul formel, d'autre à la programmation d'intelligence artificielles, d'autres à la description géométrique, au stockage et recherche d'information, au web, à la communication entre objets…

Si nous avons choisi de vous enseigner le langage C++ c'est parce qu'il offre une grande flexibilité (il permet de créer des programmes d'utilisation très variés) et il utilise une philosophie de programmation (objet) que l'on retrouve dans un grand nombre d'autres langages.

Les librairies

Chaque langage possède un certain nombre d'instructions qui permettent de décrire déjà un grand nombre de situations. Cependant, les personnes qui ont écrit ces langages n'ont pas pu penser à tout ce que voudraient faire les programmeurs.
Pour permettre de résoudre ce problème, tous les langages actuels font appel à la notion de librairie.

Une librairie peut être vue comme un moyen d'ajouter de nouveaux mots au langage informatique. Ces mots sont construits à partir du langage de base et peuvent décrire des situations très complexes.
Pour garder l'analogie culinaire
Au lieu d'écrire la recette / le programme :
→ prendre 1 kg de farine
→ prendre 500 g de sucre
…(10 lignes d'instructions diverse)
→ mélanger ceci et cela,
….(25 lignes d'instructions diverses)
→ mettre des fraises
→ cuire le tout

Si on dispose de la bonne librairie, on peut écrire :

→ Lire la librairie des pâtes à tarte
→ faire une pâte brisée
→ mettre des fraises
→ cuire le tout

En fait, dans bien des cas que vous aurez à traiter, les librairies que vous utiliserez contiendront bien plus de lignes de codes que votre programme par lui-même.

Ainsi le langage C/C++ connaît par défaut l'addition, la soustraction, la multiplication, la division, mais il ne connaît même pas sin, cos, tan…
Heureusement, toutes les variantes de C/C++ sont fournies avec un très grand nombre de librairies. Ces librairies vont étendre le langage pour nous permettre de réaliser facilement certaines descriptions de taches complexes.
Il existe ainsi des librairies pour afficher des caractères à l'écran, des librairies mathématiques, des librairies graphiques, pour le son, pour la communication réseau, pour lire des fichiers…
à ces librairies standards se rajoute des milliers de librairies encore plus spécialisées, écrites par des programmeurs (plus ou moins compétents) qui les mettent en libre service sur internet.
Posez-vous toujours la question du choix d'utiliser telle ou telle librairie dans votre code (est ce que la librairie que je vais utiliser est bien écrite, stable, portable, en développement actif, populaire… ?)

Une bonne partie de votre travail de programmation consistera à utiliser des librairies écrites par d'autres, qui vous permettront de taper le moins de lignes de code possible dans votre programme.

S'il n'existe pas de librairie vous aidant à décrire ce que vous voulez faire, plutôt que d'écrire un gros programme, il est souvent beaucoup plus judicieux d'écrire vous-même une (grosse) librairie et de l'incorporer ensuite dans le (petit) programme que vous aurez écrit. Cette approche permet de favoriser la réutilisation des codes.

Le bien coder

On pense souvent par erreur que coder un programme consiste à écrire un monologue entre une personne (vous) et un ordinateur.
C'est faux.
Coder un programme consiste à dialoguer avec un ordinateur ET à monologuer avec toutes les personnes qui auront à utiliser/améliorer/modifier votre code source.

Un bon code est un code que votre ordinateur comprendra et que vous-même ou d'autres personnes pourront réutiliser et modifier facilement même en le reprenant 10 ans après sa création :

Que fait ce programme ?:

```
PROGRAM TCVR
    IMPLICIT INTEGER(A-Z)
    PARAMETER(L=14)
    DIMENSION A(L),S(L),U(4*L-2)
    DO 10 I=1,L
    A(I)=I
 10 CONTINUE
    DO 20 I=1,4*L-2
    U(I)=0
 20 CONTINUE
    DO 110 N=1,L
    M=0
    I=1
    R=2*N-1
    GO TO 40
 30 S(I)=J
    U(P)=1
    U(Q+R)=1
    I=I+1
 40 IF(I.GT.N) GO TO 80
    J=I
 50 Z=A(I)
    Y=A(J)
    P=I-Y+N
    Q=I+Y-1
    A(I)=Y
    A(J)=Z
    IF((U(P).EQ.0).AND.(U(Q+R).EQ.0)) GO TO 30
 60 J=J+1
    IF(J.LE.N) GO TO 50
 70 J=J-1
    IF(J.EQ.I) GO TO 90
    Z=A(I)
    A(I)=A(J)
    A(J)=Z
    GO TO 70
 80 M=M+1
 90 I=I-1
    IF(I.EQ.0) GO TO 100
    P=I-A(I)+N
    Q=I+A(I)-1
    J=S(I)
    U(P)=0
    U(Q+R)=0
    GO TO 60
100 PRINT *,N,M
110 CONTINUE
    END
```

