Université de Liège

INFO0946

Introduction à la Programmation

Syllabus d'Exercices

Benoit Donnet, Géraldine Brieven, Simon Liénardy,
Thomas Leuther, David Lupien St-Pierre, Firas Safadi,
Tasnim Safadi

30 octobre 2023

Table des matières

Table des figures

Liste des tableaux

Introduction

Ce syllabus se veut être un recueil d'exercices permettant à l'étudiant de mettre en pratique les aspects théoriques et techniques vus lors des séances théoriques du cours INFO0946 (Introduction à la Programmation).

L'objectif principal du cours INFO0946 est l'acquisition des notions fondamentales de l'informatique et, en particulier, ceux liés à la programmation. Plus précisement, le cours s'articule autour des notions de syntaxe et sémantique du langage C et, surtout, de la méthodologie de la programmation. Nous étudierons donc successivement les notions suivantes :

- Variable, bloc, instruction simple (Chap. 1).
- Les structures de contrôle, i.e., conditions et boucles (Chap. 2).
- La mise en place d'une solution à un problème en suivant une méthodologie en quatre étapes : définition du problème, analyse du problème (approche systémique et architecture du code), construction du code autour de l'Invariant Graphique et tests [1] (Chap. 3).
- La mise en place de structures de données de base, i.e., tableaux, chaînes de caractères, enregistrements, énumérations, et fichiers (Chap. 4).
- La découpe du programme en modules, i.e., fonctions et procédures, permettant de structurer et réutiliser le code ainsi que la documentation de ces modules, (i.e., spécifications (Chap. 5).
- Les notions de mémoire, pointeurs, et passage de paramètres (Chap. 6).
- L'allocation dynamique de structures de données (Chap. 7).

Les niveaux d'apprentissage attendus chez les étudiants suivent la classification de Bloom [2] :

1. *Connaissance* : mémorisation et restitution d'informations dans les mêmes termes. Les notions théoriques vues dans le cadre du cours ne sont pas très nombreuses. Il est dès demandé de les maîtriser au mieux.
2. *Compréhension* : restitution du sens de l'information dans d'autres termes.
3. *Application* : utilisation de la méthodologie de développement et des algorithmes bien connus pour résoudre un problème.
4. *Analyse* : utilisation de la méthodologie de développement pour identifier les parties constituantes d'un tout.
5. *Synthèse* : utilisation de la méthodologie de développement pour exprimer comment les différentes parties (cfr. Analyse) doivent être combinées pour former un tout.
6. *Evaluation* : formulation de jugements qualitatifs ou quantitatifs sur la solution proposée.

Afin de mieux comprendre l'organisation pédagogique du cours, nous allons emprunter une métaphore musicale inspirée de Henri-Pierre Charles [3] :

1. La partie "tests" sera développée au Q2, dans le cadre du cours INFO0030.
2. B. S. Bloom, M. D. Englehart, E. J. Furst, W. H. Hill, D. R. Krathwohl. *The Taxonomy of educational objectives, handbook I : The Cognitive domain.* 1956. New York : David McKay Co., Inc.
3. H.-P. Charles. *Initiation à l'Informatique.* Ed. Eyrolles. 2000.

Pédagogie par Objectifs : le solfège est l'étude des principes élémentaires de la musique et de sa notation. Le musicien "fait ses gammes" et chaque exercice a un objectif précis pour évaluer l'apprentissage du "langage musical". Il en va de même pour l'informaticien débutant confronté à l'apprentissage d'un langage de programmation (le C pour nous).

Pédagogie par l'Exemple : l'apprentissage des grands classiques permet au musicien de s'approprier les bases du solfège en les appliquant à ces partitions connues et en les (re)jouant lui-même. L'informaticien débutant, en (re)codant lui-même les algorithmes bien connus (et en refaisant à la maison les problème résolus ensemble en séance théorique) se constituera ainsi une base de réflexes de programmation en "imitant" ces algorithmes.

Pédagogie de l'Erreur : les bogues (ou *bugs* en anglais) sont à l'informaticien ce que les fausses notes sont aux musiciens : des erreurs. Ces erreurs sont nécessaires dans l'acquisition de la connaissance. Un élève a progressé si, après s'être trompé, il peut reconnaître qu'il s'est trompé, dire où et pourquoi il s'est trompé, et comment il recommencerait sans produire les mêmes erreurs.

Pédagogie par Problèmes : connaissant ses "classiques" et les bases du solfège, le musicien devenu plus autonome peut envisager sereinement la création de ses propres morceaux. Le développement d'un projet informatique ambitieux sera "mis en musique" au cours du Q2 dans le cadre du cours INFO0030.

Dans le cours INFO0946, nous adopterons ces différentes stratégies pédagogiques sans oublier qu'en informatique, on apprend toujours mieux en faisant par soi-même.

A travers cette pédagogie, le cours cherche à développer trois "qualités" comportementales chez l'informaticien débutant : la rigueur, la persévérance, et l'autonomie.

Rigueur : un ordinateur est une machine qui exécute vite et bien les instructions qu'on lui a données. Mais cette machine ne sait pas interpréter autre chose : même mineure, une erreur provoque le dysfonctionnement de la machine. Par exemple, pour un `;` oublié dans un programme C, le code source n'est pas compilable et le compilateur nous avertit par ce type de message :

```
1    fichier.c : In function 'main' :
2    fichier.c :7 : error : syntax error before "printf"
```

Même si un humain peut transiger sur le `;`, la machine ne le peut pas : l'ordinateur ne se contente pas de "l'à peu près". Le respect des consignes, la précision et l'exactitude sont donc de rigueur en informatique !

Persévérance : face à l'intransigeance de la machine, le débutant est confronté à ses nombreuses erreurs (les siennes, pas celles de la machine ! !) et sa tendance naturelle est de passer à autre chose. Mais le papillonnage (ou zapping) est une très mauvaise stratégie en informatique : pour s'exécuter correctement, un programme doit être finalisé. L'informatique nécessite donc "d'aller au bout des choses".

Autonomie : programmer soi-même des algorithmes qu'on a définis est sans doute le meilleur moyen pour mieux assimiler les principales structures algorithmiques et pour mieux comprendre ses erreurs en se confrontant à l'intransigeante impartialité de l'ordinateur, véritable "juge de paix" des informaticiens. Par ailleurs, l'évolution continuelle et soutenue des langages de programmation et des ordinateurs nécessitent de se tenir à jour régulièrement et seule l'autoformation systématique permet de "ne pas perdre pied".

Pratique personnelle et autoformation constituent ainsi deux piliers de l'automatisation nécessaire de l'apprenti informaticien. C'est dans cette optique que ce syllabus a été rédigé.

La partie "pratique" supervisée (i.e., lors de séances) du cours INFO0946 est organisée de deux manières :

1. *Répétition.* Les séances de répétition sont organisées, pour chaque étudiant, de manière hebdomadaire. Chaque étudiant disposera de 15 séances de répétition durant le quadrimestre. Les séances de répétition sont encadrées par une Assistante, plusieurs Elèves-Moniteurs et le Professeur. Les séances de répétition ont lieu juste après le cours théorique. L'objectif de ces séances est de résoudre, sur papier, divers problèmes faisant directement référence à la matière vue au cours. Chacune des séances de répétition commence par un rappel théorique.
2. *Collaborative Design & Build* (CDB).[4] CDB est une activité collaborative (organisée sous la forme d'un jeu de rôle) vous permettant de sentir l'importance de chacune des étapes de la méthodologie de programmation. Trois séances seront organisées au cours du quadrimestre, à chaque fois à des moments clés.

Typiquement, lors de la séance, un exercice sera résolu complètement par l'Assistante. Les autres exercices seront à réaliser, si possible seul, en séance. C'est probablement le bon moment pour poser des questions (l'équipe pédagogique circule dans les rangs pendant la séance pour répondre aux questions ponctuelles) sur des points incompris de la matière. Il ne faut surtout pas attendre que le retard s'accumule, sous peine d'être noyé par la matière et de ne pas pouvoir appréhender la suite du cours et du cursus.

Le présent document contient bien plus d'exercices que ce qui pourra être fait, soit pendant les séances de répétition. L'idée alors est de permettre à l'étudiant de s'exercer par lui-même. Ce travail solitaire est vivement conseillé en vue de réussir l'examen. Les étudiants peuvent toujours contacter l'équipe pédagogique (Professeur ou Assistante) pour discuter des exercices et de leurs solutions. Un calendrier des disponibilités est donné en début d'année académique.

Un correctif de la plupart des exercices est donné dès le début de l'année sur eCampus. Attention, il s'agit là d'un exemple de code correct pour les exercices. En aucun cas, ce code ne correspond à la seule et unique solution. N'hésitez donc pas à discuter de votre propre solution avec l'équipe pédagogique.

Attention, ne soyez pas attentiste vis-à-vis de ces correctifs. Faites avant tout des exercices par vous-même sans regarder les solutions ! A l'examen, vous serez, seul, face à des problèmes inédits. Si vous n'avez pas suffisamment pratiqué (seul) pendant le quadrimestre, le résultat de l'examen risque d'être catastrophique.

En outre, toute une série d'exercices, à faire à domicile, vous sont proposés sur la plateforme CAFÉ[5] :

- Challenges.[6]. Il s'agit de (petits) exercices de programmation à faire à la maison. Il y aura en tout 6 Challenges durant le quadrimestre, chacun d'entre eux se focalisant sur un aspect important du cours. L'énoncé d'un Challenge apparait le mercredi à 18h sur eCampus et sur CAFÉ. Vous avez jusqu'au vendredi 20h pour résoudre le problème. C'est la plateforme CAFÉ qui se charge de la correction automatique de votre production, ainsi que la génération du feedback correspondant. Entre le mercredi et le vendredi, vous avez trois essais pour résoudre le problème. Seul le dernier essai compte (l'ensemble des Challenges vaut pour 10% de la note finale). Une fois la deadline passée, il vous est possible de refaire le Challenge autant de fois que vous le désirez (il n'y a plus de note).
- GAMECODES.[7]. Un GAMECODE se présente comme un exercice interactif implémenté

4. G. Brieven, L. Leduc, B. Donnet. *Collaborative Design and Build Activity in a CS1 Course : A Practical Experience Report.* In Proc. 8th International Conference on Higher Education Advances (HEAd). June 2022.
5. G. Brieven, L. Malcev, B. Donnet. *Training Students' Abstraction Skills Around a* CAFÉ. May 2023.
6. S. Liénardy, L. Leduc, D. Verpoorten, B. Donnet. *Challenges, Multiple Attempts, and Trump Cards – A Practice Report of Student's Exposure to an Automated Correction System for a Programming Challenges Activity* In International Journal of Technologies in Higher Education (IJTHE). 18(2), pg. 45–60. June 2021.
7. S. Liénardy, B. Donnet. *GameCode : Choose Your Own Problem Solving Path* In Proc. ACM International

dans la plateforme Café qui vous permet de réaliser l'exercice par vous-même en suivant le schéma de résolution qui vous intéresse. Chaque étape de la résolution dispose d'un rappel théorique, d'un ou plusieurs indices si vous êtes bloqués et de la description du raisonnement amenant à la solution. L'idée derrière le GameCode n'est donc pas de lire, linéairement, l'exercice du premier caractère au dernier. Mais bien de faire l'exercice par vous-même et de naviguer dedans (grâce aux divers liens interactifs) afin de vous aider (ou vous corriger). Vous pouvez assimilier ce type d'exercices aux livres-jeux (ou livres dont vous êtes le héros) qui ont connu une certaine gloire dans les 80's. Un GameCode contient aussi des informations de feedforward (i.e., des pointeurs vers le cours théorique pour vous aider à appréhender les points clés de la matière). Chaque fois que la matière d'un chapitre aura été vue (séance théorique + répétition), un GameCode apparaîtra sur Café. A vous de le faire (n'oubliez pas que, lors de l'interro de mi-quadri et l'examen, vous serez seul face à votre copie). Les GameCodes n'interviennent pas dans la note finale et peuvent être fait autant que vous le désirez.

Enfin, un examen d'une session antérieure (Janvier 2014) est disponible sur eCampus. En fin de quadrimestre, il est vivement conseillé à chaque étudiant de faire, à la maison, cet examen en se mettant en "situation". Le correctif est, lui aussi, disponible sur eCampus. Dans tous les cas, l'équipe pédagogique reste à votre disposition pour discuter de vos solutions à cet examen.

Computing Education Research Conference (ICER), Poster Session. August 2020.

Première partie

Séances de Répétition

Chapitre 1

Blocs, Variables, Instructions Simples

L'objectif de ce premier chapitre d'exercices est de passer en revue les éléments de base vus dans l'Introduction et ceux portant sur les bases du langage C (Chapitre 1). Les points abordés dans ce chapitre sont :

- la manipulation d'algorithmes (Sec. 1.1).
- la représentation binaire des entiers (Sec. 1.2).
- La manipulation des divers opérateurs C (Sec. 1.3).
- Les types primitifs (Sec. 1.4).
- Le bloc d'instructions (Sec. 1.5).
- Les premiers pas en écriture de code C (Sec. 1.6).

1.1 Manipulation d'Algorithmes

Dans ces exercices, nous allons manipuler des algorithmes. Pour chacun des algorithmes à appliquer, vous devez indiquer clairement toutes les étapes de votre raisonnement.

1.1.1 Calcul écrit

À l'occasion de l'apprentissage des opérations élémentaires en mathématique, les élèves de l'enseignement primaire apprennent non seulement à effectuer des opérations simples (mettant en jeu de petits nombres) mentalement mais aussi à décomposer des calculs impliquant de grands nombres à l'aide du *calcul écrit*.

Les règles d'un calcul écrit impliquent notamment :

- de découper l'opération à réaliser en opérations plus simples ;
- de retenir le résultat de toutes les opérations simples possibles [1] ;
- de combiner les résultats des opérations simples. La manière de le faire est la plupart du temps aidée par un *agencement particulier* [2] des chiffres qui composent les nombres à manipuler.

Ces règles consistent donc à répéter plusieurs fois des opérations simples et constituent, dès lors, des algorithmes. L'ordinateur sur lequel est encodé l'algorithme n'est autre que l'élève de primaire lui-même. D'ailleurs, tout comme un ordinateur, on attend d'un bon élève qu'il applique le mieux possible les règles du calcul écrit, pas qu'il puisse expliquer *pourquoi* la méthode qu'on lui a demandé d'appliquer est correcte.

1. Souvent à l'aide de moyens mnémotechniques, comme les tables de multiplications.
2. C'est-à-dire que ces chiffres sont placés d'une manière particulière qu'il convient de respecter.

1.1.1.1 Addition écrite

Soit un nombre A, composé des chiffres $(a_{n-1}a_{n-2}\dots a_1a_0)_{10}$ et un nombre B, composé des chiffres $(b_{m-1}b_{m-2}\dots b_1b_0)_{10}$. On demande que $A \geq B$ (si ce n'était pas le cas, il suffit d'inverser les rôles entre A et B).

Disposition des chiffres des nombres Il faut dessiner $n+1$ colonnes numérotées de n à 0 de la gauche vers la droite. Sur une première ligne, on encode le nombre A de la manière suivante : le chiffre a_0 est placé dans la colonne 0, le chiffre a_1 est placé dans la colonne 1 et ainsi de suite jusqu'au chiffre a_{n-1}. Le nombre B est encodé juste en dessous du nombre A, d'une manière similaire (chiffre b_i dans la colonne i). On trace ensuite une ligne horizontale en dessous du nombre B. À la droite du nombre B, on inscrit dans la colonne n le symbole « + » qui représente l'addition.

Opérations à répéter Cette opération requiert de connaitre tous les résultats possibles des additions du type $a+b+c$, avec $a, b \in [0\dots9]$ et $c \in \{0,1\}$.

Pour chaque colonne, en partant de la colonne 0 jusque la colonne $n-1$, il faut répéter cette opération :

> Additionner les chiffres qui sont présents dans la colonne au dessus de la ligne horizontale.
> Si le résultat est représentable sur un seul chiffre, inscrire ce chiffre sous la ligne horizontale, toujours dans la même colonne.
> Si le résultat n'est pas représentable sur un chiffre (il tient donc sur deux chiffres), inscrire le chiffre de droite dans la présente colonne, sous la barre horizontale et inscrire le chiffre de gauche (normalement, c'est un « 1 ») au dessus du chiffre constituant le nombre A dans la première colonne qui se trouve à gauche de la colonne dont les chiffres viennent d'être additionnés. On dit qu'il se produit un *report*.
> Si un report à lieu à la colonne $n-1$, le chiffre de gauche doit directement être écrit sous la barre horizontale, dans la colonne n.

Lecture du résultat Les chiffres inscrits sous la ligne horizontale sont lus de la gauche vers la droite pour former le résultat de l'addition appelé *somme*.

▷ **Exercice 1** Appliquer l'algorithme d'addition écrite pour calculer $78594 + 456$.

1.1.1.2 Soustraction écrite

Voici un exemple de la soustraction $3496 - 2987$ ($= 509$) :

	3	4^{+10}	9	6^{+10}
-	2_{+1}	9	8_{+1}	7
	0	5	0	9

En vous basant sur l'exemple précédent, proposez une description de l'algorithme de soustraction écrite. Utilisez des termes précis. Une personne qui n'a jamais vu de soustraction écrite doit pouvoir comprendre, sur base de votre description, comment fonctionne une soustraction écrite. Tenez compte que :

- Tous les résultats des soustractions $a-b$, avec $a \geq b$, $a \in [0,19]$, $b \in [0,10]$ sont supposés connus ;
- Les résultats des opérations $x+1$ et $x+10$, avec $x \in [0,9]$ sont supposés connus ;

- Il est inutile d'expliquer *pourquoi* l'emprunt (écrire « +10 » au dessus d'une colonne et « +1 » en dessous dans la colonne qui est juste à sa gauche) fonctionne et permet d'obtenir un résultat correct. Par contre, il convient de décrire précisément l'opération.

1.1.2 Multiplication à la Russe

La technique de multiplication dite *à la russe* consiste à diviser par deux le *multiplicateur* (et ensuite les quotients obtenus) jusqu'à un quotient nul, à noter les restes et à multiplier parallèlement le multiplicande par deux. On additionne alors les multiples obtenus du multiplicande correspondant aux restes non nuls.

Dans le produit de **6** par **4** (i.e., 6×4), on dit que

- **6** est le *multiplicande*, car c'est lui qui est répété (multiplier 6 par 4 revient à calculer $6 + 6 + 6 + 6$) ;
- **4** est le *multiplicateur*, car il indique combien de fois 6 doit être répété.

Par exemple, 123×68 :

multiplicande $M \times 2$	**multiplicateur** $m \div 2$	**reste** $m \bmod 2$	**somme partielle**
123	68	0	$(0 \times 123) + 0$
246	34	0	$(0 \times 246) + 0$
492	17	1	$(1 \times 492) + 0$
984	8	0	$(0 \times 984) + 492$
1968	4	0	$(0 \times 1968) + 492$
3936	2	0	$(0 \times 3936) + 492$
7872	1	1	$(1 \times 7872) + 492$
		$68 \times 123 =$	8364

▷ **Exercice 1** Effectuer la multiplication suivante selon la technique "à la russe" : 64×96

▷ **Exercice 2** Effectuer la multiplication suivante selon la technique "à la russe" : 45×239

1.1.3 Multiplication Arabe

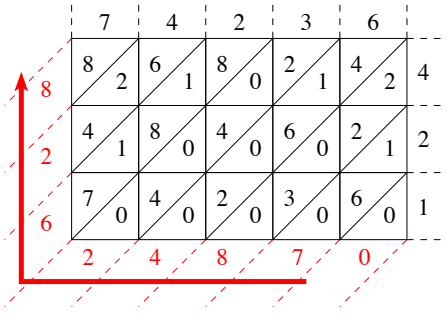


FIGURE 1.1 – Exemple de multiplication arabe : $63247 \times 124 = 7842628$.

On considère ici le texte d'Ibn al-Banna concernant la multiplication à l'aide de tableaux [3]

> Tu construis un quadrilatère que tu subdivises verticalement et horizontalement en autant de bandes qu'il y a de positions dans les deux nombres multipliés. Tu divises diagonalement les carrés obtenus à l'aide de diagonales allant du coin inférieur gauche au coin supérieur droit.
> Tu places le multiplicande au-dessus du quadrilatère, en faisant correspondre chacune de ses position à une colonne. [a] Puis, tu places le multiplicateur à gauche ou à droite du quadrilatère, de telle sorte qu'il descende avec lui en faisant correspondre égaglement chacune de ses position à une ligne. [b] Puis, tu multiplies, l'une après l'autre, chacune des positions du multiplicande du carré par toutes les positions du multiplicateur et tu poses le résultat partiel correspondant à chaque position dans le carré où se coupent respectivement leur colonne et leur ligne, en plaçant les unités au-dessus de la diagonale et les dizaines en dessous. Puis, tu commences à additionner, en partant du coin supérieur gauche : tu additionnes ce qui est entre les diagnoles, sans effacer, en plaçant chaque nombre dans sa position, en transférant les dizaines de chaque somme partielle à la diagonale suivante et en les ajoutant à ce qui y figure.
>
> *a.* Attention, l'écriture du nombre se fait de droite à gauche (e.g., 352 s'écrira donc 253).
>
> *b.* Attention, l'écriture du nombre s'effectue de bas en haut (e.g., $\begin{matrix}3\\5\\2\end{matrix}$) s'écrira donc $\begin{matrix}2\\5\\3\end{matrix}$.

La Fig. 1.1 illustre un exemple de multiplication arabe.

▷ **Exercice 1** En utilisant la méthode du tableau d'Ibn al-Banna, calculer 35617×1029

1.2 Représentation Binaire des Nombres

Dans ces exercices, nous allons manipuler l'algorithme de conversion d'un entier, en base décimale, en un entier en base binaire (et vice versa). Pour chacune des questions, vous veillerez à indiquer clairement toutes les étapes de votre raisonnement.

1.2.1 Du Décimal vers le Binaire

▷ **Exercice 1** Donnez, sur 8 bits, la représentation binaire des entiers non signés

1. 15
2. 20
3. 71
4. 100
5. 110
6. 133
7. 154

▷ **Exercice 2** Donnez la représentation binaire, sur 8 bits des entiers signés

1. -5
2. -2
3. -83
4. 42
5. -13

3. J.-L. Chabert, E. Barbin, M. Guillemot, A. Michel-Pajus, J. Borowczyk, A. Djebbar, J.-C. Martzloff. *Histoire d'Algorithmes : du Caillou à la Puce*. Ed. Belin. 1994.

1.2.2 Du Binaire vers le Décimal

▷ **Exercice 1** Représentez en base 10 les entiers non signés dont la représentation binaire est

1. 00010010
2. 00101100
3. 00100000
4. 00001001
5. 00110001

▷ **Exercice 2** Représentez en base 10 les entiers signés dont la représentation binaire est

1. 10000010
2. 10000001
3. 10111101
4. 00011001
5. 00001001
6. 11111100

1.2.3 Réflexion

▷ **Exercice 1** "There are 10 kinds of people. Those who understand binary notation, and those who don't." Expliquez.

1.3 Manipulation d'Opérateurs

1.3.1 Table de Vérité

▷ **Exercice 1** Écrivez les tables de vérité pour les expressions suivantes :

1. A || (!A && B)
2. A && (A || B)
3. !A &&!B

Vous veillerez à indiquer toutes les étapes de votre raisonnement, pas uniquement le résultat final.

1.3.2 Affectation

▷ **Exercice 1** Écrivez, si possible, de la manière la plus compacte possible les expressions suivantes :

1. `y = x + y;`
2. `x = x + 1;`
3. `y = y + 3;`
4. `b1 = b1 && b2;`
5. `b1 = b1 == b3;`
6. `b2 = b2 + 1;`

1.3.3 Opérateur bit-à-bit

▷ **Exercice 1**

Reprenons certaines représentations binaires (que vous avez déjà évaluées) de la section 1.2.2 et dérivons une représentation binaire correspondante :

Représentation binaire présentée ci-dessus	Représentation binaire correspondante
(1.2.2 - ex1.1) 00010010 (non-signé)	00100100 (non-signé)
(1.2.2 - 1.2) 00101100 (non-signé)	01011000 (non-signé)
(1.2.2 - ex1.3) 00100000 (non-signé)	01000000 (non-signé)

TABLE 1.1 – Représentations binaires non-signées : Relation ?

1. Quelle transformation applique-t-on à la représentation binaire de gauche pour obtenir la représentation binaire de droite ?
2. Quel opérateur réalise cette opération ?
3. Quelle est la relation entre les paires de valeurs représentées en binaire ?

▷ **Exercice 2**

Reprenons certaines représentations binaires (que vous avez déjà évaluées) de la section 1.2.2 et dérivons une représentation binaire correspondante :

Représentation binaire présentée ci-dessus	Représentation binaire correspondante
(1.2.2 - ex2.4) 00011001 (signé)	00110010 (signé)
(1.2.2 - ex2.5) 00001001 (signé)	00010010 (signé)
(1.2.2 - ex2.6) 11111100 (signé)	11111000 (signé)

TABLE 1.2 – Représentations binaires signées : Relation ?

1. Quelle transformation applique-t-on à la représentation binaire de gauche pour obtenir la représentation binaire de droite [4] ?
2. Quel opérateur réalise cette opération ?
3. Quelle est la relation entre les paires de valeurs représentées en binaire ?
4. Pourquoi cette relation ne s'applique pas aux représentations binaires (signées) suivantes : "10111101", "10000010"" et "10000001"" ?

▷ **Exercice 3**

Reprenons certaines représentations binaires (que vous avez déjà évaluées) de la section 1.2.2 et dérivons une représentation binaire correspondante :

1. Quelle transformation applique-t-on à la représentation binaire de gauche pour obtenir la représentation binaire de droite ?
2. Quel opérateur réalise cette opération ?
3. Quelle est la relation entre les paires de valeurs représentées en binaire ?

4. Indice : elle est similaire à précédemment, à une variante près. Quelle est cette variante ?

Représentation binaire présentée ci-dessus	Représentation binaire correspondante
(1.2.2 - ex1.1) 00010010 (non-signé)	00001001 (non-signé)
(1.2.2 - ex1.2) 00101100 (non-signé)	00010110 (non-signé)
(1.2.2 - ex1.3) 00100000 (non-signé)	00010000 (non-signé)

TABLE 1.3 – Représentations binaires non-signées : Relation ?

1.3.4 Priorité

▷ **Exercice 1** Déterminez le résultat des expressions suivantes, avec `x = 1` et `y = 2` :

1. `2 + 3 * 4 + 12 % 3`
2. `1 != (x++ == --y)`

N'hésitez pas à représenter graphiquement les expressions, comme dans le cours théorique (Chapitre 1, Slides 36 → 45).

▷ **Exercice 2** Soit un programme contenant les déclarations suivantes :

```
int i = 8;
int j = 5;
float x = 0.005;
float y = -0.01;
char c = 'c';
char d = 'd';
```

Pour rappel, la valeur entière associée au caractère `'a'` dans la table ASCII vaut 97.

Déterminez la valeur de chacune des expressions suivantes :

1. `(3 * i - 2 * j) % (2 * d - c)`
2. `2 * ((i / 5) + (4 * (j - 3)) % (i + j - 2))`
3. `i <= j`
4. `j != 6`
5. `c == 99`
6. `5 * (i + j) > 'c'`
7. `(i > 0) && (j < 5)`
8. `(i > 0) || (j < 5)`
9. `(x > y) && (i > 0) || (j < 5)`
10. `(x > y) && (i > 0) && (j < 5)`

N'hésitez pas à représenter graphiquement les expressions, comme dans le cours théorique (Chapitre 1, Slides 36 → 45).

▷ **Exercice 3** Évaluez les expressions suivantes en supposant `a = 20`, `b = 5`, `c = -10`, `d = 2`, `x = 12` et `y = 15`. Notez chaque fois la valeur rendue comme résultat de l'expression et les valeurs des variables dont le contenu a changé.

1. `(5 * x + 2 * ((3 * b) + 4))`
2. `(5 * (x + 2) * 3) * (b + 4)`
3. `a == (b = 5)`
4. `a += (x + 5)`
5. `a != (c *= (-d))`

6. `a *= c + (x - d)`
7. `a %= d++`
8. `a %= ++d`
9. `x++ * (a + c)`
10. `a = x * (b < c) + y * !(b < c)`
11. `!(x - d + c) || d`
12. `a && b || !0 && c && !d`
13. `((a && b) || (!0 && c)) && !d`
14. `((a && b) || !0) && (c && (!d))`

N'hésitez pas à représenter graphiquement les expressions, comme dans le cours théorique (Chapitre 1, Slides 36 → 45).

1.4 Les Types

1.4.1 Déclaration

▷ **Exercice 1** Quel type de variable utiliseriez-vous pour stocker :

- le nombre d'étudiant en premier Bloc ?
- le PIB (Produit Intérieur Brut) [5] de l'état Belge en euros ?
- π ?
- les votes exprimés pour un candidat à une élection ?
- le nombre de fûts dans les caves de votre cercle préféré ?

1.4.2 Conversion de Type

▷ **Exercice 1** Soit les déclarations suivantes :

```
double money = 210.7;
int euro = 25;
double d;
int i;
```

Décrivez avec précision l'effet de chacune des instructions suivantes :

```
d = euro;
d = (double) euro;
i = money;
i = (int) money;
```

▷ **Exercice 2** Comparez l'effet des instructions suivantes sur la valeur de la variable `franc` :

```
int franc;
franc = 125 / 50;
franc = 125 / 50.0;
franc = 125.0 / 50.0;
```

5. Le PIB vise à quantifier (pour un pays et une année donnés) la valeur totale de la "production de richesse" effectuée par les agents économiques (ménages, entreprises, administrations publiques) résidants à l'intérieur de ce territoire. Pour plus d'infos, cf. wikipedia.

1.5 Bloc d'Instructions

▷ **Exercice 1** Soit le code suivant :

```
int main(){
  int a;
  int b;
  int c;
  a = 1;
  b = a + 1;
  c = a + b;
  a = b;
}//fin programme
```

Pour chaque ligne de code, indiquez quelles sont les variables qui sont modifiées et les valeurs qu'elles prennent.

1.6 Écriture de Code

▷ **Exercice 1** Lors d'une opération de promotion, un magasin de composants *hardware* applique une réduction de 10% sur tous les composants. Ecrivez un programme qui lit le prix d'un composant au clavier et affiche le prix calculé en tenant compte de la réduction.

▷ **Exercice 2** Une bille de plomb est lâchée du haut d'un immeuble et tombe en chute libre. Au bout d'un temps t (exprimé en secondes), la bille est descendue d'une hauteur (en mètres) h. La valeur de h peut être calculée comme suit :

$$h = \frac{1}{2} g \times t^2.$$

avec $g = 9.81$ (exprimé en $m.s^{-2}$).

Dans un premier temps, écrivez un programme qui calcule la hauteur descendue au bout d'un temps t saisi au clavier.

Dans un second temps, écrivez un programme qui calcule la durée totale de la chute connaissant la hauteur totale h de l'immeuble saisi au clavier. On pourra, ici, utiliser la bibliothèque `math.h` qui contient un module, `sqrt`, permettant de calculer la racine carrée d'un nombre. L'utilisation de `sqrt` se fait comme suit (par exemple) :

```
#include <math.h>
#include <stdio.h>

int main(){
  int x = 16;
  double racine;

  racine = sqrt(x);
  printf("La racine carree de %d est %f\n", x, racine);
}//fin programme
```

▷ **Exercice 3** Sachant que le premier avril 2004 était un jeudi, écrivez un programme qui détermine le jour de la semaine correspondant au 4 mai de la même année. On pourra représenter les jours de la semaine par des nombres allant de 0 (lundi) à 6 (dimanche).

▷ **Exercice 4** Écrivez un programme qui affiche le chiffre des dizaines d'un nombre saisi au clavier. Même question pour les centaines.

▷ **Exercice 5** Écrivez un programme qui lit un nombre r au clavier et calcule le périmètre et l'aire d'un disque de rayon r.

Chapitre 2

Structures de Contrôle

L'objectif de ce chapitre est de se concentrer sur les structures de contrôle. Ces structures nous permettent, soit d'effectuer des tests sur les valeurs des variables (*condition* – Sec. 2.1), soit de répéter un certain nombre de fois un bloc d'instructions particulier (*itération* – Sec. 2.2).

2.1 Condition

2.1.1 Lecture de Code

L'objectif de cette section est de vous apprendre à lire et comprendre du code écrit par un autre programmeur. Soyez clair et précis dans vos réponses. Inutile d'exprimer les choses en paraphrasant le code C (i.e., *Si la variable* `i` *est plus petite que ...*), ce qui correspondrait à une traduction *littérale* du code. Pensez plutôt en terme de *transcodage* (ou traduction *littéraire*) en indiquant l'objectif final du bout de code.

La meilleure façon de résoudre les exercices qui suivent est de travailler avec des valeurs pour chacune des variables, d'exécuter, sur papier, le code avec ces valeurs et, ensuite, essayer d'inférer une relation entre les différentes variables. C'est cette relation qui vous donnera la réponse.

▷ **Exercice 1** Soit le morceau de code suivant :

```
1 if(i < j)
2   k = j;
3 else
4   k = i;
```

Que calcule ce code (i.e., que contient la variable `k` à la fin de l'exécution de ce code) ? Soyez le plus précis possible (notamment dans le vocabulaire que vous utilisez).

▷ **Exercice 2** Soit le morceau de code suivant :

```
1 if(i > j)
2   k = i;
3 else
4   k = j;
```

Que calcule ce code (i.e., que contient la variable `k` à la fin de l'exécution de ce code) ? Soyez le plus précis possible (notamment dans le vocabulaire que vous utilisez).

▷ **Exercice 3** Soit le morceau de code suivant :

```
1 int k;
2 if(i != j){
3   k = i;
4   i = j;
5   j = k;
```

```
}
```

Que calcule ce code (i.e., que contiennent les variables `k`, `i` et `j` à la fin de l'exécution de ce code) ? Soyez le plus précis possible (notamment dans le vocabulaire que vous utilisez). En particulier, quelle est l'utilité de la variable `k` ?

Pour répondre à cette question, en plus de tester différentes valeurs pour `i` et `j`, il est recommandé de faire un schéma représentant les variables et leur contenu à chaque étape du programme.

▷ **Exercice 4** Soit le morceau de code suivant :

```
int l;
if(i > j){
  l = i;
  i = j;
  j = l;
}
if(i > k){
  l = i;
  i = k;
  k = l;
}
if(j > k){
  l = j;
  j = k;
  k = l;
}
```

Que calcule ce code (i.e., que contiennent les variables `k`, `i` et `j` à la fin de l'exécution de ce code) ? Soyez le plus précis possible (notamment dans le vocabulaire que vous utilisez). En particulier, quelle est l'utilité de la variable `l` ?

Pour répondre à cette question, en plus de tester différentes valeurs pour `i` et `j`, il est recommandé de faire un schéma représentant les variables et leur contenu à chaque étape du programme.

2.1.2 Écriture de Code

▷ **Exercice 1** Une entreprise X vend deux types de produits. Les produits de type A qui donnent lieu à une TVA à 5,5% [1] et les produits de type B qui donnent lieu à une TVA à 19,6% [2]. Écrivez un programme qui lit au clavier le prix hors taxe d'un produit, saisit au clavier le type du produit et affiche le taux de TVA et le prix TTC (i.e., Toutes Taxes Comprises) du produit. Appliquer une taxe de X% revient à additionner, au prix du produit, X% du prix de ce produit. En d'autres termes,

$$\text{prix}_{\text{TTC}} = \text{prix} + \text{prix} \times \text{taux\_TVA}$$

▷ **Exercice 2** Écrivez un programme qui demande à l'utilisateur de saisir un entier relatif x ($x \in \mathbb{Z}$) et qui indique, d'une part, si ce nombre est positif ou négatif et, d'autre part, si ce nombre est pair ou impair.

▷ **Exercice 3** Écrivez un programme qui demande à l'utilisateur de saisir trois nombres réels (`double`) au clavier et les affiche à l'écran par ordre croissant.

▷ **Exercice 4** Écrivez un programme qui résout l'équation $ax + b = 0$ (a et b sont entrés au clavier par l'utilisateur). Bien évidemment, on n'oubliera pas tous les cas particuliers (notamment les cas "tout x est solution" et "pas de solution").

1. $5,5\% = 0.055$
2. $19,6\% = 0.196$

▷ **Exercice 5** Écrivez un programme qui résout l'équation $ax^2 + bx + c = 0$ (a, b, c sont entrés au clavier par l'utilisateur) en envisageant tous les cas particuliers.

▷ **Exercice 6** Écrivez un programme qui lit un entier au clavier représentant le numéro du mois et qui affiche, ensuite, à l'écran le mois en toutes lettres (janvier=1, février=2, ...). Pensez à utiliser l'instruction conditionnelle multiple (i.e., `switch`).

▷ **Exercice 7** Pour rappel, un nombre naturel correspond à une *année bissextile* s'il est multiple de 4 ; mais, parmi les multiples de 100, seuls les multiples de 400 correspondent à une année bissextile.

Écrivez un programme qui détermine si un nombre naturel entré par l'utilisateur correspond à une année bissextile ou non.

▷ **Exercice 8** Écrivez un programme qui, étant donné une heure représentée sous la forme de trois variables pour les heures, `h`, minutes, `m`, et secondes, `s`, affiche l'heure qu'il sera une seconde plus tard. Il faudra envisager tous les cas possibles pour le changement d'heure. Deux exemples de sortie (à l'écran) sont :

```
L heure actuelle est 23h12m12s
Dans une seconde il sera exactement: 23h12m13s

L heure actuelle est 23h59m59s
Dans une seconde il sera exactement: 00h00m00s
```

2.2 Itération

2.2.1 Lecture de Code

A l'instar de la Sec. 2.1.1, l'objectif de cette section est de vous apprendre à lire et comprendre du code écrit par un autre programmeur. Soyez clair et précis dans vos réponses. Inutile d'exprimer les choses en paraphrasant le code C (i.e., *la boucle s'exécute tant que la variable* `i` *est plus petite que ...*)), ce qui correspondrait à une traduction *littérale* du code. Pensez plutôt en terme de *transcodage* (ou traduction *littéraire*) en indiquant l'objectif final du bout de code.

La meilleure façon de résoudre les exercices qui suivent est de travailler avec des valeurs pour chacune des variables, exécuter le code avec ces valeurs et, ensuite, essayer d'inférer une relation entre les différentes variables. C'est cette relation qui vous donnera la réponse.

▷ **Exercice 1** Soit le morceau de code suivant :

```
1 i=1;
2 k=0;
3 while(i < j){
4   k = k + i;
5   i = i + 1;
6 }//fin while
```

Que calcule ce code (i.e., que contient la variable `k` à la fin de l'exécution de ce code) ? Soyez le plus précis possible (notamment dans le vocabulaire que vous utilisez).

▷ **Exercice 2** Soit le morceau de code suivant :

```
1 i=1;
2 k=1;
3 while(i < j){
4   k = k * i;
5   i = i + 1;
6 }//fin while
```

Que calcule ce code (i.e., que contient la variable `k` à la fin de l'exécution de ce code) ? Soyez le plus précis possible (notamment dans le vocabulaire que vous utilisez).

2.2.2 Manipulation de Boucles

▷ **Exercice 1** Soit le code suivant :

```c
#include <stdio.h>

int main(){
  int n = 5;
  int f = 1;
  int c = n;

  while(c > 1){
    f = f * c;
    c = c - 1;
  }//fin while

  printf("%d\n", f);
}//fin programme
```

- Que calcule ce code (i.e., que vaut `f` à la fin) ?
- Réécrivez ce code avec un compteur qui est incrémenté d'une unité à chaque itération plutôt que d'être décrémenté.

▷ **Exercice 2** Soit le code suivant :

```c
#include <stdio.h>

int main(){
  int i;

  for(......; ......; ......)
    printf("It's␣a␣long␣way␣to␣the␣top␣if␣you␣wanna␣Rock␣'n'␣Roll!\n");
}//fin programme
```

Complétez ce bout de code pour que le programme affiche **cinq** fois à l'écran la phrase "It's a long way to the top if you wanna Rock 'n' Roll !".