Donnez les grosses faiblesses de ce programme :

- Pas de commentaires.
- Noms de variables cryptiques.
- C'est des L minuscules ou des i majuscules ?
- …

Il existe des standards de codage (voir par exemple hungarian notation sur wikipedia) qui ont des fans et des détracteurs qui possèdent chacun autant d'éléments.
Ce qu'il faut retenir de ça est que, aujourd'hui, les environnements de programmation s'occupent de simplifier une très grande partie de votre travail et vous aident énormément à écrire et comprendre votre code ; certaines coutumes de programmation deviennent obsolètes.
Les standards de codage varient avec les employeurs, mais il y a des règles de base génériques qui sont des règles de bon sens :

- **Documentez beaucoup votre code**, indiquez avec précision à quoi sert chaque partie, variable, classe, méthode, fichier.

- **Segmentez votre code** : un groupe de fichier par classe
  - maClasse.h (h pour header, contient l'interfaçage de la classe avec l'extérieur)
  - maClasse.cpp (.cpp (pour c plus plus) ou parfois…. .cxx contient l'implémentation de la classe)
  - maClasseUnitTest.cpp (contient les tests permettant de vérifier que la classe fonctionne bien)

- **Donnez aux variables, classes, méthodes, un nom explicite** (pas trop long non plus) :
  - A (mauvais)
  - SignalVecCplx (bon)
  - Signal_Avant_Filtrage_Vecteur_Complexe (trop long, une ligne de programme risque d'être plus longue que votre écran n'est large)

- En général on **fait commencer les noms de classes par une majuscule et les instances par une minuscule** :
  - MaClasse
  - instanceDeMaClasse

    ex : Recette quiche = new Recette() ;

- Les valeurs constantes sont écrites en majuscule :
  const double PI = 3.1415927 ;

- **Limiter l'utilisation de scalaires** :
  if (r < 0.001) ….(mauvais)

  const double TOLERANCE = 0.001 ;
  (ou bien)
  #define TOLERANCE 0.001
  …….
  if (residu < TOLERANCE) …….(bon)

Détecter/gérer les erreurs

Il faut bien faire la différence entre deux cas :

- les **erreurs de codage** qui résultent d'étourderies, de fautes conceptuelles, de manque de précision… Il s'agit là de « fautes » que l'on doit corriger et faire disparaître du programme source.
- les **erreurs dues à un code qui ne tient pas compte de tous les cas de figure possibles** que l'utilisateur très inventif peut mettre en œuvre : « il n'était écrit nul par que je n'avais pas le droit de sécher mon chien dans le micro-onde !! ». Il s'agit dans ce second cas d'autonomiser le programme vis-à-vis des aléas extérieurs.

Nous disposons de différents moyens de gestions de ces deux situations. Les debuggeurs, les gestionnaires d'erreur, les unit tests.

Les « débuggers »

Ne confondez jamais un programme qui compile avec un programme qui marche.
Que votre programme ait été compilé veut juste dire que les règles d'orthographe et de grammaire ont bien été respectées et que votre programme a été traduit en une suite de taches réalisables par l'ordinateur.
Cela ne veut absolument pas dire que la suite de tâches réalisées par l'ordinateur vous donnera le résultat escompté.
Un programme de débuggage permet de suivre l'état de l'ordinateur (valeurs des variables présentes en mémoire) pour chacune des lignes de codes **traversées** (si le programme ne passe jamais par certaines lignes, leur fonctionnement ne pourra pas être analysé).

Remarque : Pour qu'un programme puisse être analysé il faut qu'il ait été compilé avec l'option de débuggage (pour que le programme de débuggage sache quel code binaire est exécuté dans chaque ligne).
Si votre code utilise des librairies extérieures qui n'ont pas été compilées avec cette option, vous ne pourrez pas suivre ce qui se passe dans les fonctions utilisant cette bibliothèque.

Le débuggage sert surtout à détecter les grosses bourdes de programmation faites dans les toutes premières versions du code et les petites erreurs dans du code aboutit quand on sait globalement ou chercher.