▷ **Exercice 3** Réécrivez le code de l'Exercice 2 en utilisant, cette fois, une boucle `while` plutôt que `for`.

▷ **Exercice 4** Réécrivez le code de l'Exercice 2 en utilisant, cette fois, une boucle `do ... while` plutôt que `for`.

▷ **Exercice 5** Réécrivez la boucle de l'Exercice 1 en utilisant l'instruction `for` plutôt que `while`.

▷ **Exercice 6** Réécrivez la boucle de l'Exercice 1 en utilisant l'instruction `do ... while` plutôt que `while`.

2.2.3 Écriture de Code

Pour chacun de ces exercices, vous veillerez à bien suivre la méthodologie vue au cours pour la construction d'une boucle. **En particulier, il vous est demandé de stipuler explicitement les quatre étapes de rédaction d'une boucle avant de rédiger la moindre ligne de code.** Faites valider les quatre étapes par un membre de l'équipe pédagogique avant de commencer la rédaction de votre code C.

Pour rappel, les quatres étapes sont les suivantes (Chap. 2, Slide 34) :

Étape 1 : déclarer une variable qui va servir de compteur et l'initialiser ;

Étape 2 : déterminer le nombre de tours de la boucle ;

Étape 3 : utiliser les 2 premières étapes pour déterminer le Gardien de Boucle ;

Étape 4 : déterminer les instructions composants le Corps de la Boucle.

▷ **Exercice 1** Écrivez un programme qui affiche, à l'écran, la table de multiplication pour un entier lu au clavier.

▷ **Exercice 2** Écrivez un programme qui calcule la somme :

$$s = 1 + 2^3 + 3^3 + \ldots + n^3.$$

en fonction de n saisi au clavier.

▷ **Exercice 3** Considérons que a, b et c sont des variables entières lues au clavier. Écrivez un programme permettant de calculer :

- la somme des entiers dans l'intervalle $[a, b[$
- la somme des entiers positifs dans l'intervalle $]a, b[$
- la somme des entiers pairs strictement inférieur à b
- la somme des diviseurs de c dans l'intervalle $[a, b]$
- le nombre d'entiers qui sont strictement inférieurs à c et à la fois multiple de a et b.

▷ **Exercice 4** Écrivez un programme qui calcule la factorielle $n!$ d'un entier n saisi au clavier.

▷ **Exercice 5** Écrivez un programme qui calcule le nombre de chiffres en base 10 d'un nombre n saisi au clavier.

▷ **Exercice 6** Écrivez un programme qui affiche la plus grande puissance de 2 inférieure à la constante $C = 2.426.555.645$.

▷ **Exercice 7** Écrivez un programme qui permet facilement d'afficher à l'écran un rectangle comme le suivant :

```
+----------------------+
|                      |
|                      |
|                      |
+----------------------+
```

La largeur et la longueur de ce rectangle seront saisies au clavier.

▷ **Exercice 8** En mathématiques, l'intégrale permet, entre autres, de calculer la surface de l'espace délimité par la représentation graphique d'une fonction. L'intégrale de la fonction $f(x)$ sur l'intervalle $[a, b]$ est représentée par

$$\int_a^b f(x)dx.$$

Il est possible d'approximer l'intégrale d'une fonction à l'aide de la méthode des *rectangles*. Cette méthode fonctionne comme suit : l'intervalle $[a, b]$ sur lequel la fonction $f(x)$ doit être intégrée est divisé en N sous-intervalles égaux de longueurs $h = \frac{(b-a)}{N}$. Les rectangles sont alors dessinés de sorte que le coin supérieur gauche, droit ou l'entièreté du côté supérieur touche le graphe de la fonction, tandis que la base se tient sur l'axe des X. L'approximation de l'intégrale

est alors calucée en ajoutant les surfaces (base $\times$ hauteur) des rectangles. Ce qui donne la formule :

$$I = h \times \sum_{i=0}^{N-1} f(a + i \times h).$$

Écrivez un programme qui permette d'approximer, par la méthode des rectangles, l'intégrale de $cos(x)$. Les valeurs de N, a, b et x seront lues au clavier. Le module `cos()` se trouve dans la librairie `math.h`.

Chapitre 3

Méthodologie de Développement

La matière vue dans les deux premiers chapitres du cours vous permet, déjà, de résoudre pas mal de problèmes. L'objectif de ce chapitre est donc d'augmenter vos compétences en algorithmique et en écriture de code. En particulier, nous allons nous attarder sur la façon dont on peut construire efficacement et correctement une solution à un problème. La méthode proposée fonctionne en quatre étapes :

Étape 1 : Définition du Problème (Sec. 3.1). Il s'agit, ici, de reformuler le problème en fonction des Inputs et Outputs et de caractériser les Inputs. La Définition du Problème peut se représenter textuellement et/ou graphiquement.

Étape 2 : Analyse du Problème (Sec. 3.2). Il s'agit, ici, de décomposer un problème plus complexe en un ensemble de Sous-Problèmes (Sp) et de les structurer. Chaque Sp doit être correctement défini.

Étape 3 : Construction par Invariant Graphique (Sec. 3.3). L'idée, ici, est de représenter graphiquement le travail d'une boucle via un Invariant Graphique et de l'utiliser pour construire les instructions avant la boucle (Zone 1), le Critère d'Arrêt, les instructions du corps de la boucle (Zone 2) et les instructions après la boucle (Zone 3). Enfin, la Fonction de Terminaison permet de prouver (mathématiquement) que la boucle se termine.

Étape 4 : Tests. Cette quatrième étape sera abordée, au Q2, dans le cadre du cours INFO0030 (Projets de Programation).

Pour les exercices qui suivent, vous allez disposer d'une fiche de validation. Il vous est donc demander d'explicitement confronter votre solution avec cette fiche avant d'interpeller l'équipe pédagogique pour vérifier votre solution. N'hésitez pas non plus à vous appuyer sur la fiche de validation pour construire votre solution.

3.1 Étape 1 : Définition du Problème

3.1.1 Fiche de Validation

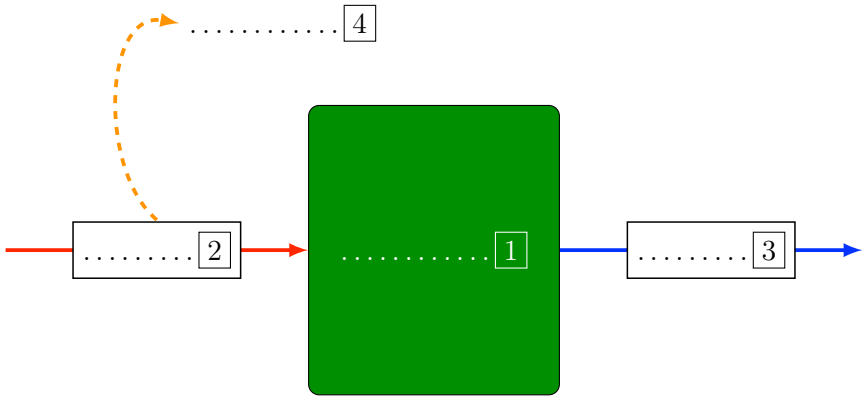
La fiche de validation pour l'Étape 1 est fournie par le tableau 3.1. Une case doit être cochée si l'item est présent. Une explication textuelle peut, éventuellement, être ajoutée. Plusieurs instances de cette fiche sont disponibles à l'Annexe 1.

Critères de l'Étape 1 (définition)	**Cohérent ?**	**Commentaires**
Input(s)	☐	
Output	☐	
Caractérisation des Inputs		
- font référence aux Inputs (et non aux variables propres à votre solution (variables d'itération, ...))	☐	
- ont un nom significatif	☐	
- ont un type existant en C	☐	
- ont une utilité clairement exprimée	☐	
Représentation Graphique		
- les Inputs sont indiqués	☐	
- les Inputs sont caractérisés	☐	
- l'Output est clairement indiqué	☐	

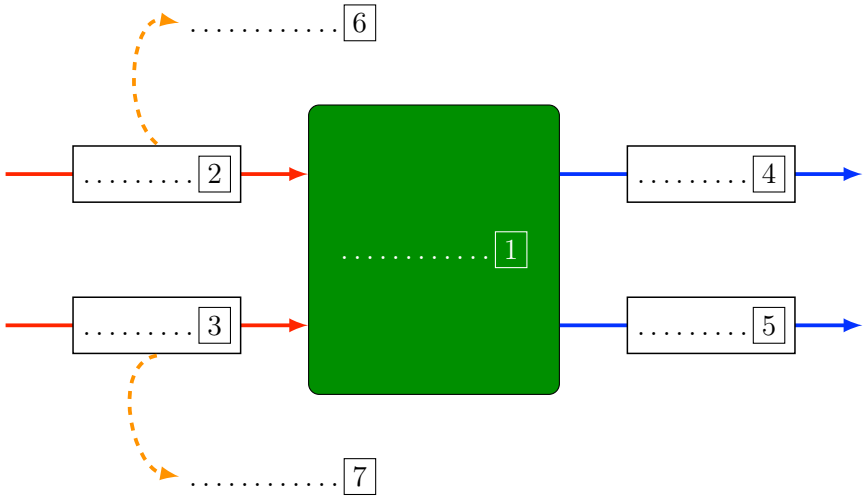
TABLE 3.1 – Fiche de validation pour l'étape 1 (Définition du Problème).

3.1.2 Définition à Compléter

▷ **Exercice 1** Soit le problème suivant : Convertir un nombre donné, exprimé en base décimale (base 10), vers une représentation hexadécimale (base 16). Compléter la représentation graphique ci-dessous définissant le problème :



▷ **Exercice 2** Soit le problème suivant : Pour un intervalle donné, il est demandé de sommer les nombres compris dedans (bornes comprises) et d'afficher la taille de l'intervalle. Compléter la représentation graphique ci-dessous définissant le problème :



3.1.3 Exercices Complets

Dans les énoncés qui suivent, il ne vous est pas demandé de rédiger le code du programme. Vous devez vous limiter à l'Étape 1 de la méthodologie.

Pour chacun des exercices, vous devez donc donner la Définition du Problème. Cette définition doit être présentée de deux façons :

1. textuelle (i.e., Input(s), Output, Caractérisation des Inputs) ;
2. graphique (i.e., voir la Sec. 3.1.2).

Pensez à vous aider de la fiche de validation (Tableau 3.1 et Annexe 1).

▷ **Exercice 1** Il est demandé d'afficher à l'écran une ligne composée de x (lu au clavier) caractères (le caractère à afficher est lui aussi lu au clavier). La ligne est délimitée à gauche par le caractère `'<'` et à droite par le caractère `'>'`. Les caractères de délimitation ne sont pas inclus dans le nombre de caractères à afficher.

Par exemple, pour $x = 5$ et le caractère `'-'`, il sera affiché à l'écran :

```
<----->
```

De même, pour $x = 10$ et le caractère `'@'`, on obtient :

```
<@@@@@@@@@@>
```

▷ **Exercice 2** Il est demandé d'afficher à l'écran n (lu au clavier) lignes composées de x (lu au clavier) caractères (le caractère à afficher est lu au clavier). Chaque ligne est délimitée à gauche par le caractère `'<'` et à droite par le caractère `'>'`. Les caractères de délimitation ne sont pas inclus dans le nombre de caractères à afficher sur chaque ligne.

Par exemple, pour $n = 3$, $x = 5$ avec le caractère `'-'`, il sear affiché à l'écran :

```
<----->
<----->
<----->
```

De même, pour $n = 5$, $x = 10$ et le caractère `'@'`, on obtient :

```
<@@@@@@@@@@>
<@@@@@@@@@@>
<@@@@@@@@@@>
<@@@@@@@@@@>
<@@@@@@@@@@>
```

▷ **Exercice 3** Il est demandé d'afficher à l'écran un *repère cartésien*. Pour rappel, un repère cartésien est défini par :

- deux droites perpendiculaires, *l'abscisse* est horizontale et *l'ordonnée* verticale ;
- leur intersection, appelée *origine du repère* ;
- quatre quadrants représentants les régions du plan que les droites délimitent.

Dans cet exercice, la longueur de l'abscisse est paire tandis que celle de l'ordonnée est impaire (mais vaut la longueur de l'abscisse + 1). Seule la longueur de l'abscisse est fournie par l'utilisateur (au clavier). L'orgine du repère doit se trouver au "milieu" des deux droites.

Par exemple, pour une absisse de longueur de 6 :

```
    ^
    |
    |
    |
---|--->
    |
    |
    |
```

▷ **Exercice 4** Déterminer si un nombre donné est un palindrome. Pour rappel, un nombre est un palindrome si il peut se lire indifféremment de gauche à droite ou de droite à gauche. Par exemple, 121 est un palindrome.

3.2 Étape 2 : Analyse de Problèmes

3.2.1 Fiche de Validation

La fiche de validation pour l'Étape 2 est fournie par le tableau 3.2. Une case doit être cochée si l'item est présent. Une explication textuelle peut, éventuellement, être ajoutée. Plusieurs instances de cette fiche sont disponibles à l'Annexe 2.

Critères de l'Étape 2 (découpe)	**Cohérent ?**	**Commentaires**
La structuration des SPs est cohérente, complète et permet de résoudre le problème principal	☐	
La structuration graphique des SPs suit les patterns.	☐	
La granularité de la découpe est suffisante.	☐	
SP ..... :		
- a un nom (et une description) explicits	☐	
- répond à un des 4 types de SP	☐	
- est générique (i.e., sa description ne se limite pas à certains exemples spécifiques)	☐	
- a une description fonctionnelle et non technique[2]	☐	
- vient avec une définition complète (Input, Output, C.I.)	☐	

[1] Il ne s'agit **pas** de décrire litéralement des instructions (par exemple, *écrire une boucle* ou encore *incrémenter une variable* ne sont pas des SPs).

[2] Pour rappel, le bon niveau de granularité d'un SP est soit un module (`scanf()` ou `printf()`), soit une boucle.

TABLE 3.2 – Fiche de validation pour l'étape 2 (Analyse du Problème).

3.2.2 Analyse à Compléter

▷ **Exercice 1** Dans la liste suivante, identifiez ce qui correspond à un SP et rattachez chacun d'eux à un des quatre types présentés dans le cours théorique (Chap. 3 – Sec. "Les types de SPs"). Attention, certains items ne correspondent pas à un SP. Dans ce cas, pensez à justifier votre réponse (cfr. Chap. 3 – Sec. "Méthode de Découpe").

1. Lire un nombre au clavier ;
2. Vérifier si un nombre est plus petit qu'un autre ;
3. Déterminer si un nombre en divise un autre ;
4. Afficher une ligne de n caractères $-$;
5. Additionner 2 nombres ;
6. Afficher un nombre ;
7. Déterminer si un nombre est premier ;
8. Caractériser les Inputs ;
9. Compter le nombre de 0 dans la représentation binaire d'un nombre donné (en base 10) ;
10. Initialiser un compteur de boucle ;
11. Compter le nombre de chiffres qui composent un nombre ;
12. Parcourir les chiffres d'un nombre de gauche à droite et supprimer certains chiffres ;
13. Analyser le problème ;
14. Déterminer si un nombre est un carré parfait ;
15. Afficher un trapèze dans le terminal ;
16. Vérifier si un nombre est composé de chiffres pairs ;
17. Déclarer une variable pour stocker une somme ;
18. Additionner des nombres compris dans un intervalle ;
19. Incrémenter la somme.

▷ **Exercice 2** Soit le problème suivant : Pour un nombre et un chiffre donnés, supprimer toutes les occurrences de ce chiffre dans le nombre et, ensuite, aficher le nombre résultant. Répondez aux sous-questions suivantes :

1. à quel exemple présenté dans le cours rattacheriez-vous ce problème ?
2. à partir de la liste présentée dans le premier exercice de la section, quels SPs vous semblent pertinents pour résoudre ce problème ?
3. comment articuleriez-vous ces SPs pour représenter le problème général posé ?

▷ **Exercice 3** Soit le problème suivant : on dispose d'une bibliothèque contenant des livres. [1] L'objectif est de trier (par ordre lexicographique) le contenu de la bibliothèque. Une technique possible de tri [2] est le *tri par insertion*. Pour résoudre ce problème, on définit les SPs suivants (non-ordonnés) :

- SP_1 : Trouver la position où insérer un livre l dans une "sous-bibliothèque".
- SP_2 : Décaler tous les livres d'une sous-bibliothèque d'une position vers la droite.
- SP_3 : Parcourir tous les livres de la bibliothèque et les trier.

1. La façon dont la bibliothèque pourrait être représentée dans notre programme n'a aucun intérêt ici. L'idée n'est pas d'implémenter la solution (le "comment") mais bien de raisonner sur les grandes fonctionnalités du système (le "quoi").
2. Il en exite d'autres, plus efficaces mais, à nouveau, cela n'a aucune importance ici.

- SP_4 : Insérer correctement un livre l dans une sous-bibliothèque.

Comment structureriez-vous ces sous-problèmes ? On attend une représentation graphique mettant en évidence les inputs et outputs de chaque SP. Pour vous aider, vous pouvez appliquer le tri par insertion sur un exemple.

3.2.3 Exercices Complets

Dans les énoncés qui suivent, il ne vous est pas demandé de rédiger le code du programme. Vous devez vous limiter à l'Étape 2 de la méthodologie.

Pour chacun des exercices, vous devez réaliser l'Analyse complète (identification des SPs, Définition de chaque SP) et la structuration des SPs (représentation textuelle et graphique). Pensez à vous aider de la fiche de validation (Tableau 3.2 et Annexe 2).

▷ **Exercice 1** Repartez des trois exercices de la Sec. 3.1.3 et complétez chaque Définition par l'Analyse complète.

3.3 Étape 3 : Construction par Invariant Graphique

3.3.1 Invariant Graphique

3.3.1.1 Fiche de Validation

La fiche de validation pour l'Étape 3 est fournie par les tableaux 3.3 et 3.4. Une case doit être cochée si l'item est présent. Une explication textuelle peut, éventuellement, être ajoutée. Plusieurs instances de cette fiche sont disponibles à l'Annexe 3.

Commençons par trois critères généraux :

Critères généraux	**Cohérent ?**	**Commentaires (référés par des tags si besoin)**
Chaque SP avec une boucle a un Invariant Graphique correspondant	☐	
Références au SP dont on parle présentes	☐	
Invariants Graphiques présentés cohérents	☐	

TABLE 3.3 – Critères Généraux pour l'étape 3 (Invariant Graphique).

En particulier, pour l'Invariant Graphique correspondant au Sp ..... :

Règles de l'Invariant Graphique	**Cohérent ?**	**Commentaires (référés par des tags si besoin)**
Le nom de la structure est approprié (Règle 1)	☐	
Le dessin est approprié (Règle 1)	☐	
Borne minimale (Règle 2)	☐	
Borne maximale (Règle 2)	☐	
Ligne de démarcation (Règle 3)	☐	
Variable d'itération (Règle 4)	☐	
La caractérisation des variables et la description de ce qui a été réalisé jusqu'à présent (Règle 5) sont :		
- cohérentes	☐	
- complètes[1]	☐	
- précises	☐	
Zone "à faire" (Règle 6)	☐	
Propriétés qui sont conservées (Règles 5+6)	☐	

[1] Toutes les propriétés liées à la zone parcourue, les variables intermédiaires et les actions réalisées sont exprimées.

Table 3.4 – Fiche de validation pour l'étape 3 (Invariant Graphique).

3.3.1.2 Compléter un Invariant Graphique

Dans les exercices qui suivent, nous vous fournissions des énoncés de problèmes et les Invariants Graphiques associés. Les Invariants Graphiques, à chaque fois, sont **incomplets**. Il vous est demandé de les compléter afin d'obtenir un Invariant Graphique valide (il n'est pas demandé de produire le code du programme).

Pour résoudre ces exercices, nous vous conseillons de :

1. définir (Input(s), Output, Caractérisation des Inputs) complètement le problème ;
2. fournir une représentation graphique de l'Output (cfr. Chapitre 3, Slides 86 → 94) en vous appuyant éventuellement sur l'Invariant Graphique incomplet ;
3. compléter l'Invariant Graphique par éclatement de la PostCondition ;
4. vérifier votre Invariant Graphique en fonction des règles de conception.

N'hésitez pas à vous aider avec l'outil Glide.

▷ **Exercice 1** L'utilisateur entre un nombre entier positif au clavier et le programme, à la fin, affiche la somme des entiers positifs jusqu'à ce nombre. Complétez la Fig. 3.1 afin de produire un Invariant Graphique permettant de résoudre ce problème.

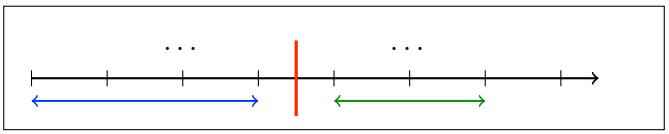


Figure 3.1 – Invariant Graphique partiel pour le calcul de la somme des entiers positifs.

▷ **Exercice 2** L'utilisateur souhaite afficher sur la sortie standard une ligne composée de n caractères. Le caractère à afficher et la longueur de la ligne sont lues au clavier. Complétez la Fig. 3.2 afin de produire un Invariant Graphique permettant de résoudre ce problème.

Figure 3.2 – Invariant Graphique partiel pour l'affichage d'une ligne de n caractères.

3.3.2 Invariant Graphique et Zone 1

Dans les exercices qui suivent, nous vous fournissions des énoncés de problèmes et les Invariants Graphiques associés. Les Invariants Graphiques, à chaque fois, sont **complets**. Il vous est demandé de dériver, des Invariants Graphiques, les variables dont on a besoin, leurs types, ainsi que les valeurs d'intialisation (i.e., Zone 1). Vous veillerez à justifier proprement vos choix sur base des Invariants Graphiques.

▷ **Exercice 1** L'utilisateur demande d'afficher à l'écran le résultat d'un exposant négatif, la base et l'exposant étant lus au clavier. L'Invariant Graphique pour résoudre ce problème est donné à la Fig. 3.3.

Indiquez les variables et leurs types ainsi que les valeurs d'intialisation (Zone 1). Justifiez sur base de l'Invariant Graphique. Il n'est pas demandé d'écrire le code C des Zones 2 et 3.

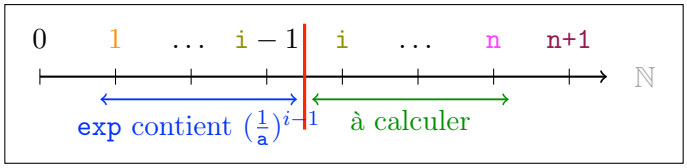


FIGURE 3.3 – Invariant Graphique pour le calcul d'un exposant négatif.

Rappel préliminaire sur les puissances :

$$a^{-n} = \left(\frac{1}{a}\right)^n$$

Par exemple :

$$3^{-2} = \left(\frac{1}{3}\right)^2$$
$$\left(\frac{2}{3}\right)^{-4} = \left(\frac{3}{2}\right)^4$$

▷ **Exercice 2** L'utilisateur désire calculer la somme des entiers dans l'intervalle $[a, b[$, avec a et b lus au clavier. L'Invariant Graphique pour résoudre ce problème est donné à la Fig. 3.4.

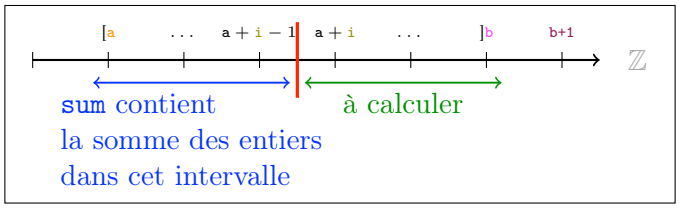


FIGURE 3.4 – Invariant Graphique pour le calcul de la somme des entiers dans un intervalle.

Indiquez les variables et leurs types ainsi que les valeurs d'intialisation (ZONE 1). Justifiez sur base de l'Invariant Graphique. Il n'est pas demandé d'écrire le code C des ZONES 2 et 3.

3.3.3 Invariant Graphique, ZONE 1 et Critère d'Arrêt

Dans les exercices qui suivent, nous vous fournissions des énoncés de problèmes et les Invariants Graphiques associés. Les Invariants Graphiques, à chaque fois, sont **complets**. Il vous est demandé de dériver, des Invariants Graphiques, les variables dont on a besoin, leurs types, les valeurs d'intialisation (i.e., ZONE *1*) ainsi que le Critère d'Arrêt de la boucle. Vous penserez aussi à dériver du Critère d'Arrêt le Gardien de Boucle. Vous veillerez à justifier proprement vos choix sur base des Invariants Graphiques.

▷ **Exercice 1** L'utilisateur désire afficher à l'écran la suite de caractères suivante :

```
+----------------------+
```

Il s'agit donc d'une suite de `'-'` entourée (à gauche et à droite) par le caractère `'+'`. La longueur de la ligne, n, correspond au nombre de `'-'` sur la ligne. n est introduit au clavier par l'utilisateur. L'Invariant Graphique pour résoudre ce problème est donné à la Fig. 3.5.

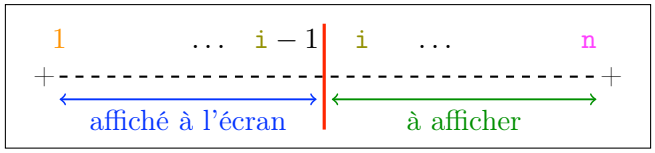


FIGURE 3.5 – Invariant Graphique pour l'affichage d'une ligne faite de '-' et '+'.

Indiquez les variables et leurs types ainsi que les valeurs d'intialisation (ZONE 1). Dérivez aussi le Critère d'Arrêt de la boucle ainsi que le Gardien de Boucle. Justifiez sur base de l'Invariant Graphique. Il n'est pas demandé d'écrire le code C des ZONES 2 et 3.

▷ **Exercice 2** La banque "MegaPognon" accepte de vous accorder un prêt si la somme de vos intérêts dépasse 1000€. Le taux d'intérêt est de 3.5% par an. Il vous est demandé d'écrire un programme permettant de déterminer, sur base d'un montant placé initialement, le nombre d'années à attendre pour pouvoir bénéficier d'un prêt. Voici un exemple d'output de votre programme :

```
Somme initiale placée: 2000 euros
1ère année: intérêt = (2000 * 3.5)/100 = 70
2ème année: intérêt = (2070 * 3.5)/100 = 72.45
...
```

L'Invariant Graphique pour résoudre ce problème est donné à la Fig. 3.6.

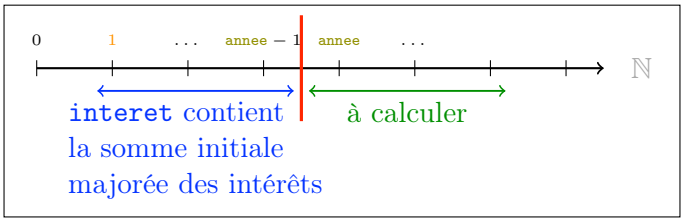


FIGURE 3.6 – Invariant Graphique pour le calcul d'intérêts.

Indiquez les variables et leurs types ainsi que les valeurs d'intialisation (ZONE 1). Dérivez aussi le Critère d'Arrêt de la boucle ainsi que le Gardien de Boucle. Justifiez sur base de l'Invariant Graphique. Il n'est pas demandé d'écrire le code C des ZONES 2 et 3.

3.3.4 Invariant Graphique et ZONE 2

Dans les exercices qui suivent, nous vous fournissions des énoncés de problèmes, les Invariants Graphiques associés et une partie du code (à savoir la ZONE 1 et la Gardien de Boucle). Les Invariants Graphiques, à chaque fois, sont **complets**. Il vous est demandé de dériver, des Invariants Graphiques, la ZONE 2. Vous veillerez à justifier proprement vos choix sur base des Invariants Graphiques.

▷ **Exercice 1** L'utilisateur lit un entier positif au clavier, n, et désire afficher à l'écran le nombre de 0 qui compose n dans sa représentation binaire. L'Invariant Graphique pour résoudre ce problème est donné à la Fig. 3.7.

Complétez l'extrait de code 3.1 en remplissant le corps de la boucle (i.e., ZONE 2). Justifiez sur base de l'Invariant Graphique.

Extrait de code 3.1 – Code à compléter pour le calcul du nombre de 0 dans la représentation binaire de n.

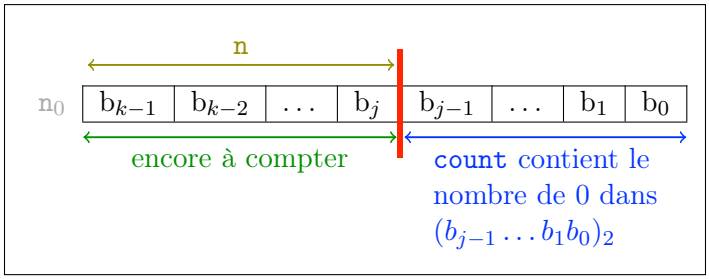


FIGURE 3.7 – Invariant Graphique pour le calcul du nombre de 0 dans la représentation binaire de n.

```c
#include <stdio.h>

int main(){
  unsigned int n, count;

  scanf("%u", &n);

  count = 0;

  while(n > 0){
    //ZONE 2 -- VOTRE CODE ICI
  }//fin while

  printf("Nombre de 0 dans %u: %u\n", n, count);
}//fin programme
```

▷ **Exercice 2** L'utilisateur entre un nombre entier positif, n, au clavier et désire obtenir, à l'écran, la liste des facteurs de n.

> Rappel préliminaire sur les facteurs.
> Le *facteur* de tout nombre est un nombre entier qui divise exactement le nombre en un nombre entier sans laisser de reste. Par exemple, 2 est un facteur de 6 car 2 divise 6 exactement sans laisser de reste.

Voici un exemple d'éxécution du programme :

```
saisir un nombre: 20
les facteurs de 20 sont: 1 2 4 5 10 20
```

L'Invariant Graphique pour résoudre ce problème est donné à la Fig. 3.8.

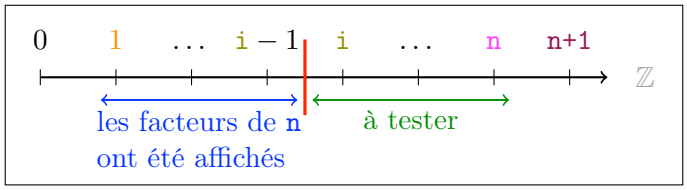


FIGURE 3.8 – Invariant Graphique pour le calcul des facteurs de n.

Complétez l'extrait de code 3.1 en remplissant le corps de la boucle (i.e., ZONE 2). Justifiez sur base de l'Invariant Graphique.

Extrait de code 3.2 – Code à compléer pour le calcul des facteurs de n.

```c
#include <stdio.h>

int main(){
  unsigned int n, i, facteur;

  printf("saisir un nombre");
  scanf("%u", &n);

  i = 1;
  printf("les facteurs de %u sont: ", n);

  while(i <= n){
    //Zone 2 -- VOTRE CODE ICI
  }//fin while

  printf("\n");
}//fin programme
```

3.3.5 Exercices Complets

▷ **Exercice 1** Repartez des trois exercices de la Sec. 3.1.3 et complétez chaque Sp par un Invariant Graphique (quand cela s'avère pertinent). Rédigez ensuite le code pour chaque Sp. Quand cela s'avère pertinent, construisez votre code (Zone 1, Critère d'Arrêt, Zone 2, Zone 3 et Fonction de Terminaison) et justifiez chaque étape sur base du Invariant Graphique. Si nécessaire, appuyez vous sur le Glide pour construire et manipuler vos Invariants Graphiques.

Chapitre 4

Structures de Données

Ce chapitre se concentre sur la notion de structures de données et les algorithmes qui tournent autour. Jusqu'à présent, les "objets" manipulés dans nos programmes étaient simples, e.g., `int`, `char`, ou encore `double`. Nous allons aborder maintenant des "objets" plus complexes que nous devons construire nous-même (i.e., ils ne sont pas prédéfinis par le langage). Tout d'abord, les exercices porteront sur une des structures de données les plus courantes, les tableaux unidimensionnels (Sec. 4.1). Nous passerons ensuite à des variations des tableaux unidimensionnels, soit les tableaux multidimensionnels (Sec. 4.2) et les chaînes de caractères (Sec. 4.3). Nous nous exercerons ensuite à manipuler des structures de données qui permettent d'avoir, sous un même nom, plusieurs variables de types différents, les enregistrements (Sec. 4.4). Nous verrons ensuite les énumérations (Sec. 4.5) et les fichiers (Sec. 4.6). Enfin, nous finirons ce chapitre avec un exercice de modélisation d'un problème via des structures de données (Sec. 4.7).

4.1 Tableaux Unidimensionnels

Pour chacun des exercices de cette section, il vous est demandé de suivre avec rigueur la méthologie vue au cours. En particulier, la réponse à chacun des exercices devra comprendre (explicitement) les étapes suivantes :

1. Définition du problème (i.e., Input(s), Output, Caractérisation des Inputs) ;
2. Analyse du problème (i.e., découpe en SP– pensez à indiquer des noms pertinents pour chaque SP– et représentation graphique de l'enchaînement des SPS) ;
3. Définition de chaque SP (i.e., Input(s), Output, Caractérisation des Inputs) ;
4. pour chaque SP nécessitant une boucle, représentez graphiquement l'Output ;
5. Dérivez l'Invariant Graphique de la représentation graphique de l'Output (éclatement de la POSTCONDITION). Confrontez votre Invariant Graphique avec les règles de conception. N'hésitez pas à vous appuyer sur le GLIDE pour la construction et la validation automatique de vos Invariants Graphiques ;
6. Écriture du code C.

Attention, pour le code, il vous est demandé de :

- déterminer les variables nécessaires et leurs valeurs d'initialisation (i.e., ZONE 1). Justifiez sur base de l'Invariant Graphique ;
- déterminer le Critère d'Arrêt. Jusitifiez sur base de l'Invariant Graphique ;
- déterminer la Fonction de Terminaison. Justifier sur base de l'Invariant Graphique ;
- déterminer le code du corps de la boucle (i.e., ZONE 2). Justifiez sur base de l'Invariant Graphique ;

- déterminer le code après la boucle (i.e., ZONE 3). Justifiez sur base de l'Invariant Graphique et vérifiez bien que vous atteignez l'Output de votre SP.

Faites valider chacune de ces étapes par un membre de l'équipe pédagogique avant de passer à l'étape suivante.

4.1.1 Manipulations Simples

Soit le code suivant servant de base à plusieurs exercices :

Extrait de code 4.1 – Code de base pour les exercices de manipulation des vecteurs.

```
1  #include <stdio.h>
2
3  int main(){
4     //Constante qui va définir la taille du tableau
5     const unsigned int N = ...;
6
7     //tableau d'entiers que nous allons manipuler
8     int tab[N];
9
10    /*
11     * Code de remplissage du tableau.
12     *
13     * Invariant Graphique:
14     *
15     *        |0          |i          N-1|N
16     *        +-----------+-------------+
17     * tab:   |           |             |
18     *        +-----------+-------------+
19     *         <---------> <------------>
20     *           rempli        encore
21     *          au clavier    à remplir
22     *
23     * Fonction de terminaison: N-i
24     */
25    unsigned int i = 0;
26    while(i < N){
27       printf("Introduisez une valeur: ");
28       scanf("%d", &tab[i]);
29       i++;
30    }//fin while -- i
31
32    //CODE À COMPLÉTER
33
34 }//fin programme
```

▷ **Exercice 1** Complétez le code de base (cfr. Listing 4.1, ligne 32) de façon à calculer et afficher la somme des éléments du tableau.

▷ **Exercice 2** Complétez le code de base (cfr. Listing 4.1, ligne 32) de façon à trouver le maximum et le minimum du tableau. Attention, vous ne pouvez utiliser qu'une seule boucle pour la recherche du maximum et du minimum.

▷ **Exercice 3** Écrivez un programme qui calcule le produit scalaire de deux vecteurs d'entiers U et V (de mêmes dimensions, `N`). Les valeurs contenues dans U et V ont déjà été introduites au clavier par l'utilisateur (code sensiblement identique au Listing 4.1).

Pour rappel, dans un repère orthonormé, le produit scalaire de deux vecteurs se calcule comme suit : Soient $u = (x, y)$ et $v = (x', y')$, le produit scalaire de u et v se calcule comme

$$\vec{u} \times \vec{v} = x \times x' + y \times y'.$$

▷ **Exercice 4** Calculez, pour une valeur X donnée (de type `float`), la valeur numérique d'un polynôme de degré n :

$$P(X) = A_n X^n + A_{n-1} X^{n-1} + \ldots + A_1 X^1 + A_0 X^0.$$

Les valeurs des coefficients $A_n, A_{n-1}, \ldots, A_1, A_0$ ont déjà été entrées au clavier et mémorisées dans un tableau `A` de type `float` et de dimension $n+1$ (code sensiblement identique au Listing 4.1).

Pour éviter les opération d'exponentiation, vous veillerez à utiliser le schéma de Horner :

$$\underbrace{A_n}\underbrace{\times X + A_{n-1}}\underbrace{\ldots}X + A_0$$

.

Pour rendre les choses plus simples, appliquons le schéma de Horner sur l'exemple suivant. Soit la fonction polynôme P définie par

$$P(x) = 2x^3 - 7x^2 + 4x - 1.$$

On souhaite calculer $P(a)$ pour $a = 5$. Le calcul de $P(5)$ nécessite 6 multiplications et 3 « additions-soustractions ». Soit

$$P(5) = 2\times5\times5\times5 - 7\times5\times5 + 4\times5 - 1 = 94.$$

Si on utilise l'écriture suivante :

$$P(x) = ((2x - 7)x + 4)x - 1.$$

alors le calcul de l'image de 5 ne nécessite plus que 3 multiplications et 3 « additions-soustractions ». Soit

$$P(5) = ((2\times5 - 7)\times5 + 4)\times5 - 1.$$

▷ **Exercice 5** Un tableau A de dimension $N+1$ contient N valeurs entières triées par ordre croissant, la $N+1^{\text{e}}$ valeur étant indéfinie. Écrivez un programme permettant d'insérer une valeur `x` donnée au clavier dans le tableau A de manière à obtenir un tableau de $N+1$ valeurs triées (le tableau A a déjà été pré-rempli avec les valeurs triées par ordre croissant – code sensiblement identique au Listing 4.1).

▷ **Exercice 6** Le crible d'Ératosthène permet de déterminer les nombres premiers inférieurs à une certaine valeur N. On place dans un tableau unidimensionnel `tab` les nombres entiers compris entre 1 et N. L'algorithme consiste, pour chaque élément `tab[i]` du tableau, à rechercher parmi tous les suivants (d'indices $i+1$ à N), ceux qui sont des multiples de `tab[i]` et à les éliminer (en les remplaçant, par exemple, par 0). Lorsque l'ensemble du tableau a subi ce traitement, seuls les nombres premiers du tableau n'ont pas été éliminés.

Écrivez un programme permettant de trouver, grâce au crible d'Ératosthène, les nombres premiers inférieurs à 500.

▷ **Exercice 7** Écrivez un programme qui lit au clavier les points de N élèves d'une classe pour un devoir et les mémorise dans un tableau à N éléments. Les notes saisies sont comprises entre 0 et 60.

Une fois les notes obtenues, il est demandé de rechercher et d'afficher :

- la note maximale
- la note minimale
- la moyenne des notes

▷ **Exercice 8** Cet exercice complète l'exercice précédent. Une fois le code de l'Exercice 7 écrit, il est demandé de le modifier comme suit : à partir des points des élèves, créer un tableau (appelons le `notes`) de dimension 7 qui est composé de la façon suivante :

`notes[6]` contient le nombre de notes à 60.

`notes[5]` contient le nombre de notes comprises entre 50 et 59.

`notes[4]` contient le nombre de notes comprises entre 40 et 49.

...

`notes[0]` contient le nombre de notes comprises entre 0 et 9.

Votre programme va devoir afficher à l'écran un graphique de barreaux (i.e., un *histogramme*) représentant le tableau `notes`. Utilisez les symboles ####### pour la représentation des barreaux et affichez le domaine des notes en dessous du graphique.