Gestion des erreurs critiques

Très souvent le programme que l'on écrit comporte des lignes de codes qui peuvent faire planter le programme, le plus courant de ces cas est de gérer une division par zéro.
Le moyen le plus simple de gérer ça est de créer un test qui vérifie si le dénominateur est nul et le remplace alors par le plus petit nombre que gère la machine. Le problème de cette approche est par exemple qu'elle fait effectuer un test pour chaque valeur utilisée en dénominateur, ce qui rallonge l'exécution du programme .

Il existe dans les langages actuels un système de gestion d'erreur qui permet au programme de gérer les erreurs qui peuvent être créées lors de son propre fonctionnement. Au lieu de déclencher un arrêt total du programme en cas d'erreur, cela permet de définir comment le programme va réagir s'il rencontre des erreurs qui devraient le faire planter.

Cela permet par exemple à l'ordinateur de ne déclencher des opérations de tests (éventuellement coûteuses en temps) que a posteriori sur les valeurs à traiter qui ont posé problème.
Exemple :
```
try{
….partie de code dangereuse….
}
catch (Exception e){
….code à exécuter si la partie dangereuse a planté
}
```

Gestion des erreurs non critiques

les commandes de try/catch permettent de gérer les causes de plantage pur et simple de vos programmes, pour des erreurs non critiques il est souvent intéressant de faire en sorte que les fonctions ou méthodes renvoient une valeur booléenne ou autre qui nous indique si la fonction/méthode a bien fonctionné :
valeur_resultat = maFonction (valeur_d_entree) ; // (bof !)

if (! maFonction(valeur_d_entree, valeur_resultat)) affiche («maFonction a planté ») ; // mieux !
Ou
if (maFonction (ValEntree,ValResu) == 0) affiche («maFonction a marché») ;
if (maFonction (ValEntree,ValResu)== -1) affiche («maFonction a planté ») ;
if (maFonction (ValEntree,ValResu) == -2) affiche («maFonction n'a pas pu créer …. ») ;
if (maFonction (ValEntree,ValResu)== -3) affiche («maFonction n'a pas assez de ….») ;

Les « unit tests »

Les unit tests sont faits pour répondre à la question : Dans la mesure ou j'ai écrit un programme qui doit être assez polymorphe est ce que certaines conditions initiales peuvent faire planter le programme ?
Les unit tests sont un ensemble de programme qui vérifient certaines parties élémentaires de votre code (unit) pour tester que tout se passe bien.

Remarque : Dans le cadre de projets de TP courts, il est souvent difficile d'écrire des tests unitaires qui ne soient pas triviaux et qui aient une réelle utilité.

Pourquoi utiliser des unit tests quand on est déjà un pro du débugger ?

- Le débuggeur permet de suivre un code pas à pas : une ligne peut être étudiée par le programmeur chaque seconde, minute, heure ?
- Les unit tests permettent de faire un crash tests d'un bout du programme en le faisant tourner pour éventuellement des milliards de conditions initiales différentes.
- Écrire des unit tests vous aide à prendre du recul sur votre code :
  - Quand on écrit un code on est concentré sur comment le faire marcher (J'ai réussi à le faire marcher dans mon contexte).
  - Quand on écrit les unit tests on est concentré sur comment l'empêcher de bien marcher (Je voudrais qu'il marche dans tous les contextes que je pourrai trouver).

L'utilité et la puissance des unit tests devient plus simple à comprendre quand on parle de code qui sera amené à évoluer au cours du temps et/ou qui sera amené à fonctionner sur des ordinateurs très différents :

Pour un jeu vidéo par exemple, un unit test important serait de vérifier si la carte graphique peut charger suffisamment rapidement un certain volume de données : test ok on peut installer le jeu, test pas ok on ne peut pas ou alors la résolution de l'image devra être diminuée.

Pour un programme de recalage d'image qui utilise une librairie de minimisation qu'on a l'habitude d'utiliser, une série de unit tests charge différentes images et teste la qualité du recalage. Si quelques années plus tard on décide de changer la librairie de minimisation ou même juste sa version, relancer les unit tests permet de savoir très rapidement si la nouvelle librairie passe les spécifications que l'on avait définies pour le programme initial.

Quand vous écrivez un programme que vous espérez rendre populaire, pensez à passer du temps à définir les tests génériques qui permettront de tester :

- Si votre programme fonctionnera bien sur toutes les plate-formes sur lesquelles il sera compilé.
- Si une modification d'une partie de votre code améliore ou dégrade votre œuvre initiale.