Voici un exemple de ce que votre programme doit afficher :

```
La note maximale est 58
La note minimale est 13
La moyenne des notes est 37.250000


6  >                              #######
5  >                      ####### #######
4  >                ####### ####### #######
3  >                ####### ####### ####### #######
2  >         ####### ####### ####### ####### #######
1  >         ####### ####### ####### ####### #######
    +-------+-------+-------+-------+-------+-------+-------+
    I 0 - 9 I 10-19 I 20-29 I 30-39 I 40-49 I 50-59 I  60   I
```

Pour cet exercice, il est plus qu'impératif de commencer par réfléchir au problème en l'analysant et en identifiant les divers Sp. Prenez le temps de bien réfléchir avant de commencer à coder.

4.1.2 Algorithmique

▷ **Exercice 1** La *distance de Hamming* entre deux tableaux de même taille est le nombre de position où les deux tableaux diffèrent. Par exemple, la distance de Hamming entre $\boxed{0}\boxed{1}\boxed{0}\boxed{0}$ et $\boxed{1}\boxed{1}\boxed{0}\boxed{1}$ égale 2.

Écrivez un programme qui calcule la distance de Hamming entre deux tableaux de même taille (on supposera que les tableaux ont déjà été remplis par l'utilisateur – code sensiblement identique au Listing 4.1).

▷ **Exercice 2** Écrivez un programme qui renverse l'ordre des éléments d'un tableau (au préalablement rempli) dans un nouveau tableau. Affichez, ensuite, à l'écran le contenu du deuxième tableau.

Par exemple, le tableau $\boxed{5}\boxed{1}\boxed{6}\boxed{2}\boxed{7}$ devient $\boxed{7}\boxed{2}\boxed{6}\boxed{1}\boxed{5}$. L'affichage à l'écran donnera :

```
[ 7 2 6 1 5 ]
```

▷ **Exercice 3** Écrivez un programme qui affiche, à l'écran, le nombre d'occurrences de chaque élément d'un tableau préalablement rempli au clavier (complétez le code de base, cfr. Listing 4.1, ligne 32). La seule hypothèse qu'on puisse faire sur le tableau, c'est qu'il ne peut contenir que des entiers strictement positifs.

▷ **Exercice 4** Même question que l'Exercice 2 mais sans passer, cette fois, par un deuxième tableau. Il faut donc faire le renversement directement dans le tableau initial. Comparez la complexité théorique (notation O) obtenue dans cette exercice avec la complexité théorique de votre code à l'Exercice 2. Quelle est la version la plus performante ?

4.2 Tableaux Multidimensionnels

4.2.1 Compléter des Invariants Graphiques

Dans les exercices qui suivent, nous vous fournissions un énoncé et les Invariants Graphiques associés. Les Invariants Graphiques, à chaque fois, sont **incomplets**. Il vous est demandé de les compléter (cfr. Chapitre 5, Slides 88 → 91).

▷ **Exercice 1** On dispose d'une matrice (i.e, tableau à deux dimensions) `mat` de dimensions $N \times M$ (i.e., N lignes et M colonnes, où N et M peuvent être différents). La matrice `mat` a déjà été remplie par l'utilisateur avec des valeurs entières. Il est demandé d'afficher, à l'écran, le contenu de `mat`.

Cet exercice ne se concentre que sur les Invariants Graphiques. Inutile, donc, de produire du code C. Afin de faciliter le travail, nous vous fournissons la définition du problème, l'analyse et des Invariants Graphiques incomplets. L'objectif de l'exercice est donc d'illustrer l'approche globale et faire en sorte que vous appréhendiez au mieux les Invariants Graphiques pour les tableaux multidimensionnels.

Pour résoudre cet exercice, nous vous conseillons de :

1. définir (Input, Output, Caractérisation des Inputs) complètement chaque Sp ;
2. fournir une représentation graphique de l'Output (cfr. Chapitre 3, Slides 69 → 76) en vous appuyant éventuellement sur l'Invariant Graphique incomplet ;
3. compléter l'Invariant Graphique par éclatement de la PostCondition ;
4. vérifier votre Invariant Graphique en fonction des règles de conception.

Définition du Problème

Input : la matrice `mat`, de dimensions $N \times M$, préalablement remplie par l'utilisateur.

Output : le contenu de la matrice `mat` est affiché à l'écran

Caractérisation des Inputs :

`N`, le nombre de lignes de la matrice (`const unsigned int N = ...;`)[1]

`M`, le nombre de colonnes de la matrice (`const unsigned int M = ...;`)

`mat`, une matrice d'entiers (`int mat[N][M];`)

Analyse du Problème On peut envisager deux Sp, comme illustré à la Fig. 4.1.

Sp_1 : Enumération et affichage des lignes de la matrice.

Sp_2 : Affichage des colonnes de la ligne courante de la matrice.

1. Peu importe la valeur de N.

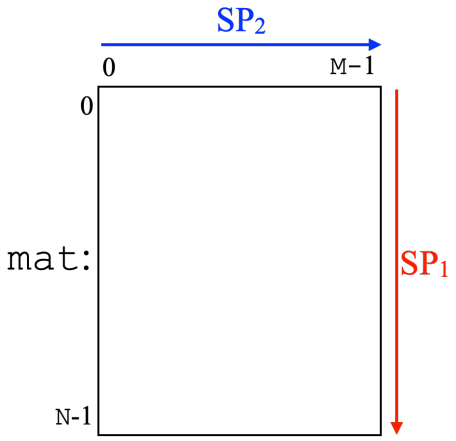


FIGURE 4.1 – Illustration des SPs pour l'affichage d'une matrice.

On voit clairement que le SP_2 est appliqué autant de fois qu'il y a de lignes dans la matrice. L'enchaînement des SP est donc une inclusion : $SP_2 \subset SP_1$.

Quoiqu'il en soit, les deux SP requièrent un traitement itératif et donc des Invariants Graphiques.

Invariant Graphique incomplet pour le SP_1 . Il s'agit ici de s'assurer de l'énumération et de l'affichage de toutes les lignes de la matrice. Complétez la Fig. 4.2 afin de produire un Invariant Graphique permettant de résoudre ce problème.

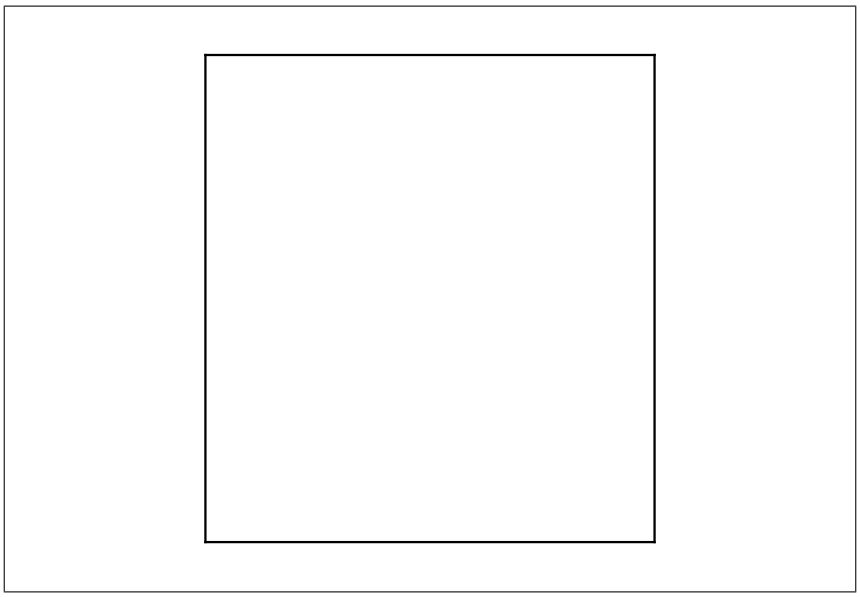


FIGURE 4.2 – Invariant Graphique partiel pour le SP_1.

Invariant Graphique incomplet pour le SP_2 . Il s'agit ici de s'assurer de l'affichage de toutes les colonnes d'une ligne de la matrice, ligne fournie par le SP_1. Complétez la Fig. 4.3 afin de produire un Invariant Graphique permettant de résoudre ce problème.

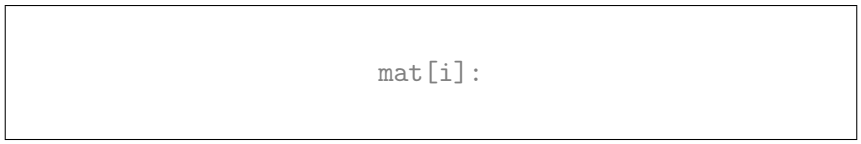


FIGURE 4.3 – Invariant Graphique partiel pour le SP_2.

4.2.2 Construction par Invariant de Boucle

Pour chacun des exercices de cette section, il vous est demandé de suivre avec rigueur la méthologie vue au cours. En particulier, la réponse à chacun des exercices devra comprendre (explicitement) les étapes suivantes :

1. définition du problème (i.e., Input, Output, Caractérisation des Inputs) ;
2. analyse du problème (i.e., découpe en SP– pensez à indiquer des noms pertinents pour chaque SP– et emboîtement des SP) ;
3. définition de chaque SP (i.e., Input, Output, Caractérisation des Inputs) ;
4. pour chaque SP nécessitant une boucle, représentez graphiquement l'Output ;
5. dérivez l'Invariant Graphique de la représentation graphique de l'Output (éclatement de la POSTCONDITION). Confrontez votre Invariant Graphique avec les règles de conception. N'hésitez pas à vous appuyer sur le GLIDE pour la construction et la validation automatique de vos Invariants Graphiques ;
6. déterminez la Fonction de Terminaison ;
7. écriture code C.

Attention, pour le code, il vous est demandé de :

- déterminer les variables nécessaires et leurs valeurs d'initialisation (i.e., ZONE 1). Justifiez sur base de l'Invariant Graphique ;
- déterminer le Critère d'Arrêt. Jusitifiez sur base de l'Invariant Graphique ;
- déterminer la Fonction de Terminaison. Justifier sur base de l'Invariant Graphique.

Faites valider chacune de ces étapes par un membre de l'équipe pédagogique avant de passer à l'étape suivante.

▷ **Exercice 1** Écrivez un programme qui remplit un tableau de 12 lignes et 12 colonnes à l'aide des caractères '1', '2' et '3' tel que :

```
1
1 2
1 2 3
1 2 3 1
1 2 3 1 2
1 2 3 1 2 3
1 2 3 1 2 3 1
1 2 3 1 2 3 1 2
1 2 3 1 2 3 1 2 3
1 2 3 1 2 3 1 2 3 1
1 2 3 1 2 3 1 2 3 1 2
1 2 3 1 2 3 1 2 3 1 2 3
```

Rappel préliminaire sur les matrices [a]

On appelle matrice un ensemble de scalaires, i.e. des nombres réels ou complexes, ordonnés sous la forme d'un tableau rectangulaire de m lignes et n colonnes. On note la matrice :

$$A = \begin{pmatrix} a_{11} & a_{12} & \cdots & a_{1n} \\ a_{21} & a_{22} & \cdots & a_{2n} \\ \vdots & \vdots & & \vdots \\ a_{m1} & a_{m2} & \cdots & a_{mn} \end{pmatrix}$$

Les nombres a_{ij} sont les *éléments* de la matrice. On note les matrices en majuscule. Leurs éléments sont notés au moyen de la minuscule correspondante. Les nombres m et n sont les dimensions de la matrice. Si $m = n$, on parle de matrice *carrée*. La dimension $m = n$ est appelée *l'ordre* de la matrice.

Les éléments $a_{ii} (i = 1, \ldots, n)$ d'une matrice carrée forment la *diagonale principale* de la matrice.

Une matrice *identité* est une matrice dont les éléments sont tous nuls sauf ceux de la diagonale principale, qui valent 1, c'est-à-dire, telle que :

$$i_{jk} = \begin{cases} 1, si\ j = k \\ 0, si\ j \neq k \end{cases}$$

Par exemple, si $n = 3$:

$$I = \begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{pmatrix}$$

La *transposée* d'une matrice A de dimensions $(m \times n)$, est la matrice A^T de dimensions $(n \times m)$ dont les éléments sont donnés par :

$$(A^T)_{ij} = a_{ji}$$

Exemple, si $A = \begin{pmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \end{pmatrix}$, alors $A^T = \begin{pmatrix} 1 & 4 \\ 2 & 5 \\ 3 & 6 \end{pmatrix}$.

L'addition matricielle est définie comme suit : Si A et B sont deux matrices de dimension $N \times M$, alors la matrice C est obtenue via $\forall i \in 1, \ldots, N, \forall j \in 1, \ldots, M, c_{i,j} = a_{i,j} + b_{i,j}$.

Par exemple,

$$\begin{pmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \end{pmatrix} + \begin{pmatrix} 7 & 8 & 9 \\ 10 & 11 & 12 \end{pmatrix} = \begin{pmatrix} 8 & 10 & 12 \\ 14 & 16 & 18 \end{pmatrix}$$

a. d'après E.J.M.Delhez, *Algèbre*, Tome 1, Centrale des Cours.

▷ **Exercice 2** Écrivez un programme qui met à zéro les éléments de la diagonale d'une matrice carrée A donnée (i.e., remplie par l'utilisateur au clavier).

▷ **Exercice 3** Écrivez un programme qui construit et affiche une matrice *carrée identité* I de dimension N.

▷ **Exercice 4** Écrivez un programme qui effectue la transposition A^T d'une matrice A de dimensions $N \times M$ en une matrice B de dimension $M \times N$. La matrice A sera remplie au clavier par l'utilisateur.

▷ **Exercice 5** Écrivez un programme qui effectue l'addition de deux matrices A et B de même dimension $N \times M$. Les deux matrices A et B seront construites au clavier par l'utilisateur et le résultat de l'addition sera placé dans une matrice C avant d'être affiché à l'écran.

▷ **Exercice 6** Écrivez un programme qui recherche dans une matrice A donnée (i.e., in-

troduite au clavier par l'utilisateur) les éléments qui sont à la fois maximum sur leur ligne et minimum sur leur colonne. Ces éléments sont appelés des *points-cols*. Votre programme doit afficher les positions et les valeurs de tous les points-cols de votre matrice A.

Par exemple, pour la matrice $\begin{pmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \end{pmatrix}$ le résultat sera le suivant :

```
point-col: (0,2): 3
```

4.3 Chaines de Caractères

Pour chacun des exercices de cette section, il vous est demandé de suivre avec rigueur la méthologie vue au cours. En particulier, la réponse à chacun des exercices devra comprendre (explicitement) les étapes suivantes :

1. définition du problème (i.e., Input, Output, Caractérisation des Inputs ;
2. analyse du problème (i.e., découpe en SP– pensez à indiquer des noms pertinents pour chaque SP– et emboîtement des SP) ;
3. définition de chaque SP (i.e., Input, Output, Caractérisation des Inputs) ;
4. pour chaque SP nécessitant une boucle, représentez graphiquement l'Output ;
5. dérivez l'Invariant Graphique de la représentation graphique de l'Output (éclatement de la POSTCONDITION). Confrontez votre Invariant Graphique avec les règles de conception. N'hésitez pas à vous appuyer sur le GLIDE pour la construction et la validation automatique de vos Invariants Graphiques ;
6. déterminez la Fonction de Terminaison ;
7. écriture code C.

Attention, pour le code, il vous est demandé de :

- déterminer les variables nécessaires et leurs valeurs d'initialisation (i.e., ZONE 1). Justifiez sur base de l'Invariant Graphique ;
- déterminer le Critère d'Arrêt. Jusitifiez sur base de l'Invariant Graphique ;
- déterminer la Fonction de Terminaison. Justifier sur base de l'Invariant Graphique.

Faites valider chacune de ces étapes par un membre de l'équipe pédagogique avant de passer à l'étape suivante.

▷ **Exercice 1** Écrivez un programme déterminant le nombre de lettres "e" (minuscules) présentes dans un texte d'une seule ligne (supposée ne pas dépasser 132 caractères) fourni au clavier.

▷ **Exercice 2** Écrivez un programme qui lit au clavier un mot (d'au plus 30 caractères) et qui l'affiche à l'envers. *Attention : rien ne certifie que le mot contient exactement 30 caractères !*

▷ **Exercice 3** Écrivez un programme qui lit au clavier un mot (d'au plus 50 caractères) et qui détermine si le mot entré est un palindrome ou non.

Pour rappel, un palindrome est un texte dont l'ordre des mots reste le même qu'on le lise de gauche à droite ou de droite à gauche. Par exemple : "kayak", "eh ça va la vache" ou encore "Narine alla en Iran".

▷ **Exercice 4** Écrivez un programme qui lit des mots au clavier et les affiche après les avoir converti en *louchebem* (i.e., "langage des bouchers").

Cette conversion consiste à :

- reporter la première lettre du mot en fin de mot, suivie des lettres 'e' et 'm'.

- remplacer la première lettre du mot par la lettre 'l'.

Par exemple, "vision" devient "lisionvem", "vache" devient "lachevem" ou encore "bonne" devient "lonnebem".

Pour simplifier, les mots entrés par l'utilisateur font cinq caractères.

▷ **Exercice 5** Écrivez un programme permettant de saisir une phrase au clavier (maximum 300 caractères) et qui affiche, ensuite, le nombre de mots contenu dans la phrase.

▷ **Exercice 6** Écrivez un programme permettant de saisir, au clavier, une chaine de caractères (maximum 100 caractères) et une sous-chaine (maximum 5 caractères) et qui indique, ensuite, combien de fois la sous-chaine est présente dans la chaine de caractères principale.

4.4 Enregistrement

Pour chacun des exercices de cette section (du moins, quand cela s'avère nécessaire), il vous est demandé de suivre avec rigueur la méthologie vue au cours. En particulier, la réponse à chacun des exercices devra comprendre (explicitement) les étapes suivantes :

1. définition du problème (i.e., Input, Output, Caractérisation des Inputs ;
2. analyse du problème (i.e., découpe en SP– pensez à indiquer des noms pertinents pour chaque SP– et emboîtement des SP) ;
3. définition de chaque SP (i.e., Input, Output, Caractérisation des Inputs(s)) ;
4. pour chaque SP nécessitant une boucle, représentez graphiquement l'Output ;
5. dérivez l'Invariant Graphique de la représentation graphique de l'Output (éclatement de la POSTCONDITION). Confrontez votre Invariant Graphique avec les règles de conception. N'hésitez pas à vous appuyer sur le GLIDE pour la construction et la validation automatique de vos Invariants Graphiques ;
6. déterminez la Fonction de Terminaison ;
7. écriture code C.

Attention, pour le code, il vous est demandé de :

- déterminer les variables nécessaires et leurs valeurs d'initialisation (i.e., ZONE 1). Justifiez sur base de l'Invariant Graphique ;
- déterminer le Critère d'Arrêt. Jusitifiez sur base de l'Invariant Graphique ;
- déterminer la Fonction de Terminaison. Justifier sur base de l'Invariant Graphique.

Faites valider chacune de ces étapes par un membre de l'équipe pédagogique avant de passer à l'étape suivante.

▷ **Exercice 1** À l'aide d'une structure, écrivez la déclaration d'un type permettant de représenter une *durée* avec trois champs indiquant (respectivement) les heures, les minutes et les secondes. Soyez très précis sur les types utilisés.

▷ **Exercice 2** À l'aide d'une structure, écrivez la déclaration d'un type pour représenter une *personne*. Une personne sera représentée par son nom et son prénom (maximum 20 caractères dans les deux cas), son âge et son sexe ('M' ou 'F').

▷ **Exercice 3** Un grossiste en composants électroniques vend quatre types de produits :

- des cartes mères (code 1)
- des processeurs (code 2)
- des barrettes mémoire (code 3)
- des cartes graphiques (code 4)

Chaque produit possède une référence (i.e. son code, qui est un nombre entier), un prix en euros et des quantités disponibles.

Il est demandé de

1. définir type pour représenter un produit
2. d'écrire un programme qui initialise le stock de produits d'un magasin et qui permette ensuite à un utilisateur de saisir une commande d'un ou de plusieurs produit(s). L'utilisateur saisit les codes des produits et les quantités commandées. L'ordinateur affiche toutes les données de la commande, y compris le prix, ainsi que le prix total de la commande.

Par exemple, l'utilisateur pourrait utiliser le programme comme suit :

```
Que souhaitez-vous commander ?
(Carte mere = 1, Processeur = 2, Memoire = 3, Carte graphique = 4, Fini = 0)
Choix  : 2
Entrez une quantité : 200

Que souhaitez-vous commander ?
(Carte mere = 1, Processeur = 2, Memoire = 3, Carte graphique = 4, Fini = 0)
Choix  : 0

Voici le detail de votre commande :
Processeurs 200 x 100.00 = 20000.00 euros
TOTAL = 20000.00 euros
```

▷ **Exercice 4** Écrivez un programme qui lit au clavier des informations dans un tableau de structures de type `Point` défini comme suit :

```
typedef struct{
  int num;
  float x;
  float y;
}Point;
```

Le nombre d'éléments du tableau sera fixé au début du programme.

Ensuite, le programme affiche à l'écran l'ensemble des informations précédentes.

▷ **Exercice 5** Dans un premier temps, définissez un type permettant de représenter une fraction mathématique. Écrivez ensuite un programme qui place n (e.g., $n = 50$) fractions dans un tableau et les affiche ensuite à l'écran.

Dans un second temps, complétez votre programme pour qu'il affiche à l'écran toutes les fractions simplifiées.

▷ **Exercice 6** On souhaite construire une structure de données, appelée `EtatCivil`, permettant d'enregistrer le nom d'une personne, sa date de naissance, la ville où elle réside et le code postal de la ville. Définissez un type pour cette structure ainsi qu'un tableau qui permet de saisir ces dix ensembles d'informations et qui les affiche ensuite à l'écran.

▷ **Exercice 7** On souhaite enregistrer les notes de mathématique et de physique pour une classe de 35 élèves et calculer, pour chacun d'eux, la moyenne de ses notes. Proposez une structure de données pertinente et écrivez un programme permettant d'effectuer la saisie des notes puis l'affichage des résultats.

4.5 Énumérations

Pour chacun des exercices de cette section (du moins, quand cela s'avère nécessaire), il vous est demandé de suivre avec rigueur la méthologie vue au cours. En particulier, la réponse à chacun des exercices devra comprendre (explicitement) les étapes suivantes :

1. définition du problème (i.e., Input, Output, Caractérisation des Inputs(s) ;

2. analyse du problème (i.e., découpe en SP– pensez à indiquer des noms pertinents pour chaque SP– et emboîtement des SP) ;

3. définition de chaque SP (i.e., Input, Output, Caractérisation des Inputs(s) ;

4. pour chaque SP nécessitant une boucle, représentez graphiquement l'Output ;

5. dérivez l'Invariant Graphique de la représentation graphique de l'Output (éclatement de la POSTCONDITION). Confrontez votre Invariant Graphique avec les règles de conception. N'hésitez pas à vous appuyer sur le GLIDE pour la construction et la validation automatique de vos Invariants Graphiques ;

6. déterminez la Fonction de Terminaison ;

7. écriture code C.

Attention, pour le code, il vous est demandé de :

- déterminer les variables nécessaires et leurs valeurs d'initialisation (i.e., ZONE 1). Justifiez sur base de l'Invariant Graphique ;

- déterminer le Critère d'Arrêt. Jusitifiez sur base de l'Invariant Graphique ;

- déterminer la Fonction de Terminaison. Justifier sur base de l'Invariant Graphique.

Faites valider chacune de ces étapes par un membre de l'équipe pédagogique avant de passer à l'étape suivante.

▷ **Exercice 1** Écrivez une déclaration de type permettant de représenter un booléen. Pour rappel, un booléen est un type de variable à deux états : vrai ou faux. Attention, le type créé doit correspondre à la définition de "vrai" (et "faux") en C.

▷ **Exercice 2** Écrivez une déclaration de type permettant de représenter, sous la forme d'une énumération, les différents jours de la semaine

▷ **Exercice 3** Écrivez une déclaration de type permettant de représenter, sous la forme d'une énumération, les différents mois de l'année. Attention, janvier doit commencer à 1.

▷ **Exercice 4** On considère un jeu de cartes dans lequel les cartes sont de quatre couleurs possibles : "trefle", "pique", "carreau" et "coeur". Pour chacune des couleurs, une carte peut prendre les valeurs suivantes : "as", "roi", "dame", "valet", "dix", "neuf", "huit" et "sept". Proposez une (ou plusieurs) structure(s) de données pour représenter un tel jeu de cartes.

Ensuite, écrivez un programme qui crée un jeu de cartes et qui, ensuite,

- mélange les cartes (à vous de voir comment) ;

- affiche le jeu de cartes mélangé.

Comment contrôler l'aléatoire en C ?

En C, on peut produire une suite pseudo-aléatoire d'entiers en utilisant la fonction `rand()` de la bibliothèque standard (`stdlib.h`). A chaque fois qu'un programme appelle cette fonction, elle retourne l'entier suivant dans la suite. Les entiers de la suite sont compris entre `0` et `RAND_MAX` (une constante prédéfinie).

Par exemple, le programme suivant :

```c
#include <stdlib.h>
#include <stdio.h>

int main(){
  int i;

  for(i=0; i<10; i++){
    int a = rand();
    printf("%d\t", a);
  }//fin for - i

  printf("\n");

  return 0;
}//fin programme
```

produit la sortie suivante :

```
16807    282475249        1622650073       984943658        1144108930
470211272        101027544        1457850878       1458777923       2007237709
```

Si on cherche à tirer aléatoirement un nombre entre 0 et 2 (compris), il suffit d'utiliser le reste de la division par 3 :

```c
int coffre_tresor = rand()%3;
```

Si on veut tirer un entier entre A et B compris, on peut faire :

```c
int x = rand() % (B-A+1) + A;
```

Pour produire un flottant entre A et B, on peut faire :

```c
double x = (double) rand() / RAND_MAX * (B-A) + A ;
```

La suite produite par `rand()` n'est pas vraiment aléatoire. En particulier, si on lance le programme précédent plusieurs fois, on obtiendra exactement la même suite.

Pour rendre le résultat apparemment plus aléatoire, on peut initialiser la suite en appelant la fonction `srand()` (définie aussi dans `stdlib.h`), à laquelle on passe en paramètre une valeur d'initialisation. La suite produite par `rand()` après cet appel dépendra de la valeur que l'on passe à `srand()` lors de l'initialisation. Une même valeur d'initialisation produira la même suite.

Par exemple, le programme suivant :

```c
#include <stdlib.h>
#include <stdio.h>

int main(){
  int i;
  srand(2);

  for(i=0; i<5; i++){
    int a = rand();
    printf("%d\t", a);
  }//fin for - i

  printf("\n");
  srand(2);

  for(i=0; i<5; i++){
    int a = rand();
    printf("%d\t", a);
  }//end for - i

  printf("\n");

  return EXIT_SUCCESS;
}//fin programme
```

produit la sortie suivante :

```
33614   564950498       1097816499      1969887316      140734213
33614   564950498       1097816499      1969887316      140734213
```

Pour que la suite soit vraiment imprévisible, on peut passer à `srand()` une valeur dépendant de l'heure. Par exemple en utilisant la fonction `time()` de la librairie standard :

```c
srand(time(NULL));
```

4.6 Fichiers

Pour chacun des exercices de cette section (du moins, quand cela s'avère nécessaire), il vous est demandé de suivre avec rigueur la méthologie vue au cours. En particulier, la réponse à chacun des exercices devra comprendre (explicitement) les étapes suivantes :

1. définition du problème (i.e., Input, Output, Caractérisation des Inputs ;
2. analyse du problème (i.e., découpe en SP– pensez à indiquer des noms pertinents pour chaque SP– et emboîtement des SP) ;
3. définition de chaque SP (i.e., Input, Output, Caractérisation des Inputs ;
4. pour chaque SP nécessitant une boucle, représentez graphiquement l'Output ;
5. dérivez l'Invariant Graphique de la représentation graphique de l'Output (éclatement de la PostCondition). Confrontez votre Invariant Graphique avec les règles de conception. N'hésitez pas à vous appuyer sur le Glide pour la construction et la validation automatique de vos Invariants Graphiques ;
6. déterminez la Fonction de Terminaison ;
7. écriture code C.

Attention, pour le code, il vous est demandé de :

- déterminer les variables nécessaires et leurs valeurs d'initialisation (i.e., ZONE 1). Justifiez sur base de l'Invariant Graphique ;
- déterminer le Critère d'Arrêt. Jusitifiez sur base de l'Invariant Graphique ;
- déterminer la Fonction de Terminaison. Justifier sur base de l'Invariant Graphique.

Faites valider chacune de ces étapes par un membre de l'équipe pédagogique avant de passer à l'étape suivante.

▷ **Exercice 1** Écrivez un programme permettant d'afficher, à l'écran, le contenu d'un fichier en numérotant les lignes. Ces lignes ne devront jamais dépasser plus de 80 caractères.

▷ **Exercice 2** Écrivez un programme qui permet de créer séquentiellement un fichier "répertoire" comportant pour chaque personne

- nom (20 caractères maximum)
- prénom (15 caractères maximum)
- âge (entier)
- numéro de téléphone (11 caractères maximum)

Les informations relatives aux différentes personnes seront lues au clavier.

▷ **Exercice 3** Écrivez un programme permettant, à partir du fichier créé par l'exercice précédent, de retrouver les informations correspondant à une personne de nom donné (l'information sur le nom est donnée au clavier).

▷ **Exercice 4** Écrivez un programme permettant de saisir 20 mots de 20 caractères maximum et de les enregistrer ligne par ligne dans un fichier appelé `mots.txt`. Écrivez ensuite un autre programme qui permet de relire le fichier et qui en affiche le contenu à l'écran.

▷ **Exercice 5** On propose d'ajouter au programme écrit à l'Exercice 8 (Sec. 4.4) une séquence d'instructions permettant d'écrire les 35 enregistrements contenant les noms, les notes et moyennes de chaque élève dans un fichier appelé `scol.txt`.

Une fois ces modifications effectuées, écrivez un nouveau programme permettant, à partir de ce fichier `scol.txt` d'afficher les résultats scolaires de la classe.

4.7 Modélisation

▷ **Exercice 1** Soit une société qui doit gérer des informations sur son personnel. Elle emploie au maximum 100 personnes. Pour chaque employé, elle enregistre le nom, le prénom, l'adresse, le sexe et, s'il s'agit d'un homme, la situation militaire (libéré, exempté, réformé ou incorporable), s'il s'agit d'une femme, son nom de jeune fille. Une adresse contient un numéro, un nom de rue, un code postal et une localité. Un nom ne dépasse jamais 30 caractères.

Proposez les structures de données nécessaires pour représenter ces informations. Soyez très précis sur les types utilisés et les noms des différentes structures.

Chapitre 5

Modularité du Code

Jusqu'à présent, les programmes réalisés étaient constitués d'un seul et unique bloc de code principal. Cependant, il est possible de découper le code en divers *modules* pouvant être appelés n'importe quand dans le programme. Une programme C devient alors une succession de modules, le module `int main()` étant le principal. Les modules se présentent sous deux formes : *fonction* (i.e., un module permettant d'effectuer un travail et renvoyant un résultat au code appelant) et *procédure* (i.e., un module permettant d'effectuer un travail mais ne retournant pas de résultat au code appelant). Un exemple de fonction est la fonction `fopen()` que nous avons manipulée avec les fichiers. Un exemple de procédure est `printf()` que nous connaissons depuis le Chapitre 1.

L'avantage de cette découpe en modules, c'est qu'elle s'applique parfaitement à la découpe en Sp. Chaque Sp peut être représenté, maintenant, par un module. Et pour peu que le Sp soit un minimum générique, il pourra être réutilisé maintes fois durant la vie du programme.

Ce chapitre sera supporté par une seule séance de répétition, avec en supplément le Challenge 4 et la troisième session de Cdb. Une fois n'est pas coutume, la séance de répétition sera chronométrée et scénarisée de la façon suivante :

1. Rappel théorique – $10min$.
2. Exercices de lecture de code pour se familiariser avec les invocations de fonctions/procédures, le passage de paramètres et le retour de fonctions (Sec. 5.1) – $15min$.
3. Exercices sur la défintion d'interfaces, i.e., spécifications et signature des fonctions/procédures (Sec. 5.2) – $15min$.
4. Un exercice complet qui reprend tous les éléments du cours, à savoir la Définition du problème, l'Analyse et la découpe en Sps (qui seront, ici, implémentés sous la forme de fonctions/procédures) et rédaction du code sur base des Invariants Graphiques (Sec. 5.3). En fonction de l'organisation, exacte, il est possible que cet exercice soit réalisé via Café. Si ce n'est pas le cas, nous ramasserons toutes vos productions à la fin de l'exercie (pensez à mettre votre matricule ULiège sur vos feuilles). Un questionnaire vous sera aussi fourni en fin de séance pour récolter vos ressentis sur cet exercice – $1h$.

5.1 Lecture de Code

L'objectif de cette section est de vous apprendre à lire et à comprendre du code écrit par un autre programmeur. Soyez clair et précis dans vos réponses. Inutile d'exprimer les choses en paraphrasant le code C (i.e., *Si la variable* `i` *est plus petite que ...*), ce qui correspondrait à une traduction *littérale* du code. Pensez plutôt en terme de *transcodage* (ou traduction *littéraire*) en indiquant l'objectif final du bout de code.

La meilleure façon de résoudre les exercices qui suivent est de travailler avec des valeurs pour chacune des variables, exécuter le code avec ces valeurs et, ensuite, essayer d'inférer une relation entre les différentes variables. C'est cette relation qui vous donnera la réponse.

Pour rappel, vous disposez de 15*min.* pour résoudre les exercices de cette section.

▷ **Exercice 1** Soit le code suivant

```c
#include <stdio.h>

int fct(int r){
  return 2 * r;
}//fin fct()

int main(){
  int n, p = 5;

  n = fct(p);

  printf("p = %d, n = %d\n", p, n);

  return 0;
}//fin programme
```

Quel est le résultat, à l'écran, de ce programme ?

▷ **Exercice 2** Soit le code suivant :

```c
#include <stdio.h>
int n=10, q=2;

int fct(int p){
  int q;
  q = 2 * p + n;
  printf("B: dans fct, n = %d, p = %d, q = %d\n", n, p, q);

  return q;
}//fin fct()

void f(){
  int p = q * n;
  printf("C: dans f, n = %d, p = %d, q = %d\n", n, p, q);
}//fin f()

int main(){
  int n=0, p=5;

  n = fct(p);
  printf("A: dans main, n = %d, p = %d, q = %d", n, p, q);
  f();

  return 0;
}//fin programme
```

Quel est le résultat, à l'écran, de ce programme ?

5.2 Définition d'Interfaces

Pour les exercices qui suivent, il n'est pas demandé d'écrire le code C des modules mais bien de produire l'interface des fonctions/procédurdes (i.e., spécifications et prototype).

Pour rappel, vous disposez de 15*min.* pour résoudre les exercices de cette section.

▷ **Exercice 1** Proposez une interface pour un module qui calcule l'arc tangente d'un angle orienté.

▷ **Exercice 2** Proposez une interface pour un module qui calcule la moyenne des valeurs d'un tableau.

▷ **Exercice 3** Proposez une interface pour un module qui calcule la variance des valeurs d'un tableau.

▷ **Exercice 4** Proposez une interface pour les modules permettant de solutionner les problèmes suivants. Il est clair que, dans ces énoncés, subsistent des ambigüités, des imprécisions. Il vous appartient justement de les éliminer.

1. Rechercher l'emplacement d'une valeur donnée dans un tableau d'entiers.
2. Rechercher l'emplacement d'une valeur donnée dans une portion d'un tableau d'entiers.
3. Calculer le nombre d'éléments communs à deux tableaux d'entiers.
4. Tester si deux tableaux d'entiers comprennent les mêmes valeurs
5. Réaliser une copie inversée d'un préfixe d'un tableau d'entiers
6. Trouver l'entier apparaissant le plus souvent dans un tableau d'entiers

▷ **Exercice 5** En programmation défensive, comment feriez-vous pour écrire les assertions permettant de vérifier la PréCondition du module suivant :

```
/*
 * @pre: a>0, b>2*a, b est pair
 * @post: ...
 */
void ma_fonction(int a, int b);
```

5.3 Exercice Complet

Dans cet exercice, vous allez devoir implémenter un programme permettant d'afficher, sur la sortie standard, différentes formes géométriques dessinées à l'aide des caractères `'*'`, `'-'`, `'+'`, `'|'` et `'␣'` (espace). Ce sera l'utilisateur qui choisit la forme géométrique à dessiner. L'utilisateur peut demander autant de formes qu'il le désire.

Rectangle . Il s'agit de dessiner sur la srotie standard un rectangle à l'aide des caractères `'+'` (pour les angles), `'-'` (pour la longueur), `'|'` pour la largeur et espace (`'␣'` – pour l'intérieur du rectangle). La longeur s'exprime en terme de nombre de caractères `'-'` et est donnée au clavier. La largeur s'exprimer en terme de nombre de caractères `'|'` et est donnée au clavier.

Pour simplifier, vous pouvez considérer que la longueur et la largeur seront toujours plus grandes ou égales à 1.

Un exemple d'exécution pour une longueur de 23 et une largeur de 3 est donnée çi-dessous.

```
+-----------------------+
|                       |
|                       |
|                       |
+-----------------------+
```

Triangle (orienté vers le haut) . Il s'agit de dessiner un triangle *isocèle* sur la sortie standard en utilisant les caractères `'*'` et espace (`'␣'`).

Le triangle dessiné doit être tel que $BC = 2 \times n - 1$ et tel que la hauteur A soit égale à n. La valeur de n est introduite au clavier par l'utilisateur.

Attention, le triangle devra être orienté vers le haut (i.e., la pointe vers le haut de l'écran).

Un exemple d'exécution pour $n = 6$ est donné çi-dessous.

```
     *
    ***
   *****
  *******
 *********
***********
```

Triangle (orienté vers le bas) . Il s'agit de dessiner un triangle *isocèle* sur la sortie standard en utilisant les caractères `'*'`. Un triangle est dit isocèle s'il a deux côtés de même longueur, deux angles de même mesure et un axe de symmétrie.

Le triangle dessiné doit être tel que $BC = 2 \times n - 1$ et tel que la hauteur A soit égale à n. La valeur de n est introduite au clavier par l'utilisateur. Un exemple d'exécution pour $n = 6$ est donné çi-dessous.

Attention, le triangle devra être orienté vers le bas (i.e., la pointe vers le bas de l'écran). Un exemple d'exécution pour $n = 5$ est donné çi-dessous.

```
***********
 *********
  *******
   *****
    ***
     *
```

Trapèze . Il s'agit de dessiner des *trapèzes isocèles* sur la sortie standard en utilisant les caractères `'*'` et `'-'`. Pour rappel, un trapèze est dit isocèle si les deux bases du trapèze (*petite base* et *grande base*) ont la même médiatrice et si celle-ci est un axe de symétrie du trapèze. [1]

Pour le dessin, le caractère `'*'` est utilisé pour représenter les bases et les lignes intermédiaires, tandis que le caractère `'-'` est utilisé pour symboliser les espaces (note : les deux caractères ont tous les deux la même largeur). Le dessin final (composé de `'-'` et de `'*'`) aura une certaine largeur (cette largeur donne le nombre total de caractères `'-'` et `'*'` sur chaque ligne), également fournie par l'utilisateur. Le dessin du trapèze devra être centré par rapport à la largeur totale du dessin.

Pour vous simplifier la vie, vous pouvez considérer que les trois valeurs introduites par l'utilisateur (petite base, grande base, et largeur totale du dessin) sont des nombres impairs. En outre, la valeur entrée pour la grande base est toujours plus grande que la petite base et la largeur est toujours plus grande ou égale à la grande base (pas besoin de vérifier, vous-même, ces deux cas extrêmes).

Un exemple de résultat pour une petite base de 5, une grande base de 9 et une largeur de 11 est donné çi-dessous.

```
---*****---
--*******--
-*********-
```

Sablier . Un *sablier* est un instrument qui permet de mesurer un intervalle de temps correspondant à la durée d'écoulement d'une quantité calibrée de "sable", à l'intérieur d'un récipient transparent. Il s'agit ici de dessiner, sur la sortie standard, un sablier en utilisant le caractère espace (`'␣'`) et le caractère `'*'`.

Un sablier est composé de deux triangles *isocèles*. Le triangle supérieur est de sommet A et de base $[BC]$.

1. Il y a d'autres conditions possibles pour qu'un trapèze soit isocèle mais ces conditions ne nous intéressent pas dans cet exercice.

Pour chaque triangle, la hauteur issue de A doit être égale à n. La valeur de n est introduite au clavier par l'utilisateur. Un exemple d'exécution pour $n = 6$ est donné çi-dessous.

```
**********
 ********
  ******
   ****
    **
     *
     *
    **
   ****
  ******
 ********
**********
```

L'exécution de votre programme doit fournir, à l'utilisateur, un menu lui permettant de choisir la forme géométrique à dessiner ou de quitter le programme. Un exemple d'exécution avec le menu est donné çi-dessous :

```
$>./programme
Veuillez indiquer ce que vous souhaitez faire:
  - dessiner un rectangle ('a')
  - dessiner un triangle orienté vers le haut ('b')
  - dessiner un triangle orienté vers le bas ('c')
  - dessiner un trapèze ('d')
  - dessiner un sablier ('e')
  - quitter le programme ('q')
```

Pour résoudre cet exercice, il est impératif d'appliquer la méthodologie vue au cours (sous peine de ne pas y arriver) et de s'appuyer fortement sur la modularité (vous constaterez que les différentes formes géométriques partagent des fonctionnalités...). Il est donc demandé de procéder comme suit :

▷ **Exercice 1** Définissez proprement le problème en suivant le canevas vus au cours. Pensez à produire une représentation graphique de votre définition.

▷ **Exercice 2** Pour chaque forme géométrique, réfléchissez à quelles sont les interactions et relations entre les différents caractères composant la forme et les données en entrée pour la forme.

Pour cette étape, il est impératif de raisonner graphiquement (i.e., sur un dessin le plus général possible) et de jouer avec des couleurs pour mettre en évidence les éléments clés.

cette étape vous aidera pour la suivante.

▷ **Exercice 3** Analysez le problème et découpez le en SPs. Chaque SP devra être implémenté comme un module (fonction ou procédure).

▷ **Exercice 4** Fournissez les interfaces de chacun de vos modules (spécifications et signatures).

▷ **Exercice 5** Quand cela est pertinent, fournissez le Invariant Graphique pour chacun de vos modules. N'hésitez pas (si l'exercice ne se fait pas via CAFÉ) à vous appuyer sur le GLIDE pour construire et valider vos Invariants Graphiques

▷ **Exercice 6** Construisez le code de chaque module en suivant la méthode vue au cours (ZONE 1, Critère d'Arrêt, Fonction de Terminaison, ZONE 2, ZONE 3). Justifiez chaque étape sur base du Invariant Graphique (appuyez vous sur le GLIDE pour la justification si l'exercice n'est pas réalisé via CAFÉ).

▷ **Exercice 7** Testez votre code !

Faites appel à l'équipe pédagogique à la fin de chaque étape. L'équipe est aussi présente pour vous aider à régler une situation bloquante.

Pour rappel, vous disposez de $1h.$ pour résoudre les exercices de cette section.

Chapitre 6

Pointeurs

Le cœur de ce chapitre est la notion de *pointeur*, i.e., une variable qui contient non plus une valeur mais, plutôt, l'adresse d'une autre zone mémoire qui, elle, contient une valeur. La Sec. 6.1 va nous permettre de manipuler les pointeurs. La Sec. 6.2 va s'intéresser au passage de paramètres des modules. En particulier, nous verrons comment effectuer un passage de paramètres par *adresse* et, donc, comment un module (fonction ou procédure) peut renvoyer plus d'un résultat.

6.1 Arithmétique des Pointeurs

▷ **Exercice 1** Pour chaque programme ci-dessous :

1. donnez la représentation graphique de la mémoire à la fin du programme ;
2. indiquez ce qu'affiche le programme.

```c
#include <stdio.h>

int main(){
  int i = 3;
  int *p;
  p = &i;
  printf("*p␣=␣%d\n", *p);

  return 0;
}//fin programme
```

```c
#include <stdio.h>

int main(){
  int i = 3;
  int *p;
  p = &i;
  *p = 5;

  printf("i=%d,␣*p=%d\n", i, *p);
  i = 7;
  printf("i=%d,␣*p=%d\n", i, *p);
  return 0;
}//fin programme
```

▷ **Exercice 2** Commentez l'affichage du programme suivant, et comparez les valeurs des pointeurs.

```c
#include<stdio.h>
int main(){
  double var = 3.14;
  double *point_var, *pointeur2;
  int var2 = 5, *pointeur3;

  point_var = &var;
  pointeur2 = point_var+1;
  pointeur3 = &var2+1;

  printf("%d␣%d␣\n",sizeof(double),sizeof(int));
  printf("%p␣␣%p␣%p␣\n", point_var, &var, pointeur2);
  printf("%p␣%p␣\n",&var2,pointeur3);

  return 0;
}//fin programme
```

▷ **Exercice 3** Comparez ces deux programmes en donnant la représentation graphique de leur mémoire en fin d'exécution.

```c
int main(){
  int i = 3, j = 6;
  int *p1, *p2;
  p1 = &i;
  p2 = &j;
  *p1 = *p2;

  return 0;
}//fin programme
```

```c
int main(){
  int i = 3, j = 6;
  int *p1, *p2;

  p1 = &i;
  p2 = &j;
  p1 = p2;

  return 0;
}//fin programme
```

▷ **Exercice 4** Ce programme n'est pas correct. Corrigez-le.

```c
#include <stdio.h>

typedef struct complexe{
  float reel;
  float img;
}Complexe;

int main(){
  Complexe C, *p;

  C.reel = 3;
  C.img = 2;
  p = &C;

  printf("reel=%f,␣img=%f\n", p.reel, p.img);

  return 0;
```

```
}//fin programme
```

▷ **Exercice 5**

ligne(s)	a	b	c	p1	p2
2,3,4	1,5	3,5	/	/	/
6	1,5	3,5	/	&a	/
7					
8					
9					
10					
11					
12					

TABLE 6.1 – Exercice 5

Soit le programme suivant :

```
1  int main(){
2    float a = 1.5;
3    float b = 3.5;
4    float c, *p1, *p2;
5
6    p1 = &a;
7    *p1 *= 2;
8    p2 = &b;
9    c = 3 * (*p2 - *p1);
10   p1 = p2;
11   *p1 = 1.5;
12   (*p2)++;
13
14   return 0;
15 }//fin programme
```

Complétez le Tableau 6.1. Que se passe-t-il si on enlève les parenthèses à la ligne 12 du programme ?

▷ **Exercice 6** En sachant qu'un `int` est codé sur 4 bytes et qu'un `double` est codé sur 8 bytes et en supposant que `i` est stocké en mémoire à l'adresse 0xCDEF3210 [1], qu'affichent ces deux programmes ?

```
1  #include <stdio.h>
2
3  int main(){
4    int i = 3;
5    int *p1, *p2;
```

1. Les nombres qui débutent par `0x...` sont des nombres hexadécimaux, i.e. écrits en base 16, plus commode pour représenter des adresses (voir INFO0061).

```
6
7    p1 = &i;
8    p2 = p1 + 1;
9
10   printf("p1 = %p , p2 = %p\n", p1, p2);
11   return 0;
12 }//fin main()
```

```
1  #include <stdio.h>
2
3  int main(){
4    double i = 3;
5    double *p1, *p2;
6
7    p1 = &i;
8    p2 = p1 + 1;
9
10   printf("p1 = %p , p2 = %p\n", p1, p2);
11   return 0;
12 }//fin programme
```

6.2 Passage de Paramètres

▷ **Exercice 1** On a demandé à un étudiant d'écrire une fonction permettant d'échanger les valeurs de 2 variables. Voici le code, erroné, produit par l'étudiant :

```
1  void swap(int a, int b){
2    a ^= b;
3    b ^= a;
4    a ^= b;
5  }//fin swap()
```

Aidez l'étudiant à corriger sa solution.

▷ **Exercice 2** Soit le code suivant :

```
1  #include <stdio.h>
2
3  char fonc1(char a, char b){
4    a = 'P';
5    b = 'Q';
6
7    if(a<b)
8      return a;
9    else
10     return b;
11 }//fin fonc1()
12
13 char fonc2(char *c1, char *c2){
14   *c1 = 'P';
15   *c2 = 'Q';
16
17   if(*c1==*c2)
18     return *c1;
19   else
20     return *c2;
21 }//fin fonc2()
22
23 int main(){
24   char a = 'X';
```

```
  char b = 'Y';
  char i, j;

  i = fonc1(a, b);
  printf("a = %c, b = %c\n", a, b);
  j = fonc2(&a, &b);
  printf("a = %c, b = %c\n", a, b);

  return 0;
}//fin programme
```

Répondez à ces questions :

1. Quelle est la valeur affectée à `i` ?
2. Quelle est la valeur affectée à `j` ?
3. Qu'affiche ce programme ?

▷ **Exercice 3** Soit la fonction C suivante :

```
void modifie(int a, int *res){
  if(a>0)
    *res = a+1;
  else
    if(a<0)
      *res = a-1;
    else
      *res = a;
}//fin modifie()
```

Pour chacun des programmes ci-dessous :

1. indiquez les valeurs des variables à la fin du programme (si celui-ci est correct)
2. sinon expliquez pourquoi ce n'est pas correct (soyez précis avec le vocabulaire que vous utilisez).

```
int main(){
   int a = 3;
   int res;

   modifie(a, res);

   return 0;
}//fin programme
```

```
int main(){
   int a = 3;
   int res, *p;

   p = &res;
   modifie(a, p);

   return 0;
}//fin programme
```

```
int main(){
   int a = 3;

   modifie(a, a);

   return 0;
}//fin programme
```

```c
int main(){
   int a = 3;
   int res;

   modifie(a, &res);

   return 0;
}//fin programme
```

```c
int main(){
  int a = 3;
   int res, *p;

   p = &res;

   modifie(a, &p);

   return 0;
}//fin programme
```

```c
int main(){
   int a = 3;

   modifie(a, &a);

   return 0;
}//fin programme
```

▷ **Exercice 4** Soit la fonction `main()` suivante appelant une fonction `calcul()` qui réalise deux opérations : la somme et la différence de deux entiers.

```c
#include <stdio.h>

int main(){
  int a = 12, b = 45;
  int somme, difference;

  calcul(a, b, &somme, &difference);

  printf("a␣+␣b␣=␣%d,␣a␣-␣b␣=␣%d\n", somme, difference);

  return 0;
}//fin programme
```

Donnez le prototype de la fonction `calcul()` et, ensuite, écrivez le code de cette fonction.

Chapitre 7

Allocation Dynamique

Ce chapitre se concentre sur les structures de données *dynamiques*. Jusqu'à présent, les structures de données manipulées avaient un côté statique, e.g., la taille d'un tableau devait être définie directement dans le code et connue à la compilation. On va voir, dans ce chapitre, comment on peut créer (par exemple) des tableaux à la volée, en cours d'exécution du programme.

La Sec. 7.1 va être une première entrée dans la création, à la volée, de tableaux tandis que la Sec. 7.2 nous permettra de mettre les mains dans le cambouis.

7.1 Allocation Dynamique de Mémoire

▷ **Exercice 1** Ces programmes sont-ils corrects ? Expliquez.

```c
#include <stdlib.h>
int main(){
  int i, *p;

  i = 3;
  *p = 5;

  return 0;
}//fin programme
```

```c
#include <stdlib.h>
int main(){
  int i, *p;

  i = 3;
  p = &i;
  *p = 5;

  free(p);

  return 0;
}//fin programme
```

▷ **Exercice 2** Soit le programme suivant :

```c
#include <stdlib.h>

int main(){
  int **p;

  p = (int *)malloc(sizeof(int));
```

```
7   **p = 5;
8
9   free(p);
10
11  return 0;
12 }//fin programme
```

Dans ce programme, quel est le type de `p`, `*p`, `**p` ? En outre, ce programme est-il correct ? Justifiez.

7.2 Écriture de Code

Pour chacun des exercices de cette section, il vous est demandé de suivre avec rigueur la méthologie vue au cours. En particulier, la réponse à chacun des exercices devra comprendre (explicitement) les étapes suivantes :

1. définition du problème (i.e., Input, Output, Caractérisation des Inputs) ;
2. analyse du problème (i.e., découpe en SP– pensez à indiquer des noms pertinents pour chaque SP) ;
3. identification des SP que vous allez implémenter sous la forme de modules et pour chaque module, proposez une interface ;
4. pour chaque module nécessitant une boucle, proposez un Invariant Graphique en vous appuyant sur une représentation graphique de la POSTCONDITION. Confrontez votre Invariant Graphique avec les règles de conception. N'hésitez pas à vous appuyer sur le GLIDE pour la construction et la validation automatique de vos Invariants Graphiques ;
5. déterminez la Fonction de Terminaison ;
6. écriture code C de chaque module.

Attention, pour chaque segment de code impliquant une boucle (et donc un Invariant Graphique), il vous est demandé de :

- déterminer les variables nécessaires et leurs valeurs d'initialisation (i.e., ZONE 1). Justifiez sur base de l'Invariant Graphique ;
- déterminer le Critère d'Arrêt. Jusitifiez sur base de l'Invariant Graphique ;
- déterminer la Fonction de Terminaison. Justifier sur base de l'Invariant Graphique.

Faites valider chacune de ces étapes par un membre de l'équipe pédagogique avant de passer à l'étape suivante.

En outre, vous penserez à appliquer les principes de la programmation défensive (vérification des préconditions, retour des fonctions – e.g., retour du `malloc()` –, ...) chaque fois que cela s'avère pertinent.

▷ **Exercice 1** Soit une fonction `int rand(int min, int max)` qui retourne un nombre entier aléatoire compris dans l'intervalle [`min`, `max`]. Écrivez une fonction qui alloue un tableau d'entiers de taille donnée et qui remplit celui-ci avec des entiers aléatoires compris dans un intervalle donné.

La fonction demandée est déclarée comme suit :

```
1 int rand_tab(unsigned int n, int min, int max, int **dest);
```

où

- `n` est la taille du tableau,

- `min` et `max` sont les bornes inférieure et supérieure des nombres aléatoires à insérer dans le tableau,
- `dest` la destination du tableau généré.

La fonction retourne zéro si le tableau a été généré et -1 si ce n'est pas le cas (si `min` $>$ `max` ou `dest == NULL`).

▷ **Exercice 2** Le triangle de Pascal est une représentation des coefficients binomiaux dans un triangle. Le nombre à la ligne i et la colonne j tel que $0 \leq j \leq i$ est égal à C_i^j. Le triangle de Pascal peut être construit en utilisant les règles suivantes :

$$\begin{cases} C_i^0 &= C_i^i = 1 \text{ avec } i \geq 0 \\ C_n^p &= C_{n-1}^{p-1} + C_{n-1}^p \text{ avec } p \geq 1,\ n \geq 1,\ n-1 \geq p \end{cases}.$$

Écrivez un module qui génère un tableau triangulaire contenant les n premières lignes du triangle de Pascal. Celle-ci est déclarée comme suit :

```
1 int **tri_pas(unsigned int n);
```

où `n` est le nombre de lignes du triangle de Pascal à générer. La fonction retourne un tableau triangulaire de `n` lignes généré suivant la définition donnée ci-dessus.

▷ **Exercice 3** Soit une fonction `int f(int *x)` donnée. Cette fonction est capable de produire un nombre limité et a priori inconnu d'entiers. Lorsque `f` est appelée, si un entier est produit, il est stocké dans l'espace pointé par `x` et `f` retourne une valeur différente de zéro. Sinon, `f` retourne zéro et aucun appel suivant ne produira d'entier.

Écrivez un programme qui remplit, au moyen de `f`, un tableau d'entiers t de taille `n` donnée (`n` ≥ 0) à partir de l'indice zéro et tant que `f` produit un nombre. À la fin de l'exécution du programme, la variable `lim` contient le nombre d'entiers que `f` a produit (`lim` $\leq$ `n`).

▷ **Exercice 4** Soit la suite u_n définie par $u_0 = 1$ et $u_{n+1} = 3u_n^2 + 2u_n + 1$. Dans un premier temps, spécifiez et écrivez un module qui prend en paramètre un entier n et qui retourne un tableau contenant les n premiers termes de la suite u_n.

Dans un second temps, écrivez le programme principal qui va utiliser la fonction que vous venez de définir. L'entier n sera passé en paramètre de votre programme. Votre fonction principale devra donc récupérer l'argument passé. Ensuite, appeler la fonction de construction de la suite u_n et afficher les résultats à l'écran.

▷ **Exercice 5** Commencez par définir une structure de données permettant de gérer un étudiant. Un étudiant est représenté par son nom, son prénom, son matricule, 4 notes d'examen, sa moyenne, et son classement.

Spécifiez et écrivez ensuite un module qui remplit un tableau d'étudiants tant que l'utilisateur souhaite en ajouter et que le tableau n'est pas rempli. Attention, la moyenne et le classement ne sont pas des données à lire au clavier.

Spécifiez et écrivez ensuite un module qui prend en argument un tableau d'étudiants et qui calcule la moyenne de chaque étudiant ainsi que la moyenne de la promo.

Sur base de cette fonction, spécifiez et écrivez un module qui calcule le classement de chaque étudiant.

Enfin, spécifiez et écrivez un module qui réordonne les étudiants en fonction de leur classement (du premier au dernier).

▷ **Exercice 6** On veut organiser les fiches de renseignements des élèves dans un fichier.

1. Ecrivez un programme qui permet
   (a) la saisie et la sauvegarde des fiches dans un fichier nommé `Eleves.txt`. La fin de la saisie est possible si nous répondons par 'N' à la question "Voulez-vous saisir une nouvelle fiche ?".

(b) l'affichage des fiches.

2. Chaque fiche comporte les éléments suivants : nom, prénom, numéro de téléphone, âge et adresse. Proposez une structure de données permettant de manipuler une fiche.

▷ **Exercice 7** Dans cet exercice, il est demandé de :

1. Définir une structure de données appelée `Rectangle` qui contient trois champs réels : longueur, largeur et périmètre.
2. Demander à l'utilisateur combien de rectangle il veut définir. Soit `n` ce nombre.
3. Construire un tableau dynamique de rectangle, appelé `T` et de taille `n`.
4. Spécifier et écrire un module `calcul_perimetre` qui calcule le périmètre d'un rectangle donné.
5. Spécifier et écrire un module `creer_rectangle` qui demande à l'utilisateur la longueur et la largeur d'un rectangle et écrit dans `T` les valeurs saisies dans la première case vide de `T`.
6. Spécifier et écrire un module `modifier` qui transforme un rectangle en un rectangle plus petit en divisant sa longueur et sa largeur par deux.
7. Spécifier et écrire un module `affiche` qui affiche un rectangle.
8. Afficher tous les rectangles, avant et après modification.
9. Appliquer le module `modifier` à tous les rectangles du tableau.
10. Finalement, écrire le programme principal qui, en se servant des fonctions précédemment définies, demande à l'utilisateur de saisir les longueurs et largeurs de `n` rectangles, puis calcule leur périmètre, puis affiche les rectangles, les modifie, (il faut aussi mettre à jour leur périmètre) et effectue enfin un dernier affichage de tous ces rectangles.

▷ **Exercice 8** On considère un tableau à m lignes et n colonnes entrées dans un fichier. La première ligne du fichier contient les nombres m et n. Les lignes suivantes du fichier contiennent les coefficients du tableau. Les colonnes sont séparées par des espaces.

```
m n
a_0,0    a_0,1    a_0,2     ...   a_0,n-1
a_1,0    a_1,1    a_1,2     ...   a_1,n-1
...      ...      ....      ...   ...
a_m-1,0  a_m-1,1  a_m-1,2   ...   a_m-1,n-1
```

1. Spécifiez et écrivez un module d'allocation du tableau à deux dimensions de m lignes et n colonnes, les nombres m et n étant passés en paramètres.
2. Spécifiez et écrivez un module de libération de mémoire pour un tableau à deux dimensions.
3. Spécifiez et écrivez un module qui réalise le chargement du fichier dans un tableau de tableaux (ou matrice) A.
4. Spécifiez et écrivez un module qui calcule la somme des coefficients de chaque ligne de la matrice A et qui met les résultats dans un tableau de m nombres. Le module retournera ce tableau.
5. Écrivez le programme principal qui affiche la somme des coefficients de chaque ligne d'une matrice stockée dans un fichier, et libère la mémoire.

▷ **Exercice 9** Une *permutation* est un tableau T contenant chaque nombre de $0, 1, \ldots, n-1$ exactement une fois, et peut être interprétée de la façon suivante : T est une fonction envoyant l'élément i vers l'élément $T[i]$. L'*inverse* de cette permutation s'obtient en échangeant chaque élément de T avec sa position. Par exemple :

$$T = \begin{array}{|c|c|c|c|c|}\hline 0&1&2&3&4\\\hline 2&1&4&3&0\\\hline\end{array} \rightarrow \begin{array}{|c|c|c|c|c|}\hline 2&1&4&3&0\\\hline 0&1&2&3&4\\\hline\end{array} = \begin{array}{|c|c|c|c|c|}\hline 0&1&2&3&4\\\hline 4&1&0&3&2\\\hline\end{array} = T^{-1}$$

Ecrivez un module renvoyant l'inverse d'une permutation donnée.

▷ **Exercice 10** Le produit de deux permutations P et Q est une permuation exprimant la composée des fonctions représentées par P et Q appliquées dans cet ordre. Par exemple :

$$P = \begin{array}{|c|c|c|c|}\hline 0&1&2&3\\\hline 1&3&0&2\\\hline\end{array}, Q = \begin{array}{|c|c|c|c|}\hline 0&1&2&3\\\hline 2&1&3&0\\\hline\end{array} \Rightarrow P \times Q = \begin{array}{|c|c|c|c|}\hline 0&1&2&3\\\hline 1&0&2&3\\\hline\end{array}$$

Écrivez un module renvoyant le produit de deux permutations de même taille données.

▷ **Exercice 11** Écrivez un module renvoyant la matrice M d'une permutation P donnée, définie comme suit :

$$M[i][j] = \begin{cases} 1 & \text{si } P[i] = j \\ 0 & \text{sinon} \end{cases}.$$

Deuxième partie

Annexes – Fiches de Validation

Annexe 1

Définition du Problème – Fiche de Validation

Cette annexe présente plusieurs instances de la fiche de validation pour la définition d'un problème. Cette fiche est à utiliser dans le cadre des exercices de la Sec. 3.1.

Critères de l'Étape 1 (définition)	**Cohérent ?**	**Commentaires**
Input(s)	☐	
Output	☐	
Caractérisation des Inputs		
- font référence aux Inputs (et non aux variables propres à votre solution (variables d'itération, ...))	☐	
- ont un nom significatif	☐	
- ont un type existant en C	☐	
- ont une utilité clairement exprimée	☐	
Représentation Graphique		
- les Inputs sont indiqués	☐	
- les Inputs sont caractérisés	☐	
- l'Output est clairement indiqué	☐	

TABLE 1.1 – Fiche de validation pour l'étape 1 (Définition du Problème).

Critères de l'Étape 1 (définition)	**Cohérent ?**	**Commentaires**
Input(s)	□	
Output	□	
Caractérisation des Inputs		
- font référence aux Inputs (et non aux variables propres à votre solution (variables d'itération, ...))	□	
- ont un nom significatif	□	
- ont un type existant en C	□	
- ont une utilité clairement exprimée	□	
Représentation Graphique		
- les Inputs sont indiqués	□	
- les Inputs sont caractérisés	□	
- l'Output est clairement indiqué	□	

TABLE 1.2 – Fiche de validation pour l'étape 1 (Définition du Problème).

Critères de l'Étape 1 (définition)	**Cohérent ?**	**Commentaires**
Input(s)	□	
Output	□	
Caractérisation des Inputs		
- font référence aux Inputs (et non aux variables propres à votre solution (variables d'itération, ...))	□	
- ont un nom significatif	□	
- ont un type existant en C	□	
- ont une utilité clairement exprimée	□	
Représentation Graphique		
- les Inputs sont indiqués	□	
- les Inputs sont caractérisés	□	
- l'Output est clairement indiqué	□	

TABLE 1.3 – Fiche de validation pour l'étape 1 (Définition du Problème).

Critères de l'Étape 1 (définition)	**Cohérent ?**	**Commentaires**
Input(s)	□	
Output	□	
Caractérisation des Inputs		
- font référence aux Inputs (et non aux variables propres à votre solution (variables d'itération, ...))	□	
- ont un nom significatif	□	
- ont un type existant en C	□	
- ont une utilité clairement exprimée	□	
Représentation Graphique		
- les Inputs sont indiqués	□	
- les Inputs sont caractérisés	□	
- l'Output est clairement indiqué	□	

TABLE 1.4 – Fiche de validation pour l'étape 1 (Définition du Problème).

Annexe 2

Analyse de Problèmes – Fiche de Validation

Critères de l'Étape 2 (découpe)	**Cohérent ?**	**Commentaires**
La structuration des SPs est cohérente, complète et permet de résoudre le problème principal	□	
La structuration graphique des SPs suit les patterns.	□	
La granularité de la découpe est suffisante.	□	
SP ..... :		
- a un nom (et une description) explicits	□	
- répond à un des 4 types de SP	□	
- est générique (i.e., sa description ne se limite pas à certains exemples spécifiques)	□	
- a une description fonctionnelle et non technique[2]	□	
- vient avec une définition complète (Input, Output, C.I.)	□	

[1] Il ne s'agit **pas** de décrire litéralement des instructions (par exemple, *écrire une boucle* ou encore *incrémenter une variable* ne sont pas des SPs).

[2] Pour rappel, le bon niveau de granularité d'un SP est soit un module (`scanf()` ou `printf()`), soit une boucle.

TABLE 2.1 – Fiche de validation pour l'étape 2 (Analyse du Problème).

Critères de l'Étape 2 (découpe)	**Cohérent ?**	**Commentaires**
La structuration des SPs est cohérente, complète et permet de résoudre le problème principal	☐	
La structuration graphique des SPs suit les patterns.	☐	
La granularité de la découpe est suffisante.	☐	
SP ..... :		
- a un nom (et une description) explicits	☐	
- répond à un des 4 types de SP	☐	
- est générique (i.e., sa description ne se limite pas à certains exemples spécifiques)	☐	
- a une description fonctionnelle et non technique.	☐	
- vient avec une définition complète (Input, Output, C.I.)	☐	

TABLE 2.2 – Fiche de validation pour l'étape 2 (Analyse du Problème).

Critères de l'Étape 2 (découpe)	**Cohérent ?**	**Commentaires**
La structuration des SPs est cohérente, complète et permet de résoudre le problème principal	□	
La structuration graphique des SPs suit les patterns.	□	
La granularité de la découpe est suffisante.	□	
SP ..... :		
- a un nom (et une description) explicits	□	
- répond à un des 4 types de SP	□	
- est générique (i.e., sa description ne se limite pas à certains exemples spécifiques)	□	
- a une description fonctionnelle et non technique.	□	
- vient avec une définition complète (Input, Output, C.I.)	□	

TABLE 2.3 – Fiche de validation pour l'étape 2 (Analyse du Problème).

Critères de l'Étape 2 (découpe)	**Cohérent ?**	**Commentaires**
La structuration des SPs est cohérente, complète et permet de résoudre le problème principal	□	
La structuration graphique des SPs suit les patterns.	□	
La granularité de la découpe est suffisante.	□	
SP ..... :		
- a un nom (et une description) explicits	□	
- répond à un des 4 types de SP	□	
- est générique (i.e., sa description ne se limite pas à certains exemples spécifiques)	□	
- a une description fonctionnelle et non technique.	□	
- vient avec une définition complète (Input, Output, C.I.)	□	

TABLE 2.4 – Fiche de validation pour l'étape 2 (Analyse du Problème).

Critères de l'Étape 2 (découpe)	**Cohérent ?**	**Commentaires**
La structuration des SPs est cohérente, complète et permet de résoudre le problème principal	□	
La structuration graphique des SPs suit les patterns.	□	
La granularité de la découpe est suffisante.	□	
SP ..... :		
- a un nom (et une description) explicits	□	
- répond à un des 4 types de SP	□	
- est générique (i.e., sa description ne se limite pas à certains exemples spécifiques)	□	
- a une description fonctionnelle et non technique.	□	
- vient avec une définition complète (Input, Output, C.I.)	□	

TABLE 2.5 – Fiche de validation pour l'étape 2 (Analyse du Problème).

Critères de l'Étape 2 (découpe)	**Cohérent ?**	**Commentaires**
La structuration des SPs est cohérente, complète et permet de résoudre le problème principal	□	
La structuration graphique des SPs suit les patterns.	□	
La granularité de la découpe est suffisante.	□	
SP ..... :		
- a un nom (et une description) explicits	□	
- répond à un des 4 types de SP	□	
- est générique (i.e., sa description ne se limite pas à certains exemples spécifiques)	□	
- a une description fonctionnelle et non technique.	□	
- vient avec une définition complète (Input, Output, C.I.)	□	

TABLE 2.6 – Fiche de validation pour l'étape 2 (Analyse du Problème).

Annexe 3

Construction par Invariant Graphique– Fiche de Validation

Critères généraux	**Cohérent ?**	**Commentaires (référés par des tags si besoin)**
Chaque SP avec une boucle a un Invariant Graphique correspondant	□	
Références au SP dont on parle présentes	□	
Invariants Graphiques présentés cohérents	□	

TABLE 3.1 – Critères généraux pour l'étape 3 (Invariant Graphique).

Règles de l'Invariant Graphique	**Cohérent ?**	**Commentaires (référés par des tags si besoin)**
Le nom de la structure est approprié (Règle 1)	☐	
Le dessin est approprié (Règle 1)	☐	
Borne minimale (Règle 2)	☐	
Borne maximale (Règle 2)	☐	
Ligne de démarcation (Règle 3)	☐	
Variable d'itération (Règle 4)	☐	
La caractérisation des variables et la description de ce qui a été réalisé jusqu'à présent (Règle 5) sont :		
- cohérentes	☐	
- complètes[1]	☐	
- précises	☐	
Zone "à faire" (Règle 6)	☐	
Propriétés qui sont conservées (Règles 5+6)	☐	

TABLE 3.2 – Fiche de validation pour l'étape 3 (Invariant Graphique).

Critères généraux	**Cohérent ?**	**Commentaires (référés par des tags si besoin)**
Chaque Sp avec une boucle a un Invariant Graphique correspondant	☐	
Références au Sp dont on parle présentes	☐	
Invariants Graphiques présentés cohérents	☐	

Table 3.3 – Critères généraux pour l'étape 3 (Invariant Graphique).

Règles de l'Invariant Graphique	**Cohérent ?**	**Commentaires (référés par des tags si besoin)**
Le nom de la structure est approprié (Règle 1)	☐	
Le dessin est approprié (Règle 1)	☐	
Borne minimale (Règle 2)	☐	
Borne maximale (Règle 2)	☐	
Ligne de démarcation (Règle 3)	☐	
Variable d'itération (Règle 4)	☐	
La caractérisation des variables et la description de ce qui a été réalisé jusqu'à présent (Règle 5) sont :		
- cohérentes	☐	
- complètes[1]	☐	
- précises	☐	
Zone "à faire" (Règle 6)	☐	
Propriétés qui sont conservées (Règles 5+6)	☐	

TABLE 3.4 – Fiche de validation pour l'étape 3 (Invariant Graphique).

Critères généraux	**Cohérent ?**	**Commentaires (référés par des tags si besoin)**
Chaque Sp avec une boucle a un Invariant Graphique correspondant	☐	
Références au Sp dont on parle présentes	☐	
Invariants Graphiques présentés cohérents	☐	

Table 3.5 – Critères généraux pour l'étape 3 (Invariant Graphique).

Règles de l'Invariant Graphique	**Cohérent ?**	**Commentaires (référés par des tags si besoin)**
Le nom de la structure est approprié (Règle 1)	☐	
Le dessin est approprié (Règle 1)	☐	
Borne minimale (Règle 2)	☐	
Borne maximale (Règle 2)	☐	
Ligne de démarcation (Règle 3)	☐	
Variable d'itération (Règle 4)	☐	
La caractérisation des variables et la description de ce qui a été réalisé jusqu'à présent (Règle 5) sont :		
- cohérentes	☐	
- complètes[1]	☐	
- précises	☐	
Zone "à faire" (Règle 6)	☐	
Propriétés qui sont conservées (Règles 5+6)	☐	

TABLE 3.6 – Fiche de validation pour l'étape 3 (Invariant Graphique).

Critères généraux	**Cohérent ?**	**Commentaires (référés par des tags si besoin)**
Chaque Sp avec une boucle a un Invariant Graphique correspondant	□	
Références au Sp dont on parle présentes	□	
Invariants Graphiques présentés cohérents	□	

Table 3.7 – Critères généraux pour l'étape 3 (Invariant Graphique).

Règles de l'Invariant Graphique	**Cohérent ?**	**Commentaires (référés par des tags si besoin)**
Le nom de la structure est approprié (Règle 1)	☐	
Le dessin est approprié (Règle 1)	☐	
Borne minimale (Règle 2)	☐	
Borne maximale (Règle 2)	☐	
Ligne de démarcation (Règle 3)	☐	
Variable d'itération (Règle 4)	☐	
La caractérisation des variables et la description de ce qui a été réalisé jusqu'à présent (Règle 5) sont :		
- cohérentes	☐	
- complètes[1]	☐	
- précises	☐	
Zone "à faire" (Règle 6)	☐	
Propriétés qui sont conservées (Règles 5+6)	☐	

TABLE 3.8 – Fiche de validation pour l'étape 3 (Invariant Graphique).

Critères généraux	**Cohérent ?**	**Commentaires (référés par des tags si besoin)**
Chaque Sp avec une boucle a un Invariant Graphique correspondant	☐	
Références au Sp dont on parle présentes	☐	
Invariants Graphiques présentés cohérents	☐	

Table 3.9 – Critères généraux pour l'étape 3 (Invariant Graphique).

Règles de l'Invariant Graphique	**Cohérent ?**	**Commentaires (référés par des tags si besoin)**
Le nom de la structure est approprié (Règle 1)	☐	
Le dessin est approprié (Règle 1)	☐	
Borne minimale (Règle 2)	☐	
Borne maximale (Règle 2)	☐	
Ligne de démarcation (Règle 3)	☐	
Variable d'itération (Règle 4)	☐	
La caractérisation des variables et la description de ce qui a été réalisé jusqu'à présent (Règle 5) sont :		
- cohérentes	☐	
- complètes[1]	☐	
- précises	☐	
Zone "à faire" (Règle 6)	☐	
Propriétés qui sont conservées (Règles 5+6)	☐	

TABLE 3.10 – Fiche de validation pour l'étape 3 (Invariant Graphique).

Critères généraux	**Cohérent ?**	**Commentaires (référés par des tags si besoin)**
Chaque Sp avec une boucle a un Invariant Graphique correspondant	□	
Références au Sp dont on parle présentes	□	
Invariants Graphiques présentés cohérents	□	

Table 3.11 – Critères généraux pour l'étape 3 (Invariant Graphique).

Règles de l'Invariant Graphique	**Cohérent ?**	**Commentaires (référés par des tags si besoin)**
Le nom de la structure est approprié (Règle 1)	☐	
Le dessin est approprié (Règle 1)	☐	
Borne minimale (Règle 2)	☐	
Borne maximale (Règle 2)	☐	
Ligne de démarcation (Règle 3)	☐	
Variable d'itération (Règle 4)	☐	
La caractérisation des variables et la description de ce qui a été réalisé jusqu'à présent (Règle 5) sont :		
- cohérentes	☐	
- complètes[1]	☐	
- précises	☐	
Zone "à faire" (Règle 6)	☐	
Propriétés qui sont conservées (Règles 5+6)	☐	

TABLE 3.12 – Fiche de validation pour l'étape 3 (Invariant Graphique).

Critères généraux	**Cohérent ?**	**Commentaires (référés par des tags si besoin)**
Chaque SP avec une boucle a un Invariant Graphique correspondant	☐	
Références au SP dont on parle présentes	☐	
Invariants Graphiques présentés cohérents	☐	

TABLE 3.13 – Critères généraux pour l'étape 3 (Invariant Graphique).

Règles de l'Invariant Graphique	**Cohérent ?**	**Commentaires (référés par des tags si besoin)**
Le nom de la structure est approprié (Règle 1)	☐	
Le dessin est approprié (Règle 1)	☐	
Borne minimale (Règle 2)	☐	
Borne maximale (Règle 2)	☐	
Ligne de démarcation (Règle 3)	☐	
Variable d'itération (Règle 4)	☐	
La caractérisation des variables et la description de ce qui a été réalisé jusqu'à présent (Règle 5) sont :		
- cohérentes	☐	
- complètes[1]	☐	
- précises	☐	
Zone "à faire" (Règle 6)	☐	
Propriétés qui sont conservées (Règles 5+6)	☐	

TABLE 3.14 – Fiche de validation pour l'étape 3 (Invariant Graphique).

Critères généraux	**Cohérent ?**	**Commentaires (référés par des tags si besoin)**
Chaque Sp avec une boucle a un Invariant Graphique correspondant	□	
Références au Sp dont on parle présentes	□	
Invariants Graphiques présentés cohérents	□	

Table 3.15 – Critères généraux pour l'étape 3 (Invariant Graphique).

Règles de l'Invariant Graphique	**Cohérent ?**	**Commentaires (référés par des tags si besoin)**
Le nom de la structure est approprié (Règle 1)	☐	
Le dessin est approprié (Règle 1)	☐	
Borne minimale (Règle 2)	☐	
Borne maximale (Règle 2)	☐	
Ligne de démarcation (Règle 3)	☐	
Variable d'itération (Règle 4)	☐	
La caractérisation des variables et la description de ce qui a été réalisé jusqu'à présent (Règle 5) sont :		
- cohérentes	☐	
- complètes[1]	☐	
- précises	☐	
Zone "à faire" (Règle 6)	☐	
Propriétés qui sont conservées (Règles 5+6)	☐	

TABLE 3.16 – Fiche de validation pour l'étape 3 (Invariant Graphique).

Critères généraux	**Cohérent ?**	**Commentaires (référés par des tags si besoin)**
Chaque Sp avec une boucle a un Invariant Graphique correspondant	☐	
Références au Sp dont on parle présentes	☐	
Invariants Graphiques présentés cohérents	☐	

Table 3.17 – Critères généraux pour l'étape 3 (Invariant Graphique).

Règles de l'Invariant Graphique	**Cohérent ?**	**Commentaires (référés par des tags si besoin)**
Le nom de la structure est approprié (Règle 1)	☐	
Le dessin est approprié (Règle 1)	☐	
Borne minimale (Règle 2)	☐	
Borne maximale (Règle 2)	☐	
Ligne de démarcation (Règle 3)	☐	
Variable d'itération (Règle 4)	☐	
La caractérisation des variables et la description de ce qui a été réalisé jusqu'à présent (Règle 5) sont :		
- cohérentes	☐	
- complètes[1]	☐	
- précises	☐	
Zone "à faire" (Règle 6)	☐	
Propriétés qui sont conservées (Règles 5+6)	☐	

TABLE 3.18 – Fiche de validation pour l'étape 3 (Invariant Graphique).

Critères généraux	**Cohérent ?**	**Commentaires (référés par des tags si besoin)**
Chaque Sp avec une boucle a un Invariant Graphique correspondant	□	
Références au Sp dont on parle présentes	□	
Invariants Graphiques présentés cohérents	□	

Table 3.19 – Critères généraux pour l'étape 3 (Invariant Graphique).

Règles de l'Invariant Graphique	**Cohérent ?**	**Commentaires (référés par des tags si besoin)**
Le nom de la structure est approprié (Règle 1)	☐	
Le dessin est approprié (Règle 1)	☐	
Borne minimale (Règle 2)	☐	
Borne maximale (Règle 2)	☐	
Ligne de démarcation (Règle 3)	☐	
Variable d'itération (Règle 4)	☐	
La caractérisation des variables et la description de ce qui a été réalisé jusqu'à présent (Règle 5) sont :		
- cohérentes	☐	
- complètes[1]	☐	
- précises	☐	
Zone "à faire" (Règle 6)	☐	
Propriétés qui sont conservées (Règles 5+6)	☐	

TABLE 3.20 – Fiche de validation pour l'étape 3 (Invariant Graphique).